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

SISTEMAS DE INFORMAÇÃO DE ESCRITÓRIOS: UM MODELO PARA ESPECIFICAÇÕES TEMPORAIS

por

Nina Edelweiss

Tese submetida como requisito parcial
para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Dr. José Palazzo Moreira de Oliveira
Orientador

Porto Alegre, junho de 1994.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Edelweiss, Nina

Sistemas de Informação de Escritórios: um Modelo para Especificações Temporais / Nina Edelweiss. - Porto Alegre: CPGCC da UFRGS, 1994

187 p. : il.

Tese (doutorado) - Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1994. Orientador:Oliveira, José Palazzo Moreira de.

Tese: Sistemas de Informação de Escritórios, Especificações Formais, Modelagem Temporal, Orientação a Objetos, Reutilização de Especificações.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Sistema de Biblioteca da UFRGS

[redacted]

681.32.074(043)
E22S

INF
1994/250596-1
1994/09/19

MOD. 2.3.2

Para

Carlos,

Marcos, Flávio e Roberta.

AGRADECIMENTOS

Este trabalho é o resultado não só de alguns anos de curso de doutorado, mas de tudo o que até hoje vivi. O conhecimento se constrói lentamente, fruto de esforço próprio mas influenciado por fatos e pessoas. Somente a passagem do tempo - este fator tão importante - permite que o conhecimento seja sedimentado e que evolua para um trabalho completo.

É tarefa impossível compor uma lista completa de agradecimentos, na qual constem os nomes de todas as pessoas que foram importantes para o desenvolvimento deste trabalho. São tantos anos vividos e tantas pessoas a serem lembradas! Mesmo procurando citar somente aqueles mais intimamente ligados a este trabalho específico, seguramente estarei sendo injusta com alguns. Que estes me perdoem.

Agradeço:

- ao colega e amigo *José Palazzo Moreira de Oliveira* que tão bem soube orientar o desenvolvimento desta tese;
- à professora *Barbara Pernici* que me acolheu na Universidade de Udine, na Itália, onde iniciou o desenvolvimento desta tese, por sua atenção, dedicação e amizade;
- ao amigo *José Mauro Volkmer de Castilho*, pelo importante auxílio na definição da parcela de lógica temporal deste trabalho;
- a todos os professores e amigos do Instituto de Informática, que sempre me incentivaram e auxiliaram nesta tão longa jornada;
- aos amigos *Daltro José Nunes* e *Paulo Alberto de Azeredo*, pelo apoio quando de minha transferência da UFSC para a UFRGS;
- à amiga *Maria Lúcia Blanck Lisboa*, a quem devo o retorno à vida acadêmica após um longo afastamento, pela sua confiança e incentivo;
- a meus pais - *Erna* e *Werner Krahe* - por todos os ensinamentos que me passaram e por terem sempre incentivado minha carreira acadêmica, especialmente nos muitos momentos em que esta meta parecia estar distante demais;
- a meus maravilhosos filhos *Roberta*, *Flávio* e *Marcos*, dos quais roubei tanta atenção e cuidados, sendo sempre retribuída com entusiasmo e apoio. Deles tenho hoje muito a aprender, mais do que lhes ensinei, com suas idéias tão lindas de humanidade, de justiça e de amor;
- e principalmente a meu marido e melhor amigo *Carlos*, pelo seu apoio, compreensão, estímulo, exemplo e amor. A qualidade de nossa convivência durante estes muitos anos mostra que a vida a dois dá um sentido todo especial à nossa existência.

ITHACA

When you start on the way to Ithaca,
Wish that the way be long,
Full of adventure, full of knowledge,
The Laestrygones and the Cyclopes
And angry Poseidon, do not fear;
Such, on your way, you shall never meet
If your thougts are lofty, if a noble
Emotion touch your mind, your body,
The Laestrygones and the Cyclopes
The angry Poseidon you shall not meet
If you carry them not in your soul,
If your soul sets them not up before you.

Wish that the way be long,
That on many summer mornings,
With great pleasure, great delight,
You enter harbours for the first time seen;
That you stop at Phoenician marts
And procure the goodly merchandise,
Mother-of-pearl and corals, amber and ebony,
And sensual perfumes of all kinds,
Plenty of sensual perfumes especially;
To wend your way to many Egyptian cities,
To learn and yet to learn from the wise.

Ever keep Ithaca in your mind,
your return thither is your voyage,
But do not hasten at all your voyage,
Better that it lasts for many years;
And full of years at length you anchor at your isle
Rich with all that you have gained on the way;
Do not expect Ithaca to give you riches.

Ithaca gave you your fair voyage.
Without her you would not have ventured on the way.
But she has no more to give you.

And if you find Ithaca a poor place, she has not mocked you.
You have become so wise, so full of experience
That you have understood already what these Ithacas mean.

Poema de C. Kavafis
Tradução de G. Seferis

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Sistemas de Informação de Escritórios são tipos particulares de sistemas de informação. São sistemas sócio-técnicos, muito complexos, com grande influência humana. O tempo tem grande importância no tratamento das informações, tanto na representação de informações temporais explícitas como em restrições de ordem temporal e em características que são alteradas com a evolução da aplicação.

Neste trabalho é apresentado um modelo de dados para ser utilizado como método de especificação de requisitos de sistemas de informação de escritórios. O desenvolvimento de especificações é uma tarefa bastante complexa, devendo possibilitar a representação de todas as características da aplicação, tanto as estáticas como as comportamentais.

A especificação de uma aplicação geralmente é muito extensa, sendo dispendido um tempo considerável em sua elaboração. Um aspecto importante no desenvolvimento de especificações é a verificação da corretude destas, devido à complexidade que apresentam. Uma maneira de gerar especificações mais corretas e em menor espaço de tempo é através da reutilização de especificações anteriormente construídas e já validadas através de implementações.

Neste trabalho optou-se por utilizar um modelo de dados orientado a objetos para especificar os sistemas de informação de escritórios. A utilização do paradigma de orientação a objetos na modelagem permite a definição de uma biblioteca de classes de objetos, classes estas identificadas em diversas especificações realizadas em um determinado domínio de aplicação. As classes constantes desta biblioteca podem ser reutilizadas de maneira bastante eficiente em especificações posteriores.

A representação das características dinâmicas de uma aplicação, tais como a evolução dos objetos dentro do escritório, requer a possibilidade de representação de propriedades temporais. O modelo de dados utilizado na especificação deve permitir a representação de aspectos temporais tanto para definição de dados definidos em um domínio temporal como para representar a evolução dos valores assumidos pelos objetos durante sua existência. A possibilidade de representação dos aspectos temporais é um dos principais pontos desenvolvidos na elaboração do modelo de dados apresentado neste trabalho.

O modelo apresentado denomina-se TF-ORM (*Temporal Fuctionality in Objects with Roles Model*), sendo um extensão do modelo F-ORM (*Functionality in Objects with Roles Model*) [DEA 91a,b,c]. Foram incorporadas a este modelo os necessários aspectos temporais e foram ampliadas características básicas para captar melhor as particularidades do domínio de sistemas de informação de escritórios, principalmente no que concerne à representação de alguma parcela de trabalho humano. O modelo resultante, TF-ORM, é um modelo de dados orientado a objetos, temporal, que utiliza o conceito de papéis para representar os diferentes comportamentos de um objeto.

A possibilidade de recuperação de informações de um banco de dados que implemente o modelo de dados proposto foi também considerada. Neste trabalho é

apresentada uma linguagem de recuperação de informações (linguagem de consulta) para o modelo de dados TF-ORM. Especial atenção é dada às consultas temporais. Como no modelo proposto são armazenados tanto o tempo de transação como o de validade, a linguagem de recuperação pode ser utilizada para recuperar informações referentes ao estado atual do banco de dados (informações atualmente válidas), a respeito de estados passados e futuros (informações válidas no passado e que serão válidas em estados futuros, de acordo com o atual conhecimento dos dados) e referentes a histórias passadas do banco de dados (informações que se acreditava como válidas em algum momento do passado).

É apresentado em estudo de caso completo com o objetivo de validar o modelo de dados proposto e sua eficiência na especificação deste tipo de sistema de informação.

O trabalho apresenta ainda a descrição de um ambiente de apoio a especificações que faz uso de uma biblioteca de classes. Através deste ambiente, a especificação de uma aplicação é construída gradualmente, estando disponíveis opções de listagem das partes já definidas e de informações da biblioteca de classes para serem reutilizadas. As especificações geradas através deste ambiente apresentam pouca possibilidade de erro, uma vez que são efetuados vários procedimentos de verificação das informações fornecidas.

PALAVRAS-CHAVE: Sistemas de Informação de Escritórios, Especificações Formais, Modelagem Temporal, Orientação a Objetos, Reutilização de Especificações.

Title: Office Information Systems: a Model for Temporal Specifications

ABSTRACT

Office Information Systems constitute special kinds of information systems - very complex socio-technical systems, with large human influence. Time is very important in the information representation, not only for explicit temporal information but also for restrictions on temporal ordering of activities to be executed and to record the temporal evolution of property values.

In this work a data model to be used as a requirement specification method for office informations systems is presented. Specification development is a complex task. All the possible characteristics of an application, static and behavioral, shall be represented.

An applications' specification is usually not a trivial task, and a big amount of time is used to develop it. Specifications are usually very complex and the possibility of verifying the correctness is an important aspect to be considered. The reuse of specifications is a way to obtain a better rate of correctness spending less time, using parts of specifications that were already validated through previous implementations.

An object-oriented data model is used in this work to specify office information systems. The use of the object-oriented paradigm allows the definition of a class library containing classes identified in several specifications constructed in a specific application domain. The reuse of classes of such a library in new specifications gives efficiency to the process.

To represent the dynamic characteristics of an application, such as the documents evolution within the office, it is necessary to be able to represent temporal properties. The data model used in the specification shall allow the representation of these temporal aspects not only to model data defined in a temporal domain but also to represent the objects values evolution during an objects existence. The representaion of temporal aspects is one of the main issues developed in the data model defined in this work.

The data model is called TF-ORM (*Temporal Fuctionality in Objects with Roles Model*), and is an extention to the F-ORM (*Functionality in Objects with Roles Model*) model [DEA 91a,b,c]. To this model the necessary temporal aspects were added and the basic characteristics were enlarged to better capture the particular aspects of the office information systems domain, specially concerning the representation of some parts of human work. The resulting model, TF-ORM, is an object-oriented temporal data model and uses the concept of roles to represent an objects different behaviors.

The possibility of retrieving information from a database specified through this data model is also considered and a query language for the TF-ORM data model is presented. Special attention is given to the retrieval of temporal information. The TF-ORM data model stores both the transaction and the valid time associated to dynamic properties and allows the retrieving of information reffering the databases' actual state (actually valid information), information refering the databases' past and future states

(valid information in the past and information that will be valid in the future according to the actual data knowledge) and information about the databases' past history (information that was believed to be true in some moment in the past).

A complete case study is presented to validate the TF-ORM data model and its efficiency as a specification method for office information systems.

An environment to support specifications using TF-ORM is described. This environment uses a class library. The use of this environment allows stepwise construction of a specification, with the access to listings of the already defined parts and to the information of the class library to be reused. The contructed specifications present a small rate of errors, due to several verification procedures embedded in the environment.

KEYWORDS: Office Information Systems, Formal Specifications, Temporal Modeling, Object Orientation, Specification Reuse.

1.INTRODUÇÃO

A implementação de um sistema de informação é uma tarefa bastante complexa. Diversos ciclos de vida do desenvolvimento de um sistema de informação foram propostos na literatura [BOE 88, GIR 90, HEN 90], apresentando diferentes fases a serem executadas e algumas diferenças entre a ordem de execução entre estas fases. Uma fase, entretanto, aparece em todos eles: a especificação dos requisitos do sistema a ser implementado. A especificação dos requisitos se baseia em uma coleta dos requisitos no mundo real, procurando identificar e descrever todas as características que o futuro sistema deverá apresentar. Deve ser totalmente independente da implementação, servindo de base para a realização da implementação.

Diversos métodos de especificação foram desenvolvidos, procurando captar, da melhor maneira possível, todos os aspectos a serem definidos. Dois tipos distintos de métodos de especificação podem ser identificados: métodos algébricos (como, por exemplo, especificações algébricas [KLA 83, KOH 86, WIR 89]) e métodos que se baseiam em modelos da aplicação a ser especificada (como o método VDM [BJO 89, KOH 86, JON 86]). Alguns métodos de especificação apresentam inclusive a possibilidade de uma verificação formal da especificação, como o método OBJ que serve para definir e testar especificações algébricas [GOG 78a,b, GOG 82]. Métodos formais são geralmente muito complexos, difíceis de serem utilizados por pessoas não especializadas. Alguns utilizam meios gráficos com o objetivo de facilitar o entendimento por parte de pessoas não especializadas, como o sistema OSIRIS [BAR 85], o método REMORA [ROL 82] e diversos métodos que utilizam redes de Petri [REI 85] (como, por exemplo, o método GOFOR [RIC 87a]). Entretanto, a modelagem de sistemas complexos dá origem a gráficos muito amplos, tornando-se difícil a sua compreensão.

Uma forma bastante usual de especificação dos requisitos é através da definição do esquema que será implementado em um banco de dados que modele o sistema de informação. Podem ser utilizados o modelo de dados correspondente a um banco de dados relacional ou modelos de dados não convencionais, tais como os dos sistemas IRIS [FIS 87, LYN 84, 86a,b], OFFIS [BRA 84, KON 82], INSYDE [KIN 85] ou TODOS [HEI 88, PER 88]. Desta maneira a passagem da fase de especificação para a implementação em um banco de dados fica bastante simplificada. Dentre os diferentes modelos de dados que foram desenvolvidos, os modelos de dados orientados a objetos têm recebido grande atenção dos meios de pesquisa atuais, como por exemplo, o modelo de dados do sistema ORION [BAN 87, GAR 88] e o do sistema O_2 [BAN 88, LÉC 88]. Os modelos de dados orientados a objetos apresentam uma característica que permite um aumento da qualidade e da eficiência das especificações geradas: a possibilidade de reutilização de partes de especificações anteriormente realizadas e já validadas. Esta reutilização é feita através da utilização de uma biblioteca de classes onde estão armazenadas classes de objetos.

Sistemas de Informação de Escritórios (SIEs) são tipos particulares de sistemas de informação. São sistemas sócio-técnicos, muito complexos, com grande influência humana. O tempo tem grande importância no tratamento das informações, tanto na representação de informações temporais explícitas como em restrições de ordem temporal

e em características que são alteradas com a evolução da aplicação. Sofrem muita influência dos avanços tecnológicos que devem ser absorvidos pelos sistemas implementados, com o risco de torná-los obsoletos. A especificação destes sistemas deve, por isso, ser independente da implementação, permitindo alterações da tecnologia utilizada.

A especificação de um SIE deve levar em conta a influência humana, apresentando não só a estrutura das informações que serão manipuladas, mas também a semântica do ambiente em que são desenvolvidas as aplicações. Deve, ainda, representar as características temporais, representadas através de tipos de dados definidos em domínios temporais assim como de execução de processos interdependentes. Métodos de especificação que se baseiam em modelos de dados utilizam este modelo de dados como ferramenta de especificação da semântica do escritório.

Vários métodos de especificação têm sido propostos para sistemas complexos, os quais incluem SIEs. Todos os sistemas anteriormente exemplificados são adequados para este tipo de aplicações. A especificação de atividades pouco estruturadas e não estruturadas, nas quais o influência humana é predominante, entretanto, tem sido pouco considerada. Como exemplo de sistemas que representam alguma atividade pouco estruturada podemos citar os sistemas KNOs [CAS 88, TSI 87] e AMS [TUE 88].

Alguns ambientes de desenvolvimento de software têm sido construídos para auxiliar no desenvolvimento de SIE como por exemplo, o ambiente TODOS [HEI 88]. Apresentam ferramentas para apoiar cada uma das fases deste desenvolvimento, alguns de forma integrada, outros não. A implementação destes ambientes envolve pesquisas nas áreas de Bancos de Dados, Inteligência Artificial e Engenharia de Software, além da área específica de SIEs.

O objetivo principal deste trabalho é a definição de um modelo de dados a ser utilizado em especificações de SIEs. Este modelo de dados deverá apresentar características próprias à reutilização de especificações anteriormente realizadas, representar os aspectos temporais da aplicação e levar em consideração alguma parcela de trabalho caracteristicamente humano.

De uma análise dos modelos de dados já existentes foi selecionado um modelo de dados orientado a objetos específico para SIEs, o modelo F-ORM (*Functionality in Objects with Roles Model*) [DEA 91a,b,c], o qual utiliza o conceito de papéis para permitir a representação de diferentes comportamentos de um objeto. Este modelo não apresenta, entretanto, a possibilidade de representação de aspectos temporais nem de atividades que resultem em decisões humanas, ambos fatores considerados importantes para a completa representação das atividades destes ambientes. Decidiu-se, então, basear o modelo proposto neste selecionado, estendendo-o de modo a captar estes aspectos e detalhando outros considerados como importantes para uma melhor representação de aplicações de SIEs. O modelo resultante foi denominado **modelo de dados TF-ORM (*Temporal Functionality in Objects with Roles Model*)**, ressaltando a importância da possibilidade de representação de aspectos temporais.

A representação de restrições temporais e da dimensão temporal são aspectos essenciais em diversas aplicações de SIE, como por exemplo na representação de trabalho cooperativo. Sistemas de bancos de dados tradicionais suportam os aspectos temporais somente através de programas de aplicação, não apresentando aspectos temporais no seu modelo de dados. A tendência atual é a de representar os aspectos temporais diretamente no esquema conceitual do banco de dados e na correspondente linguagem de recuperação de informações, sendo eles assim suportados diretamente pelo sistema gerenciador do banco de dados. Para isto os aspectos temporais devem estar embutidos no modelo de dados utilizado na especificação e posterior implementação do banco de dados.

Ferramentas de apoio à especificação auxiliam sobremaneira a construção de especificações, gerando produtos mais corretos e de maneira mais rápida. O modelo de dados utilizado como método de especificação deverá apresentar uma ferramenta de apoio à especificação que implemente o paradigma de reutilização de especificações anteriormente realizadas através da utilização de uma biblioteca de classes.

A possibilidade de manipular e de recuperar informações de um banco de dados é um fator importante em sua especificação. A completa definição de um modelo de dados requer que seja também definida um linguagem de recuperação de informações correspondente a ele. Considerando modelos de dados orientados a objetos, uma linguagem de recuperação de informações deve levar em consideração vários aspectos próprios deste paradigma. Outro aspecto desenvolvido neste trabalho foi a definição de uma linguagem de recuperação de informações para o modelo de dados proposto, denominada **linguagem TF-ORM**. Nesta linguagem devem ser considerados tanto os 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.

A seguir é descrita a forma como está estruturado este trabalho.

O capítulo 2 analisa especificações de sistemas de informação de escritórios. Inicialmente são apresentados os sistemas de informação de escritórios e são vistos os diferentes tipos de atividades que podem ser desenvolvidas nestas aplicações, as estruturas dos dados e os tipos de pessoas envolvidas no desenvolvimento das tarefas. Em seguida a especificação de requisitos é situada no ciclo de desenvolvimento de um sistema de informação de escritório e algumas considerações a respeito de métodos de especificação de requisitos são feitas.

Devido à grande importância que apresentam neste tipo de aplicações, os aspectos temporais que devem ser representados na especificação de uma aplicação são analisados detalhadamente no capítulo 3. São analisados os requisitos temporais de sistemas de informação de escritórios e proposta uma taxonomia para eles. Os itens relevantes e as diferentes formas de representação temporal são apresentados a seguir.

No capítulo 4 são feitas algumas considerações a respeito do paradigma de reutilização aplicado à nível de especificação. É analisada a utilização de modelos de dados orientados a objetos com o objetivo de propiciar a reutilização de especificações.

No capítulo 5 são apresentados os modelos de dados ORM e F-ORM, modelos preliminares dos quais resultou o modelo de dados TF-ORM proposto neste trabalho.

O modelo de dados TF-ORM é apresentado no capítulo 6. São vistos os tipos de classes definidas, as propriedades pré-definidas que armazenam informações a respeito da vida de instâncias, os conceitos envolvidos nas abstrações de agregação e de especialização, as formas utilizadas para representar as informações temporais e como pode ser efetivada a reutilização de especificações através da utilização deste modelo de dados em especificações. A sintaxe completa da linguagem de definição de dados do modelo TF-ORM é apresentada no Anexo 1.

No capítulo 7 é analisada a possibilidade de recuperação de informações em bancos de dados temporais. São inicialmente analisados os diferentes tipos de recuperações que podem ser efetuados e é proposta uma taxonomia para as recuperações em bancos de dados temporais baseada nas diferentes histórias que podem ser consideradas. Em seguida é vista a influência da utilização do paradigma de orientação a objetos na recuperação de informações.

O capítulo 8 traz a definição da linguagem de recuperação de informações TF-ORM. São apresentados a forma geral da sintaxe das possíveis consultas e alguns exemplos de sua utilização. A sintaxe completa é apresentada, sob forma de uma BNF, no Anexo 2.

No capítulo 9 é apresentado um ambiente implementado para apoiar especificações feitas através da utilização do modelo de dados TF-ORM, ambiente este que faz uso de uma biblioteca de classes para propiciar a reutilizações de classes anteriormente definidas. São descritas as ferramentas implementadas e feitas algumas considerações a respeito dos modelos de dados internos utilizados.

Um estudo de caso para exemplificar a utilização do modelo de dados TF-ORM em especificações de sistemas de informação de escritórios é apresentado no capítulo 10. São descritos o problema e as soluções adotadas para a sua especificação. O desenvolvimento completo da especificação é apresentado no Anexo 3.

No capítulo 11 são apresentadas as conclusões deste trabalho, sendo acrescentadas algumas idéias quanto à sua continuação.

2. ESPECIFICAÇÃO DE SISTEMAS DE INFORMAÇÃO DE ESCRITÓRIOS

Os Sistemas de Informação de Escritórios (SIE) são casos particulares de sistemas de informação. São sistemas sócio-técnicos, muito complexos, apresentando grande influência humana. A especificação das parcelas estruturadas destes sistemas bem como do relacionamento entre as parcelas estrtuturadas e as pouco estruturadas é fundamental para que se obtenha uma implementação eficiente.

Este capítulo apresenta, inicialmente, as principais características dos SIEs, salientando os aspectos mais importantes de seu desenvolvimento. Em seguida são analisados alguns métodos de especificação existentes, específicos para estes sistemas.

2.1. Características de Sistemas de Informação de Escritórios

Um escritório é uma unidade funcional de processamento de informações de uma organização [NEW 80]. Consiste de um número elevado de elementos (pessoas, documentos) interagindo entre si de modos geralmente bem definidos. A comunicação entre eles pode ser efetuada através da passagem de mensagens, verbalmente ou através de documentos. O conhecimento é compartilhado por todos os elementos do escritório. As entidades do escritório executam suas atividades de modo concorrente e assíncrono. São executadas diversas atividades diferentes que interagem entre si. A existência de pessoas como elementos integrantes e fundamentais de um escritório faz com que as atividades desempenhadas incluam decisões humanas (parcela pouco estruturada).

O processo de automação de um escritório consiste em fornecer ferramentas computacionais às pessoas que nele trabalham, de modo a auxiliá-las no trabalho rotineiro e nas atividades decisórias. O conjunto destas ferramentas forma um Sistema de Informação de Escritório (SIE).

SIEs são tipos particulares de Sistemas de Informação. Um SIE ideal deve ser construído com base nos seguintes componentes [ADI 86]: (i) um banco de dados para armazenar as informações contidas nos documentos, (ii) um meio de comunicação, tal como uma rede local, (iii) um conjunto de atividades que caracterizam as funções dos empregados do escritório, e (iv) um conjunto de usuários que utilizam o sistema.

O tempo tem grande importância no tratamento das informações. É necessário, por exemplo, para representar a duração de eventos, o calendário, as previsões e o tempo de vida de documentos ou de operações. Restrições temporais a determinadas atividades são fatores fundamentais para representar a interação entre as diferentes atividades que podem ser executadas neste tipo de aplicação. A necessidade de inclusão do tempo associado às informações armazenadas no banco de dados foi reforçada com o recente crescimento do uso de sistemas de apoio à decisão para a área gerencial dos SIEs. É fundamental a capacidade de manipulação do tempo e de informações históricas para o planejamento empresarial, a investigação de relações causais e análise retrospectiva [ARI 86].

2.1.1. Classificação das Atividades de SIEs

Um dos fatores que diferencia os SIEs de outros sistemas de informação é o fato das ações executadas sobre as informações serem as mais variadas possíveis, sofrendo muito a influência humana. As atividades executadas nos SIE podem ser classificadas quanto ao grau de influência humana que apresentam, que corresponde diretamente à possibilidade de serem automatizadas, isto é, executadas por computadores. As atividades que são totalmente automatizáveis são chamadas de **atividades estruturadas**. Aquelas que possuem partes de trabalho que necessitam necessáriamente ser desenvolvidas por pessoas, por envolverem ações não automatizáveis, como por exemplo, tomadas de decisão, são chamadas de **atividades pouco estruturadas**. Os SIE apresentam, ainda, atividades que não podem ser executadas por computadores, devendo ser totalmente desenvolvidas por pessoas. São chamadas de **atividades não estruturadas**, constituindo o assim chamado "trabalho criativo" do ser humano. As atividades pouco estruturadas ou não estruturadas podem ser apoiadas por computadores, através de indicações de passos a serem seguidos, da detecção e correção de erros, etc.

As aplicações de SIEs são compostas por diversas atividades interagindo entre si. Os três tipos de atividades acima citados podem aparecer em uma mesma aplicação. O grau de apoio informatizado que pode ser providenciado para uma determinada aplicação é função direta do grau de estruturação das diversas atividades envolvidas na mesma. As atividades pouco estruturadas podem ser desdobradas em partes automatizadas e outras que dependem de decisões humanas. A figura 2.1 apresenta um esquema dos tipos de atividades que podem estar presentes em uma aplicação e das possibilidades de apoio informatizado.

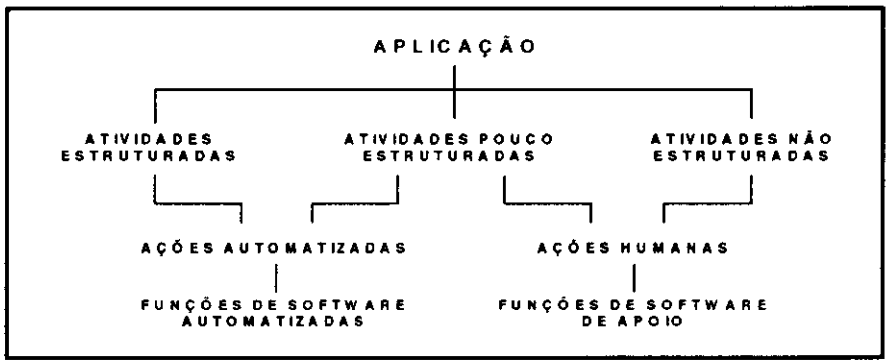

Fig. 2.1 : Tipos de Atividades e Apoio Informatizado

2.1.2. Estrutura dos Dados Manipulados

As estruturas de dados que guardam as informações dos SIEs são usualmente derivadas, se não totalmente então pelo menos em parte, dos requisitos iniciais de processamento do sistema. Os objetos que guardam informações e que circulam em um escritório podem ser os mais diversos possíveis. Os documentos em forma de papéis (formulários, fichas, etc.) são somente um subconjunto limitado destes objetos.

A representação destes documentos em um banco de dados é simples. Os sistemas de informação tradicionais somente tratavam de informações formatadas (que podem representar documentos comuns). Os avanços da tecnologia permitiram que se abrisse a possibilidade de armazenar novas formas de informação, tais como voz, imagem e textos longos. As modernas estações de trabalho têm (ou terão brevemente) todas as capacidades computacionais necessárias para a representação de dados não formatados. Elas oferecem equipamentos de alta resolução apropriados para as aplicações que requerem o processamento de imagens, além de discos óticos com enorme capacidade de armazenamento, capazes de armazenar textos, vozes e imagens digitalizadas. Os avanços tecnológicos tornaram possível, portanto, o desenvolvimento de sistemas de informação baseados em bancos de dados muito grandes, distribuídos, contendo informações não-formatadas, inclusive objetos complexos multimidia.

Os modelos utilizados para representar as informações nos SIEs devem, portanto, ser semanticamente ricos para permitir a representação de tais formas de informações. Para armazenar as informações do sistema devem ser utilizados bancos de dados que apresentem, além dos dados não-formatados, a semântica do ambiente da aplicação no seu esquema conceitual.

Outro aspecto importante que deve estar presente no modelo de dados é a representação do tempo associado às informações. O banco de dados para armazenar as informações do SIE deverá, portanto, ser um banco de dados temporal [JEN 93a], armazenando não somente os dados atuais mas todos os valores anteriormente definidos e permitindo, desta maneira, que estados passados (de momentos anteriores ao momento atual) do banco de dados possam ser reconstituídos. Informações quanto ao futuro também podem ser armazenadas, para permitir previsões ou para controlar eventos futuros.

Os bancos de dados dos SIEs, além de suportar um grande número de informações, apresentam linguagens para especificar as transações para serem utilizadas na manipulação destes dados. Estas transações, em um SIE, representam as possíveis atividades que podem ser executadas sobre as informações. O que caracteriza a diferença destes sistemas de informação dos demais é que estas transações podem não estar completamente definidas (atividades pouco estruturadas). O método utilizado para especificar o SIE deverá, portanto, apresentar ferramentas para contornar este problema.

2.1.3. Pessoas Envolvidas nos SIEs

Dois grupos distintos de usuários atuam em um SIE além dos projetistas do sistema: (1) os projetistas de aplicações, e (2) os usuários finais do sistema. Os projetistas de aplicações são especialistas que possuem conhecimento necessário para entender a especificação dos dados e dos requisitos do sistema de armazenamento, podendo assim projetar aplicações sobre estes dados. Os usuários finais, entretanto, não conhecem o sistema de armazenamento dos dados, mas possuem conhecimentos importantes sobre o ambiente em que se desenvolverá a aplicação. O auxílio deste dois tipos de usuários é muito importante no desenvolvimento de um SIE, apoiando o trabalho do projetista.

2.2. Ciclo de Vida do Desenvolvimento de um SIE

Diversos ciclos de vida de desenvolvimento podem ser encontrados na literatura [BOE 88, GIR 90, HEN 90]. No caso de desenvolvimento de sistemas de informação, as seguintes fases principais estão geralmente presentes: (i) coleta de requisitos junto aos usuários do sistema; (ii) organização e análise destes requisitos, resultando em uma especificação inicial do sistema, geralmente semiformal; (iii) definição do sistema, na qual é feita a opção das atividades que deverão ser automatizadas; (iv) especificação formal dos requisitos, efetuada por um projetista com base nos requisitos anteriormente definidos; esta especificação constitui o projeto lógico (esquema conceitual) do sistema a ser implementado; (v) projeto físico do sistema, para fins de sua implementação; (vi) implementação do sistema; e (vii) testes e manutenção do sistema. Estas fases não são estritamente seqüenciais - em cada uma delas devem ser levados a efeito procedimentos de validação e/ou verificação, procurando detectar o mais cedo possível eventuais erros de interpretação dos requisitos ou de projeto. Na maioria das vezes é construído um protótipo do sistema antes de sua implementação, com base na especificação formal dos requisitos, de modo a simular a execução do sistema e validá-lo junto aos usuários.

Diversos ambientes de desenvolvimento de software têm sido desenvolvidos para apoiar, através de ferramentas apropriadas, desde a coleta de requisitos e a análise do sistema, até o seu projeto. Nestes ambientes são utilizados métodos de especificação muito variados, cada um deles apresentando diferentes modelos de representação de dados e linguagens de especificação. A utilização de ferramentas para auxiliar no desenvolvimento de SIEs é muito importante devido à complexidade dos sistemas e à rapidez com que devem ser implementados. Segundo [AHL 84], a utilização destas ferramentas facilita (i) o rápido desenvolvimento de sistemas mais confiáveis, (ii) o desenvolvimento incremental de uma aplicação, (iii) a produção de sistemas extremamente flexíveis, e (iv) o desenvolvimento de aplicações comuns para usuários individuais.

Nos SIEs, a **coleta de requisitos** é efetuada em conjunto pelos empregados do escritório, seus gerentes e o analista do sistema, geralmente com o auxílio de ferramentas apropriadas. Nesta fase incial, as ferramentas somente apoiam o trabalho do analista, não podendo executá-lo mecanicamente. Isto porque, nesta fase, o envolvimento das pessoas é muito grande e muito importante. As informações recebidas são informais ou pouco-formais. As ferramentas visam a auxiliar na manipulação destas informações, organizando-as em alguma estrutura não rígida. Esta estrutura será utilizada para a análise do escritório e servirá de base ao projeto lógico (especificação formal) do mesmo.

As ferramentas utilizadas nesta fase inicial geralmente se baseiam em modelos e métodos muito difundidos [CHE 76, DEM 79]. Quase sempre apresentam interfaces gráficas para a representação dos esquemas de dados e para os fluxos de informação e de controle. Os aspectos dinâmicos das aplicações geralmente não são suportados por estas ferramentas.

A **especificação inicial dos requisitos**, resultante da coleta de requisitos, é geralmente feita de forma textual, através da utilização de linguagem natural. É, conseqüentemente, imprecisa e apresenta ambigüidades. São definidos as atividades a

serem executadas pelo sistema, os modelos de dados e as transações possíveis sobre estes dados.

A **especificação formal dos requisitos** tem por objetivo expressar estes mesmos requisitos de uma forma precisa, sem ambigüidades. Nesta fase é efetuado o projeto lógico do escritório. A utilização de um formalismo permite inclusive, em alguns casos, mecanismos de verificação através dos quais pode ser garantida a qualidade da especificação proposta. É importante que a formulação inicial dos requisitos seja compatível com o formalismo a ser empregado na especificação formal. Os métodos existentes para especificação formal são muito variados. Na seção seguinte serão analisados alguns dos métodos para especificação de SIEs.

A especificação formal serve de base não só para processos de prototipação do sistema, mas também para a sua implementação. Deve, no entanto, ser totalmente independente da implementação a ser efetuada, de modo a não sofrer a influência dos avanços tecnológicos que ocorrem.

Existem diversas maneiras de efetuar a validação de uma especificação: através da análise da própria especificação pelos usuários, da prototipação da especificação ou de métodos de controle do processo de especificação.

Para permitir a validação através da análise da especificação, esta deverá ser entendida não só pelos projetistas do sistema mas também pelos seus usuários (projetistas de aplicações e usuários finais). Estes últimos é que validam o sistema especificado, sugerindo modificações e identificando inconsistências. A validação de uma especificação formal, por parte dos projetistas do sistema e dos seus usuários, pode ser encarada como um contrato efetuado entre eles antes do início efetivo da implementação. É aconselhável a utilização de modelos de dados semânticos, pois estes facilitam a comunicação entre os diferentes grupos de pessoas que utilizam a especificação, apresentando estruturas de dados de alto nível e primitivas de manipulação de dados acessíveis a todos.

A fase de **prototipação do sistema** tem por objetivo validar a especificação formal através da simulação da execução do sistema. É o processo de validação mais utilizado em SIEs. Já existem alguns sistemas de especificação executável [GOG 82, PER 88], permitindo a prototipação da própria especificação. A validação, neste caso, não é efetuada mecanicamente, mas pelos usuários, que observam a execução do sistema prototipado, verificando se todos os requisitos solicitados estão sendo convenientemente atendidos.

É implementada uma versão rudimentar executável do sistema, considerando somente seus aspectos principais. O protótipo implementado não deve ser utilizado como uma implementação definitiva, pois não representa todo o sistema desejado. O protótipo é utilizado como ferramenta de validação não só da especificação, mas também dos requisitos especificados. Deve ser de rápida implementação, para poder ser questionado pelos usuários e, eventualmente, resultar em modificações da especificação, o que levaria a uma nova prototipação. Além do objetivo de validação, a fase de prototipação também é utilizada para definir as interfaces entre o sistema e seus usuários. Isto geralmente é feito pelo próprio usuário, através da utilização das ferramentas de prototipação.

A prototipação é de grande importância na área de SIE pois garante os testes em pequena escala antes de serem efetuadas grandes modificações em um escritório, modificações estas que vão, inevitavelmente, alterar substancialmente o trabalho lá desenvolvido. É importante, portanto, no momento da escolha de um método para especificar um SIE, considerar a possibilidade do modelo obtido ser prototipado através de ferramentas adequadas, já disponíveis para utilização.

Em paralelo à fase de prototipação pode ser executado um processo de **verificação formal** da especificação. Métodos de verificação formal se baseiam em procedimentos matemáticos, garantindo a corretude da especificação antes de sua implementação. A verificação formal ainda é pouco efetuada na prática devido à falta de ferramentas para realizá-la. Muitas pesquisas tem sido desenvolvidas nesta área, deixando entrever que no futuro isto será possível.

A última fase do desenvolvimento de um SIE é o **projeto da arquitetura física**, com a finalidade de sua implementação. Esta fase também se baseia na especificação formal. São levados em consideração os equipamentos existentes no mercado, para que possa ser efetuada a implementação do sistema.

2.3. Métodos de Especificação Formal de SIE

A obtenção de uma especificação que represente todos os aspectos de um SIE é muito difícil, devido à complexidade das atividades desempenhadas nestes ambientes. Muitas destas atividades não podem ser completamente especificadas por dificuldades de identificação e de formalização, ocorrendo grande número de exceções e de casos especiais. Além disso, em se tratando de ambientes sócio-técnicos, é grande a influência humana, que dificilmente pode ser formalizada. Existe, portanto, a necessidade de métodos de especificação especiais para utilização na área de SIE.

O modelo de escritório obtido através da especificação deverá refletir, além dos aspectos usuais de um sistema de informação, outros intrínsecos desta área. Segundo [RIC 87a], deverão representar (i) a estrutura organizacional do escritório e suas possíveis mudanças, (ii) a hierarquia de composição das unidades funcionais (atividades), (iii) a alocação ou a remoção de recursos humanos e técnicos para uma unidade funcional, ou em conseqüência de sua execução, (iv) coordenação de atividades concorrentes, (v) participação simultânea de diversas pessoas, com objetivos possivelmente diferentes, em uma mesma atividade, (vi) envolvimento simultâneo de uma mesma pessoa em mais de uma atividade, (vii) troca de informações entre pessoas e tarefas, (viii) tomadas de decisão por pessoas ou por grupos, e (ix) responsabilidades para delegação e execução de tarefas.

Os componentes essenciais de um método de especificação formal são, apud [ALV 88], (i) os modelos, definidos pelo conjunto de conceitos e de regras relativas à utilização do método, fixando o vocabulário e o tipo de abstração, (ii) as linguagens, descrições formais das imagens do sistema de informação segundo os respectivos modelos, (iii) a sequência de aplicação do método, que guia o trabalho do analista na descrição e na especificação do sistema, e (iv) as ferramentas disponíveis, que podem

auxiliar na especificação, na documentação, na avaliação, na tradução e na simulação da especificação.

2.3.1. Modelos Resultantes da Especificação Formal de SIEs

Quanto aos tipos de modelos resultantes da especificação de um SIE, estes podem ser classificados conforme visões do próprio escritório ou dos diferentes aspectos do trabalho nele desenvolvido. Estes modelos podem ser classificados [NEW 80] em modelos: (i) de bancos de dados, (ii) de fluxo de informações (baseados em processos), (iii) procedimentais (baseados em agentes), (iv) de tomada de decisão, e (v) comportamentais. Segundo [BAR 85, BRA 84], os modelos mais comuns são os baseados em bancos de dados, em processos e em agentes, além dos modelos mistos, que envolvem simultaneamente mais de uma visão do escritório.

Nos **modelos baseados em bases de dados**, a ênfase principal recai nos dados armazenados e manipulados pelo sistema, ou seja, nos aspectos estáticos do escritório. Entretanto, os aspectos dinâmicos, tais como operações do banco de dados, também são modelados através destes modelos. Os seus elementos básicos são os tipos de dados (objetos que serão manipulados pelo sistema) e as operações de manipulação destes dados. Todas as atividades do escritório são modeladas através de operações sobre os dados. Para estes modelos, muito tem sido desenvolvido na área de orientação a objetos, existindo já diversos sistemas gerenciadores que podem ser utilizados para a especificação e a prototipação de SIEs. Exemplos de modelos baseados em dados são o do sistema OFFIS [BRA 84, KON 82] e do IRIS [DER 86, FIS 87, LYN 86a,b].

Os **modelos baseados em processos** usam as atividades executadas concorrentemente no escritório como os elementos mais relevantes, ou seja, se concentram nos seus aspectos dinâmicos. A finalidade destes modelos é produzir uma visão integrada de todas as atividades executadas no ambiente. O objetivo principal destes modelos é descrever o fluxo de controle dentro do ambiente, e as regras que devem ser satisfeitas pelas atividades. Exemplos de método que segue este modelo são os métodos GOFOR [RIC 87a] e o INSYDE [KIN 85].

Nos **modelos baseados em agentes** o escritório é modelado sob ponto de vista das funções executadas pelos elementos ativos do ambiente (os agentes). O modelo descreve o escritório associando um conjunto de funções aos diferentes agentes. Estas funções representam os papéis que eles assumem ao executar suas tarefas, os domínios autorizados para suas atuações e seus relacionamentos com outros agentes. Deste modo, além dos dados e dos processos presentes nos outros modelos, neste também existe um terceiro conjunto de elementos a considerar na modelagem, que é aquele formados pelas pessoas que trabalham no escritório. Exemplo de modelo baseado em agentes, apud [BRA 84], é o STRUCTURAL OFFICE MODEL.

Os **modelos de tomada de decisão** dão ênfase às atividades de tomada de decisão de gerentes e de outros empregados do escritório. Segundo [NEW 80], existe um grande fator de incerteza envolvido neste modelos.

Finalmente, os **modelos comportamentais** se fixam nos aspectos sociais do escritório, envolvendo situações e encontros que fazem parte das tarefas de processamento das informações do escritório. Estes modelos se baseiam em aspectos sociológicos e comportamentais. Estes modelos, devido à natureza dos aspectos que apresentam, são mais facilmente representáveis através de métodos informais de especificação.

Alguns modelos são mistos, como por exemplo aqueles que utilizam dados e processos simultaneamente na modelagem, como OSIRIS [BAR 85] e C-TODOS [PER 88].

2.3.2. Modelos de Dados Orientados a Objetos como Método de Especificação de SIEs

Dentre os diferentes modelos a serem utilizados para especificar formalmente um escritório, uma das formas mais utilizada é através da definição de um **esquema** a ser implementado em um banco de dados que represente o sistema. Neste enfoque, a especificação formal de um SIE consiste de um esquema conceitual, composto de um modelo conceitual e de uma linguagem formal. Através do modelo conceitual devem estar representadas todas as características do escritório, tanto as estáticas como as dinâmicas. As características estáticas descrevem os aspectos estruturais do escritório, tais como documentos, objetos, mensagens, agentes, etc. As dinâmicas descrevem a evolução das entidades estáticas no esquema do escritório, ou seja, descrevem o fluxo válido de informações. Esta descrição é feita através de um conjunto de conceitos e de regras formais, regras estas que incluem restrições temporais. O modelo deve, portanto, representar a estrutura das informações manipuladas, seus relacionamentos e seu comportamento.

Uma das vantagens da especificação formal através do modelo de dados reside no fato da utilização de um só formalismo, não sendo necessário que o usuário aprenda uma linguagem de especificação adicional. As linguagens de especificação geralmente são de difícil compreensão e utilização devido à sua complexidade, sendo o emprego de um só formalismo em todo o sistema um fator importante para a utilização adequada do sistema. O usuário aprende as capacidades do sistema através do mesmo formalismo que irá utilizar no desenvolvimento de suas aplicações. Também os implementadores do sistema, os redatores de manuais, os engenheiros do banco de dados, que precisam conhecer o modelo de dados, terão a sua tarefa simplificada, uma vez que a especificação é feita através deste próprio modelo.

A descrição formal do esquema conceitual é efetuada através de uma linguagem formal de especificação. Este tipo de linguagem possui construtores especiais, capazes de descrever todos os aspectos do modelo conceitual - a estrutura das informações, os processos e sua sincronização. Alguns métodos utilizam representações gráficas que facilitam a especificação das propriedades dinâmicas e, sob certos aspectos, o entendimento do sistema especificado. Estas representações se baseiam em métodos formais, tais como as Redes de Petri [HEU 88, REI 85] e o método REMORA [ROL 82].

Os **modelos de dados orientados a objetos** têm sido muito utilizados para representar SIEs. Em [VLA 93] é demonstrado que são mais apropriados a estes tipos de aplicações do que outros tipos de métodos, tais como especificação de funções, tipos abstratos de dados, lógica de predicados e especificação de processos.

Três dos princípios fundamentais dos modelos de dados orientados a objetos proporcionam mecanismos apropriados à análise e à especificação de sistemas complexos [BRU 93]. São eles: (i) o princípio da localidade, o qual permite a concentração em um só objeto; (ii) o princípio do refinamento, que permite o refinamento de objetos através de elos de herança; e (iii) o princípio da globalidade, o qual permite localizar diferentes tipos de dependência entre objetos.

Bancos de dados orientados a objetos apresentam a semântica da aplicação em seu esquema conceitual. Não baseiam a estruturação dos dados em conceitos de registros lógicos, como é feito nos modelos relacional, hierárquico e de redes, mas apresentam o ambiente da aplicação através de objetos (entidades). Estes objetos são relacionados através de atributos e classificados através de classes e de subclasses. Os modelos de dados orientados a objetos introduzem um conjunto muito rico de primitivas de estruturação que suportam abstrações tais como classificação, generalização - especialização e agregação.

Os objetos, que representam elementos e conceitos de um ambiente de aplicação, são entidades únicas no banco de dados, com sua identidade e existência própria. Podem ser referenciados independentemente dos valores de seus atributos. Esta é a maior vantagem que apresentam sobre os modelos orientados a registros, nos quais os objetos, representados pelos registros, somente podem ser referenciados em termos dos valores de seus atributos.

Um banco de dados orientado a objetos se caracteriza por: (i) ser formado por um conjunto de tipos abstratos de dados, com operações definidas sobre estes tipos, as quais caracterizam as propriedades ou o comportamento dos tipos, sendo os objetos instâncias destes tipos; e (ii) serem os objetos acessados e manipulados somente através das operações definidas para os tipos, ficando a estruturação dos dados e os detalhes da implementação, totalmente escondidos. Os objetos são definidos, portanto, somente através de suas propriedades (seus comportamentos).

Um sistema que se baseie em bancos de dados orientados a objetos deverá obedecer às principais características do paradigma de orientação a objetos que, segundo [BAN 89] são: (i) objetos complexos, construídos a partir de objetos simples através de construtores de objetos (listas, conjuntos, tuplas); (ii) identidade dos objetos, que terão existência independente de seus valores; (iii) encapsulamento, sendo a única parte visível do objeto a sua interface, ficando sua estrutura e sua implementação escondidos da utilização; (iv) tipos ou classes, que reúnem conjuntos de objetos com características iguais; (v) herança de propriedades definida pela hierarquia de tipos e classes (caracterizando hierarquias de especialização/generalização); (vi) "*overriding*" ("*overloading*"), característica que permite que métodos (propriedades) sejam redefinidas para tipos mais específicos, levando à possibilidade de um mesmo nome de método ter

significados diferentes conforme o objeto sobre o qual é aplicado; e "*late binding*", feito em tempo de execução, para permitir a execução da propriedade anterior; (vii) extensibilidade, sendo possível definir novos tipos e classes que serão tratados pelo sistema exatamente da mesma maneira daqueles pré-definidos; e (viii) completeza computacional, permitindo que qualquer função computável seja programável através da utilização do sistema, o que possibilita a definição destes novos tipo e classes.

2.3.3. Alguns Métodos de Especificação de SIE

A maior parte dos métodos encontrados na literatura para especificar SIEs utiliza o modelo de banco de dados, sendo a especificação composta pela definição do modelo de dados a ser implementado. Notou-se que a grande preocupação dos sistemas implementados nos últimos anos está sendo a de criar novos modelos de dados (além de estender os modelos de dados já implementados) para poder atender a aplicações mais complexas, entre as quais se situam os SIEs. A maior parte dos métodos encontrados utiliza o paradigma de orientação a objetos no modelo de dados.

Os sistemas de gerência de bancos de dados orientados a objetos IRIS [DER 86, FIS 87, LYN 86a,b], ORION [BAN 87, GAR 88, KIM 87], ENCORE [HOR 87] e ODM/DODM [LYN 84], PROBE que implementa o modelo de dados PDM [MAN 86] e POSTGRES [STO 86] são todos próprios para a área de SIEs. Apresentam ambientes específicos para apoio à definição do esquema que implementam. A ênfase na possibilidade de definição de novos tipos de dados gerou o sistema EXODUS [CAR 86, CAR 88a, RIC 87b], que serve para a implementação de um banco de dados. Através dele pode ser criado um novo modelo de dados. Serve, também, para estender os tipos de dados já existentes.

A maior parte dos métodos existentes somente analisa as tarefas estruturadas dos SIEs. São destinados a representar o trabalho rotineiro e repetitivo do escritório, com entradas e saídas bem definidas, que pode ser realizado por procedimentos automatizados. Exemplos de sistemas que também permitem a representação de algum trabalho pouco estruturado são o sistema baseado em KNOs (*KNowledge acquisition, dissemination, and manipulation Objects*) [CAS 88, TSI 87], um modelo orientado a objetos para suporte de atividades pouco estruturadas em ambientes concorrentes [WOO 86] e o sistema AMS [TUE 88].

3. ASPECTOS TEMPORAIS EM ESPECIFICAÇÕES

Ao construir a especificação de um sistema de informação, não só a estrutura dos dados manipulados deve ser definida, mas também sua dinâmica - seu comportamento com a passagem do tempo. Na coleta dos requisitos do sistema, a qual fornece os elementos necessários à posterior especificação, devem ser identificados os requisitos temporais da aplicação em questão. O método utilizado na especificação do sistema de informação correspondente à aplicação deve permitir que todos os aspectos temporais sejam representados.

Nas seções seguintes inicialmente são feitas algumas considerações a respeito de aspectos temporais em sistemas de informação. Em seguida é apresentada uma taxonomia, desenvolvida neste trabalho, para os diferentes requisitos temporais identificados em SIEs. Finalmente, são apresentadas as diferentes formas utilizadas para representação de aspectos temporais.

3.1. Importância da Especificação de Aspectos Temporais

Informações temporais, tais como valores temporais, restrições temporais e caracterísitcas de evolução temporal, estão presentes em grande número de aplicações do mundo real, em especial da área de SIEs aqui tratada. A representação de aspectos temporais é fundamental na representação de uma aplicação através de um modelo de dados. A possibilidade de representação de informações temporais e de restrições de integridade dinâmica e a possibilidade de manipulação de dados históricos deve ser considerada no modelo de dados utilizado.

A representação dos aspectos temporais na especificação de um sistema de informação é importante por mais de um motivo: (i) o sistema pode apresentar informações temporais a serem introduzidas no banco de dados que o representa, sob forma de informação propriamente dita; (ii) processos a serem executados podem apresentar interações temporais, interações estas que devem também ser representadas; (iii) determinadas tarefas podem apresentar pré-condições à sua execução, as quais podem ser representadas através de restrições temporais; e (iv) condições de integridade temporal do banco de dados podem ser necessárias.

Diversos modelos de dados foram estendidos para possibilitar a representação de aspectos temporais. Como importantes extensões do modelo relacional [COD 70] podem ser citados os modelos HRDM (*Historical Relational Data Model*) de Clifford [CLI 87, CLI 93], IXRM (*Interval-extended Relational Model*) de Lorentzos [LOR 93], TRM (*Temporal Relational Model*) de Navathe [NAV 93] e o modelo de Tansel [TAN 88, TAN 93]. Elmazri e Wuu [ELM 90, 93] apresentam o modelo TEER (*Temporal EER*), uma extensão temporal para o modelo ER.

O tratamento de tempo em modelos de dados orientados a objetos está presente em alguns dos modelos recentemente apresentados [ARA 90, CHE 93, CLI 88a, GRE 86, LOU 91a, MYL 90, OLI 93, SER 91, SU 93, WUD 92]. Entretanto, a representação de aspectos temporais em bancos de dados orientados a objetos tem sido feita nestes modelos

de uma forma bastante limitada. Em sua grande maioria os aspectos temporais são tratados da mesma forma como o foram nos modelos relacionais, não sendo levada em consideração a natureza da orientação a objetos e dos problemas que podem advir da utilização deste paradigma, tais como [PIS 93]: (i) quais são os aspectos temporais relevantes aos objetos e quais as propriedades que são essenciais à existência de um objeto independentemente do tempo; (ii) como deve ser descrita a estrutura de um objeto ao evoluir com o tempo, quando pode apresentar diferentes representações; (iii) quais os relacionamentos temporais entre objetos; (iv) como são modeladas as ações internas de um objeto e suas facetas comportamentais em conseqüência a transições; (v) quais as operações necessárias ao tratamento do tempo no contexto de orientação a objetos; e (vi) quais as restrições temporais necessárias quando considerados objetos.

3.2. Classificação de Requisitos Temporais

A representação de aspectos temporais é necessária em sistemas de informação para representar não somente as informações temporais a serem introduzidas no banco de dados representativo do sistema, mas também para modelar as interações entre os processos que podem ser executados nestes ambientes. Analisando o domínio considerado (Sistemas de Informação de Escritórios), dois tipos diferentes de requisitos temporais são identificados: requisitos representados por informações temporais incondicionais a respeito de eventos e requisitos condicionais.

3.2.1. Requisitos Temporais Incondicionais

Requisitos temporais incondicionais são aqueles que são explicitamente definidos. Representam um instante ou intervalo de tempo específico associado a uma informação. Podem se apresentar sob duas formas: bem-definidos ou parcialmente definidos.

3.2.1.1. Requisitos Temporais Incondicionais Bem-definidos

Requisitos temporais incondicionais bem-definidos são aqueles representados explicitamente por uma informação temporal, a qual serve para definir a informação de forma unívoca. Três tipos diferentes destes requisitos podem ser identificados:

- registro de um elemento temporal associado a um evento, como por exemplo a data de aniversário de uma pessoa ou o horário de um encontro comercial;
- o tempo de duração de um evento, como por exemplo, a duração de uma reunião;
- o período de tempo durante o qual uma informação é válida, como por exemplo, uma taxa de câmbio monetária, a qual é válida durante um certo período de tempo.

3.2.1.2. Requisitos Temporais Incondicionais Parcialmente Definidos

Em diversas aplicações é necessária a representação de informações temporais incompletas, caracterizando requisitos parcialmente definidos (não definidos completamente). Este tipo de requisito é geralmente utilizado para representar eventos

que podem ocorrer em mais de uma data. Exemplos deste tipo de requisito são os seguintes:

- a ocorrência de um evento **antes** ou **depois** de uma determinada data, como por exemplo, a entrega da declaração de Imposto sobre a Renda, a qual deve ser entregue qualquer dia até uma data determinada;

- a ocorrência de um evento **durante** um intervalo de tempo, como por exemplo a matrícula de um aluno na UFRGS, a qual deve ser efetuada entre duas datas determinadas.

3.2.2. Requisitos Temporais Condicionais

Requisitos temporais condicionais representam informações temporais que não são explicitamente definidas, sendo expressas através de condições. Dois tipos diferentes de requisitos temporais condicionais podem ser identificados:

- requisitos temporais condicionais que representam restrições casuais na possível execução de processos; estes requisitos controlam o início da execução de processos, coordenando a execução de processos concorrentes. Através deles pode ser representado que o início da execução de um processo seja determinado pela ocorrência de um determinado evento. Pode-se também condicionar o início da execução de um processo a qualquer instante antes (ou depois) da ocorrência de algum evento.

- requisitos temporais condicionais podem ser representados através de restrições temporais a valores; através destas restrições podem ser relacionadas informações de diferentes tempos de definição, implicita ou explicitamente. Como exemplo deste tipo de requisito podemos considerar a restrição que controla o fato de que um novo valor de salário nunca deve ser inferior a um valor anteriormente definido para esta propriedade.

3.3. Formas de Representação Temporal

Pesquisas recentes [ARA 91, GAB 91, MAI 91a, WIE 91] apresentam diferentes formas para representar o tempo. Ao escolher uma forma de representação temporal, alguns itens relevantes devem ser definidos. A forma escolhida se reflete em interpretações diferentes dos conceitos temporais envolvidos. A seguir são analisados alguns dos aspectos relevantes a serem definidos quando da escolha de uma forma de representação temporal e a conseqüente interpretação de entidades temporais envolvidas na representação.

3.3.1. Ordem no Tempo

A definição de uma ordem a ser seguida no tempo é fundamental quando utilizada alguma representação temporal. O mais comum é que se assuma que o tempo flui **linearmente**; isto implica em uma ordenação total entre quaisquer dois pontos no tempo.

Definidos dois pontos diferentes no tempo t e t', representando a ordem de precedência temporal através do operador "<", uma e somente uma das seguintes expressões é verdadeira: $t < t'$ ou $t' < t$.

Em alguns casos pode ser considerado **tempo ramificado** ("*branching time*"). Para estes a restrição linear é abandonada permitindo a possibilidade de dois pontos diferentes serem sucessores (ramificação no futuro) ou antecessores (ramificação no passado) imediatos de um mesmo ponto. Uma ramificação no futuro implica que podem ser considerados múltiplos possíveis desenvolvimentos futuros do domínio, enquanto que uma ramificação no passado admite múltiplas histórias passadas do domínio em questão. A combinação "passado linear, futuro ramificado" trabalha com uma só história passada e admite múltiplas histórias futuras, representando desta maneira a realidade atual de uma forma bastante fiel.

Uma última opção de ordenação temporal é considerar o **tempo circular**. Esta forma pode ser utilizada para modelar eventos e processos recorrentes [MAI 91b].

3.3.2. Variação Temporal

Outro item importante a ser definido é qual a forma de variação temporal considerada [SEG 88b]. Duas formas basicamente diferentes podem ser consideradas: tempo contínuo e tempo discreto. O tempo é **contínuo** por natureza. Entretanto, sem grande perda de generalidade, o tempo pode ser considerado como **discreto**. Esta segunda forma de representação simplifica grandemente a implementação de modelos de dados.

Modelos de dados que suportam uma noção discreta de variação temporal são baseados em uma linha de tempo composta de uma seqüência de intervalos temporais consecutivos, que não podem ser decompostos, de idêntica duração. Estes intervalos são denominados *chronons* [JEN 93a]. A duração particular de um *chronon* não é necessariamente fixada no modelo de dados, ficando para ser definida em implementações particulares do modelo de dados.

Outra forma de variação discreta é a **variação escada** (por eventos), segundo a qual um valor fica constante desde sua definição até o instante em que outro valor for definido. Este é o tipo de variação mais comum nas implementações de bancos de dados temporais.

A variação temporal de um modelo de dados pode, ainda, ser **definida por uma função** que permite a interpolação de valores para obter os valores não definidos. Esta função de interpolação pode ser definida pelo usuário ou incluída na modelagem conceitual.

3.3.3. Granularidade Temporal

A granularidade temporal de um sistema consiste na duração de um *chronon*. Entretanto, dependendo da aplicação considerada, às vezes é necessário considerar simultaneamente diferentes granularidades (minutos, dias, anos) para permitir uma melhor representação da realidade. Embora o *chronon* do sistema seja único, é possível simular

estas diferentes granularidades através de funções e operações disponíveis nos sistemas gerenciadores do banco de dados que implementam o modelo.

3.3.4. Instante no Tempo

O conceito de instante, representando um particular ponto no tempo, depende da forma de variação temporal considerada. Quando é considerado tempo contínuo, um instante é um ponto no tempo de duração infinitesimal. Neste caso os instantes são isomórficos com os números reais. Quando, no entanto, é considerada a variação temporal discreta, um instante é representado por um dos *chronons* da linha de tempo suportada pelo modelo. Na variação discreta os instantes são isomórficos aos números inteiros ou a um subconjunto destes. Diz-se que um evento ocorre no tempo t se ocorre em qualquer tempo durante o *chronon* representado por t. Um *chronon*, que é a menor duração de tempo suportada por um SGBD temporal, pertence à representação discreta de tempo.

Considerando a ordem de variação temporal linear, temos a existência de um instante especial, correspondente ao **instante atual** (***now***), o qual se move constantemente ao longo do eixo do tempo. Este ponto define o que é considerado como passado (qualquer ponto anterior a este) e como futuro (qualquer ponto posterior a ele).

3.3.5. Intervalo Temporal

Um intervalo temporal é caracterizado pelo tempo decorrido entre dois instantes. Depende também da forma de representação temporal definida no modelo. Quando é considerado tempo contínuo, o intervalo consiste de infinitos instantes de tempo. Na variação discreta um intervalo é representado por um conjunto finito de *chronons* consecutivos. Um intervalo é representado pelos dois instantes que o delimitam. Dependendo da pertinência ou não dos instantes limites ao intervalo este pode ser aberto (os limites não pertencem ao intervalo), semiaberto (um dos limites pertence ao intervalo) ou fechado (ambos os limites pertencem ao intervalo). Quando um dos limites é representado pelo instante atual (*now*) temos a representação particular de um intervalo cujo tamanho varia com a passagem do tempo.

Quando considerados intervalos, a variação temporal é linear. Se um intervalo fechado for representado por $[I1, I2]$, uma das duas fórmulas deve ser verdadeira:

$$I1 < I2 \text{ ou } I1 = I2.$$

3.3.6. Duração Temporal

A representação de um duração temporal pode também ser considerada como primitiva. Durações temporais podem ser basicamente de dois tipos, dependendo do contexto em que são definidas: fixas e variáveis. Uma *duração fixa* independe do contexto de sua definição. Um exemplo típico de uma duração fixa é uma hora que tem sempre, independentemente do contexto de sua utilização, a duração de 60 minutos. Já a *duração variável* depende do contexto, sendo um exemplo típico a duração de um mês, que pode ser de 28, 29, 30 ou 31 dias.

3.3.7. Limites do Tempo

Conforme a representação temporal utilizada, o conceito de limites no tempo pode variar [MAI 91b] . Quando considerados somente pontos no tempo, os limites do tempo se referem a considerar ou não o tempo como infinito. O conceito de tempo infinito consiste em considerar que todo ponto no tempo apresente sempre um sucessor e um antecessor. Em modelos orientados a objetos este conceito fica limitado, por exemplo, ao tempo de vida de um objeto. No caso das teorias baseadas em intervalos, os limites do tempo se referem geralmente à pertinência ou não dos pontos limites ao intervalo, definindo se os intervalos são abertos ou fechados em um ou em ambas as extremidades.

3.3.8. Representação Temporal Explícita e Implícita

A definição de tempo pode ser feita de forma explícita, através por exemplo da associação de um instante de tempo a uma informação (*timestamping*), ou de forma implícita através da utilização de uma linguagem de lógica temporal.

Para a representação explícita de tempo é necessária a definição de um *elemento temporal primitivo*, como instantes ou intervalos. Um grande número de sistemas temporais utiliza como elemento temporal primitivo o instante, como por exemplo os modelos TODM (*Temporally Oriented Data Model*) de Ariav [ARI 86] e OODAPLEX de Wuu e Dayal [WUU 93].

Sistemas nos quais os mecanismos de raciocínio a respeito de tempo são de fundamental importância, tais como sistemas de *scheduling*, a noção de intervalo temporal é tomada como primitiva; nestes casos os eventos são representados como intervalos muito pequenos. Um importante enfoque para representação temporal é a álgebra intervalar de Allen [ALL 83]. Os eventos são representados por intervalos de tempo. Estes intervalos são relacionados entre si através de relacionamentos temporais, descritos por predicados de uma linguagem de lógica temporal. Os modelos TRM de Navathe [NAV 93], TEER de Elmasri e Wuu [ELM 90, 93], TELOS de Mylopoulos [MYL 90] e RML de Greenspan [GRE 86] se baseiam nesta teoria.

Uma outra forma de representação explícita de tempo é através da associação de conjuntos de instantes temporais a cada informação - conjuntos de pontos no tempo representando os diferentes tempos de validade. Esta forma de representação temporal foi introduzida por Gadia [GAD 88], sendo posteriormente utilizada em diversos outros modelos, como por exemplo no modelo relacional de Tansel [TAN 93].

A utilização de lógica temporal para especificação de propriedades temporais é encontrada em diversos sistemas e linguagens [BOL 83, COR 91, GRE 86, LOU 91b, MAI 91a, SCH 83]. No Cálculo de Eventos [KOW 86], raciocínio a respeito de eventos e tempo é realizado em um ambiente de programação em lógica. A mais importante contribuição deste enfoque é a possibilidade de tratar de informações incertas e imprecisas, tais como *antes* e *depois*.

3.3.9. Tempo de Transação e Tempo Válido

Snodgrass and Ahn propuseram uma taxonomia para o tempo em bancos de dados [SNO 85, 86] que consiste basicamente de três conceitos temporais distintos: (i) *tempo de transação*, o tempo no qual é feita a atualização do banco de dados; (ii) *tempo válido*, representando o período de validade da informação armazenada; e (iii) *tempo definido pelo usuário*, consistindo de propriedades temporais definidas explicitamente pelos usuários em um domínio temporal e manipuladas pelos programas de aplicação. É hoje consenso na comunidade de modelagem temporal de que tanto a representação do tempo de validade de uma informação como o do seu tempo de transação devem ser consideradas. Estes dois tempos são ortogonais, podendo ser tratados separadamente ou em conjunto.

O tempo de validade pode ser representado de formas distintas, dependendo do elemento temporal básico utilizado no modelo. Quando for utilizado o elemento temporal ponto no tempo, o tempo de validade pode ser representado: (i) através de um ponto no tempo indicando o início da validade, permanecendo o valor válido até que inicie o tempo de validade de outro valor; ou (ii) através de dois pontos no tempo, o primeiro indicando o início da validade e segundo, o final da validade. Nos modelos que utilizam o intervalo temporal como elemento temporal básico, o tempo de validade é definido através do intervalo de validade do valor.

3.3.10. Bancos de Dados Temporais

A possibilidade de manipular informações históricas - estados passados do banco de dados que representa a aplicação - é também um fator importante no modelo de dados utilizado, por permitir a representação de restrições temporais e de condições de integridade temporal. Dependendo da possiblidade de representação de informações históricas, os bancos de dados foram classificados por Snodgrass e Ahn [SNO 85, 86] em:

- *bancos de dados instantâneos* (*"snapshot databases"*): são os bancos de dados tradicionais, nos quais somente a última definição dos valores é armazenada. Sempre que um novo valor para uma determinada propriedade é definido, o valor anterior desta propriedade é perdido. Nestes bancos de dados os aspectos temporais somente podem ser tratados através de tempos definidos pelos usuários e através de aplicações externas;

- *bancos de dados de tempo de transação* (*"rollback databases"*): nestes bancos de dados todos os valores definidos ficam armazenados permanentemente, valendo a partir do momento de sua definição. Cada valor armazenado tem associado o seu tempo de transação - tempo em que foi feita a definição do valor no banco de dados. Através deste tempo associado estados passados do banco de dados podem ser recuperados. Entretanto, não é possível fazer alterações de dados passados ou futuros. São bancos de dados orientados à implementação, uma vez que o tempo de transação é fornecido pelo sistema gerenciador do banco de dados;

- *bancos de dados de tempo de validade* (*"historical databases"*): modelam a realidade de uma forma mais real ao associar a cada informação o seu tempo de validade na

aplicação e não o instante de sua definição. Diversos valores para uma mesma propriedade, com tempos de validade diferentes, são mantidos pelo banco de dados. A utilização do tempo de validade associado a um valor permite que sejam feitas alterações de valores válidos no presente, no passado ou no futuro. Entretanto, como não é armazenado o tempo de transação, estados passados do banco de dados não podem ser recuperados, estando registrada somente a versão atual da validade dos dados; e

- *bancos de dados bitemporais* ("*temporal databases*"): combinam os dois tipos anteriores armazenando, para cada valor, seus tempos de transação e de validade correspondentes. Este tipo de banco de dados armazena não somente o conhecimento atual dos dados mas também todo seu histórico - todos os estados passados do banco de dados ficam armazenados. Além disso, alterações nos dados presentes, passados e futuros podem ser realizadas.

Esta taxonomia foi amplamente aceita pelos meios de pesquisa, sofrendo alterações somente os nomes fornecidos a cada uma das classes de bancos de dados (apresentados aqui entre parêntesis). Os nomes aqui adotados seguem os resultados obtidos através de uma lista de discussão para a definição de uma nomenclatura única para aspectos temporais. Nesta lista colaboraram, durante os anos de 1992 e 1993, os internacionalmente mais renomados pesquisadores em aspectos de modelagem temporal. O resultado do debate foi condensado no documento [JEN 93a]. Segundo a nomenclatura definida, *bancos de dados temporais* é um termo hoje utilizado para denominar bancos de dados que apresentem suporte a qualquer aspecto de tempo - seja tempo de transação, de validade ou ambos, com exceção do tempo definido pelo usuário.

4. REUTILIZAÇÃO DE ESPECIFICAÇÕES

As pesquisas atuais têm dado grande importância à possibilidade de reutilização de software, com o objetivo de aumentar a produtividade do desenvolvimento e a qualidade dos produtos gerados. Procura-se obter produtos mais corretos, em um menor espaço de tempo.

A reutilização pode ser efetuada em três níveis diferentes do processo de desenvolvimento de um produto de software: especificação, projeto e implementação (reutilização de código). Para que a reutilização seja realmente efetiva, a possibilidade de reutilização deve ser levada em consideração desde o início do desenvolvimento de um produto de software, já na fase de especificação dos requisitos do produto. A especificação de requisitos é uma das fases iniciais do desenvolvimento de um sistema. Através de um processo de coleta de requisitos junto aos usuários são levantados os requisitos que o sistema final deverá apresentar. Estes são analisados e organizados, resultando na especificação dos requisitos. Os requisitos são expressos através de um formalismo conceitual, devendo representar tanto as aspectos estáticos como os dinâmicos da aplicação. O projeto e posterior implementação do sistema de informação se baseiam nesta especificação. A reutilização empregada a nível de especificação deve simplificar a posterior reutilização nos níveis de projeto e de implementação do sistema. Além de aumentar a produtividade do processo de especificação, a reutilização resulta ainda em um aumento da qualidade e da confiabilidade da especificação, uma vez que são utilizadas partes de outras especificações anteriormente testadas e implementadas.

Algumas das vantagens que podem ser identificadas quando se faz uso do paradigma de reutilização a nível de especificações são as seguintes:

- diminuição do custo de desenvolvimento da especificação;
- diminuição do custo de verificação/validação da especificação;
- aumento da produtividade no desenvolvimento;
- aumento da qualidade da especificação gerada;
- padronização de especificações desenvolvidas a partir de uma biblioteca comum a várias aplicações; e
- melhor inter-operabilidade entre diferentes equipes que utilizam a mesma biblioteca.

Um dos primeiros trabalhos encontrados na literatura fazendo uso do conceito de reutilização a nível de conhecimento abstrato é o de Tueni [TUE 88], apresentando o método AMS para modelagem de sistemas de informação de escritórios. Trata-se de um método baseado em conhecimento, no qual tanto os aspectos declarativos dos conceitos do escritório e seus relacionamentos, como a organização do conhecimento do escritório em pacotes abstratos, são representados em diferentes níveis de abstração. Esta organização permite a reutilização do conhecimento abstrato, através do conceito de *Mopas*.

Quando utilizado como formalismo de especificação a linguagem de descrição de um modelo de dados, o modelo utilizado deve levar em consideração a possibilidade de reutilizar especificações posteriormente. Deverá, para tal, apresentar facilidades para armazenamento de informações em uma biblioteca de componentes de software, para sua posterior recuperação e eventual modificação.

Uma das técnicas mais utilizadas para propiciar uma eventual reutilização de especificações é a da definição de um domínio de aplicações, no qual diversas aplicações específicas diferentes poderiam ser desenvolvidas. Uma vez definidos os elementos deste domínio genérico, as especificações das aplicações particulares são realizadas reutilizando e adaptando os elementos definidos. Exemplos de sistemas que utilizam este conceito são encontrados em [GAN 91, NEI 84, SUT 91]. A definição do domínio de aplicação é um ponto bastante crítico - se o domínio é por demais específico, dificilmente os elementos nele definidos poderão ser reutilizados em aplicações diferentes; se, no entanto, este domínio é muito genérico, a necessidade de adaptação dos elementos será tão grande que não se justifica o esforço de reutilização. O domínio de sistemas de informação de escritórios por si só define um domínio para o qual é possível montar uma biblioteca de tamanho adequado à reutilização, sendo ao mesmo tempo genérico o bastante para permitir a modelagem de um grande número de aplicações.

4.1. Aspectos Gerais de Reutilização de Especificações

É importante que, quando da realização de uma especificação reutilizando componentes de outras especificações, que o domínio de aplicação dos componentes reutilizáveis seja, se não o mesmo, pelo menos compatível com o da aplicação em desenvolvimento. Três aspectos importantes devem ser considerados quando da utilização de uma biblioteca de componentes: como identificar os componentes que serão incluídos na biblioteca, como organizar a biblioteca e como recuperar os componentes armazenados.

A identificação dos componentes que deverão ser incluídos na biblioteca deve ser realizada por um especialista. É um processo iterativo, devendo ser repetido constantemente, a intervalos temporais pré-fixados. Inicialmente é feita uma *análise de domínio* (análise "a priori" ou precursora), no qual são identificados os componentes que têm a possibilidade de serem utilizados em mais de uma aplicação - a partir da análise de possíveis aplicações no domínio considerado são abstraídos componentes genéricos, definidos para posterior reutilização e armazenados em uma biblioteca de componentes. Para que possam ser reutilizados, estes componentes deverão ser descritos de forma bastante clara através de algum formalismo, com seus interfaces perfeitamente definidos. Técnicas de análise de domínio estão sendo desenvolvidas, constituindo o que ficou conhecido como *engenharia de domínio* [GIR 92]. Através da análise de aplicações efetuadas reutilizando componentes da biblioteca, especializando estes componetes e criando novos componentes, deve ser feita a manutenção da biblioteca, a intervalos de tempo regulares (análise "a posteriori"). Componentes da biblioteca podem ser então modificados conforme a necessidade mostrada nas aplicações, além de permitir a inclusão de novos componentes identificados no domínio considerado. Estudos recentes mostram o desenvolvimento de técnicas de manutenção de bibliotecas de componentes [CAS 91].

A organização da biblioteca que armazena os componentes a serem reutilizados é tão importante quanto a definição destes componentes - se o tempo gasto para localizar um componente for por demais demorado, a reutilização perde sua razão de ser, deixando de ser eficiente. Além de possibilitar a recuperação dos componentes, a forma escolhida para organizá-los deve apresentar flexibilidade suficiente para aceitar inclusões de novos componentes. Vários métodos podem ser utilizados nesta organização: identificação através de chaves, redes semânticas, etc. Na biblioteca MUSEION [GAN 91], por exemplo, é utilizado um método de classificação adaptado da classificação facetada da proposta por Prieto-Diaz [PRI 87]. O componente reutilizável é visto como um documento textual, cujos conceitos são descritos através de palavras-chave, as quais são mapeadas para as palavras do domínio de aplicação, sendo a recuperação feita através das palavras-chaves. Em [HEN 92] é descrito um sistema de classificação de objetos em uma biblioteca, com um sistema de catalogação associado.

Para reutilização de componentes de uma biblioteca estes devem ser localizados na biblioteca e especializados para se adaptarem a aplicações genéricas. A recuperação de componentes em uma biblioteca é um dos aspectos principais para a efetiva validade da reutilização [STA 84]. A reutilização não terá validade se o esforço dispendido para recuperar um componente for muito superior ao que seria gasto para a construção de um novo. Os componentes armazenados devem ser facilmente localizados, de acordo com aspectos que se quer utilizar. O ideal é que a recuperação seja auxiliada por ferramentas [PIN 90]. O armazenamento de componentes deve ser efetuado levando em consideração a sua posterior recuperação.

A reutilização de um componente sem qualquer adaptação é muito difícil de acontecer. É importante, portanto, que a biblioteca apresente, além de informações a respeito da estrutura dos componentes armazenados, informações adicionais que auxiliem a sua posterior reutilização através de indicações de como adaptá-los a aplicações específicas. As possíveis formas de reutilizar um componente podem ser classificadas em [LEN 87]:

- reutilização do componente sem nenhuma adaptação, tal como se encontra na biblioteca;
- modificação feita manualmente, após o componente ser recuperado; pode ser efetuada uma especialização do componente para adaptá-lo à aplicação específica, ou feita a alteração de alguma de suas partes;
- adaptação já prevista na definição do componente, deixando elementos a serem definidos durante a aplicação ("template" ou esqueleto);
- adaptação através de parametrização do componente, também prevista na sua definição.

Exemplos de bibliotecas de componentes são a do sistema DRACO [NEI 84], a biblioteca MUSEION [PON 91], a "Basic Eiffel Libraries" [MEY 90] e um sistema de classificação e recuperação de documentos [CEL 92].

4.2. Paradigma de Orientação a Objetos como Ferramenta para a Reutilização de Especificações

Como já citado anteriormente, a especificação de sistemas de informação pode ser feita através da definição de um modelo de dados. A utilização de um modelo de dados orientado a objetos apresenta vantagens para a reutilização de especificações.

O paradigma de orientação a objetos [KOR 90, LAD 88] baseia-se no princípio de que os problemas envolvem objetos do mundo real, interagindo entre si com comportamentos determinados e evoluindo no tempo. Os aspectos principais de uma especificação feita segundo este paradigma são a identificação e a representação destes objetos. Neste caso a especificação será composta da definição de um conjunto de classes de objetos, os quais interagem entre si através de trocas de mensagens e por mecanismos de herança.

Uma especificação feita utilizando o paradigma de orientação a objetos deve apresentar as seguintes características [SUT 91]:

- para cada classe de objetos devem ser especificados suas características estruturais e seu comportamento;
- a especificação final é constituída por uma rede de objetos que se comunicam assincronamente;
- os interfaces de cada objeto devem ser especificados através dos tipos de eventos/mensagens que podem receber e enviar, com seus efeitos de disparo nos métodos dos objetos,
- modelos orientados a objetos devem apresentar mecanismos de classificação e de herança;
- modelos de objetos devem ser abstratos, não apresentando detalhes de implementação.

Dois aspectos fundamentais do paradigma de orientação a objetos fazem com que ele seja apropriado à reutilização de especificações: o encapsulamento de informações e a possibilidade de construção de objetos complexos a partir de outros objetos.

O *encapsulamento de informações* permite que sejam construídas classes de objetos genéricas, próprias à posterior reutilização. É definido somente o comportamento externo dos objetos de uma classe, ficando os detalhes internos contidos no objeto. Dada uma biblioteca de classes de objetos de uso geral, definidas de forma mais ou menos abstrata, estas podem ser modificadas e/ou interligadas para gerar outras classes de objetos, reutilizadas em aplicações específicas. A definição precisa de seus interfaces permite sua identificação para fins de reutilização.

A possibilidade de construir *objetos complexos* a partir de outros, através da especialização e/ou composicão destes permite que objetos armazenados na biblioteca sejam utilizados como partes de novos objetos definidos na especificação. Além disso, é

possível a definição de objetos em diversos níveis de abstração através de hierarquias de classificação, de generalização e de agregação, com herança de propriedades entre os diferentes níveis. Estas estruturas dão origem a uma organização facilmente recuperável. Deste modo, classes genéricas podem ser especializadas para que se adaptem a aplicações particulares, ou combinadas para formar novas classes.

A possibilidade de reutilizar classes de objetos tem sido largamente utilizada em programação orientada a objetos - como por exemplo, através das biblioteca de classes fornecida pela linguagem SMALLTALK. No campo de especificações, entretanto, este aspecto ainda não tem sido muito desenvolvido, embora as características de orientação a objetos sejam muito apropriadas ao desenvolvimento de especificações baseadas na reutilização de classes.

A utilização do paradigma de orientação a objetos em especificações baseadas em reutilização de objetos apresenta dois passos principais [SUT 91]: (1) um processo de abstração no qual são especificadas classes de objetos genéricos, próprios à posterior reutilização; e (2) um processo de especialização destes objetos para se adaptarem a especificações específicas.

O emprego do paradigma de objetos faz com que a biblioteca de componentes seja, na realidade, uma **biblioteca de classes de objetos**. É importante lembrar que as classes integrantes de uma biblioteca são, na realidade, *templates* dos objetos que poderão ser instanciados nas aplicações.

Uma especificação feita através de um modelo orientado a objetos, reutilizando classes de uma biblioteca, vai ser composta, portanto, de:

- classes novas;
- classes reutilizadas na mesma forma como se encontram na biblioteca;
- classes reutilizadas porém modificadas;
- classes obtidas por especialização de outras da biblioteca; e
- classes compostas por outras constantes da biblioteca.

5. MODELOS PRELIMINARES ORM E F-ORM

O modelo de dados a ser apresentado é uma extensão do modelo de dados F-ORM, um modelo orientado a objetos que utiliza o conceito de papéis e foi desenvolvido no âmbito do projeto ITHACA [PRO 89], um projeto ESPRIT II da comunidade européia. Através deste conceito são representados separadamente os comportamentos diferentes de um objeto, facilitando o processo de análise da aplicação e de representação da evolução dos objetos através do tempo. Este método é uma extensão do modelo ORM, visando especificamente a representacão de sistemas de informação de escritórios. Recursos e processos são representados separadamente, sendo todos os seus inter-relacionamentos completamente definidos. A forma de representação destes dois conceitos, entretanto, é semelhante. Desta forma é evitada a necessidade de aprendizagem de dois formalismos diferentes para expressar estes dois conceitos. Um dos objetivos buscados no desenvolvimento do método F-ORM foi o de obter um modelo que propicie o desenvolvimento de especificações através de reutilização de classes pré-definidas. O particionamento do comportamento de objetos de uma classes em diferentes papéis que estes objetos podem desempenhar facilita sobremaneira o processo de reutilização destas classes.

A seguir são apresentadas as principais características dos dois modelos que dão origem ao TF-ORM.

5.1. ORM

ORM (*Object with Roles Model*) [PER 90] é um modelo de dados orientado a objetos que utiliza o conceito de papéis (*roles*) para descrever o comportamento dos objetos de uma classe.

5.1.1. O Conceito de Papéis

Nos modelos orientados a objetos tradicionais um objeto é criado como uma instância de uma classe, sendo-lhe associado um identificador de instância que o distingüe univocamente dos outros objetos. Durante todo seu ciclo de vida este objeto pertence somente a esta classe, mesmo se a partir de algum determinado momento apresente características comportamentais que possam ser identificadas com outra classe. Para exemplificar suponhamos uma classe pessoa que apresenta duas subclasses: estudante e empregado. Se um estudante passa a ser empregado, a instância correspondente deveria trocar de classe, o que geralmente não é permitido. Alguns modelos de dados permitem migração entre classes, restringindo esta geralmente à migração entre classe e subclasse, tal como no modelo PROBE [MAN 86]. Não seria possível, no entanto, que a mesma instância fosse simultaneamente empregado e estudante.

Outra característica dos modelos orientados a objetos tradicionais é não permitir a instanciação múltipla de um mesmo objeto, como no caso de uma mesma pessoa que apresente mais de um emprego - a definição de empregado como uma subclasse de pessoa permite somente uma instância desta subclasse.

A introdução do conceito de papéis tem por objetivo separar a representação dos aspectos estáticos de um objeto dos seus aspectos dinâmicos. Um objeto continua a ser uma instância de uma única classe, mas poderá assumir ao longo de sua existência um ou mais papéis, isolada ou simultaneamente, em diferentes tempos; deste modo é possível representar a evolução dinâmica do objeto no tempo. O conceito de papéis permite também que um objeto apresente, simultaneamente, diversas instâncias de um mesmo papel, possibilitando desta maneira a representação de variações de um mesmo comportamento. Considerando como exemplo uma classe pessoa, um objeto desta classe pode desempenhar os papéis de empregado, aluno e professor; pode ser simultaneamente aluno e empregado; e pode apresentar mais de uma instância do papel empregado, representando dois empregos diferentes.

O conceito de papéis é muitas vezes confundido com o de subclasse. A diferença fundamental entre uma subclasse e um papel está na identidade dos objetos. A existência de uma instância de uma subclasse está condicionada à instanciação desta subclasse. Quando utilizada a representação através de um papel, no entanto, um objeto existe como instância da classe independentemente do fato de estar ou não desempenhando o papel. Considerando o exemplo anterior, se empregado fosse representado como subclasse de pessoa, o objeto empregado somente poderia ser criado no momento em que esta pessoa iniciasse no emprego. Utilizando a representação de empregado através de um papel, o objeto pessoa existe independentemente desta pessoa possuir um emprego. Homens e mulheres são tipicamente subclasses da classe pessoa.

Um objeto assume um determinado papel através de um mecanismo de instanciação análogo àquele utilizado para as classes. Cada instância de um papel terá um identificador interno único associado. Deste modo são diferenciadas não somente as instâncias de diferentes papéis mas também as diferentes instanciações de um mesmo papel.

O conceito de papéis tem sido utilizado em outros trabalhos, com significado semelhante ao aqui considerado [ALB 93, RIC 91, SCI 89, SU 91a]. Através deste conceito são separados os diferentes comportamentos de um objeto, facilitando o processo de análise destes comportamentos. Os relacionamentos entre os diferentes comportamentos (papéis) são especificados através de regras e de restrições, as quais governam o comportamento concorrente.

A separação dos diferentes comportamentos de um objeto em papéis propicia facilidades para a reutilização de especificações. Supondo que exista uma biblioteca de classes que utilizem o conceito de papéis para representar diferentes comportamentos, é possível reutilizar uma classe identificada considerando apenas alguns dos comportamentos representados na biblioteca, dependendo das necessidades da aplicação que está sendo desenvolvida.

5.1.2. Classes e Papéis em ORM

No modelo ORM uma classe é definida por um nome único em toda a especificação de classes *cn* e por um conjunto de papéis R_i, cada papel representando um comportamento diferente deste objeto:

$$class = (cn, R_0, R_1, \ldots, R_i, \ldots, R_n)$$

Cada papel R_i consiste de um nome Rn_i, de um conjunto de propriedades P_i deste papel, de um conjunto de estados abstratos S_i que o objeto pode apresentar neste papel, de um conjunto de mensagens M_i que o objeto pode receber e enviar neste papel, e de um conjunto de regras Ru_i (regras de transição de estado e regras de integridade):

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

Os nomes dos papéis devem ser únicos dentro de uma classe. As propriedades são descrições abstratas das características de um objeto (seus atributos). São implementadas como variáveis de instâncias. Cada propriedade de P_i é definida através de um nome p_i e de um domínio d_i, o qual define o conjunto de valores que pode assumir:

$$(p_i, d_i) \in P_i$$

Os domínios podem ser simples ou complexos. Domínios simples são constituídos de classes e de domínios pré-definidos (*strings*, inteiros, booleanos, reais). Domínios complexos podem ser definidos por agregações de outros domínios; podem ser representados por conjuntos de domínios simples ou complexos (caracterizando propriedades multivaloradas) e multiconjuntos (conjuntos com repetição) de domínios simples ou complexos.

Os estados $s_i \in S_i$ descrevem estados abstratos de um objeto. Um determinado estado define o contexto de execução corrente de um objeto em um determinado papel.

As mensagens $m_i \in M_i$ descrevem quais as mensagens que podem ser enviadas e quais as que podem ser recebidas pelo objeto. Cada mensagem é representada por um nome e por uma lista opcional de argumentos, através dos quais são transmitidos os valores. Uma seta prefixando o nome da mensagem indica se esta é recebida ("←") ou enviada ("→"). As mensagens determinam as mudanças de estado do objeto. Somente serão descritas as mensagens que são relevantes às mudanças de estado do papel.

O comportamento dos objetos de uma classe é governado através de regras. Dois tipos de regras são utilizados em ORM: regras de transição de estados e regras de integridade.

As regras de transição de estados descrevem o comportamento de um objeto ao desempenhar um determinado papel. Expressam as possíveis seqüências de estados que o objeto naquele papel pode apresentar, definindo quais as mensagens que podem ser recebidas e enviadas em cada estado. Uma regra de transição de estados $s_i \in S_i$ representa uma transição válida entre dois estados de um papel, dependendo eventualmente da ocorrência de uma mensagem. A transição pode causar o envio de uma ou mais mensagens. É representada da seguinte forma:

$$\begin{aligned} &m : [state([Oid_1,] [Rid_1,] s_1),] [\leftarrow msg([Oid_2,] [Rid_2,] m_1)] \\ &\quad \Rightarrow [\{\rightarrow msg([Oid_3,] [Rid_3,] m_2)\}] [\, state([Oid_4,] [Rid_4,] s_2),] \end{aligned}$$

onde *rn* é o nome da regra, e Oid_i e Rid_i são respectivamente o identificador de um objeto e o identificador da instância de um papel. O primeiro predicado do lado esquerdo da regra testa se o objeto identificado por Oid_1, quando desempenhando o papel Rid_1, se encontra no estado s_1. O segundo predicado indica o recebimento de uma mensagem do objeto identificado por Oid_2 que desempenha o papel Rid_2. O recebimento da mensagem neste estado desencadeia a transição para o estado definido pelo segundo argumento do lado direito da regra, podendo esta transição ocasionar o envio de uma ou mais mensagens. Tanto a definição dos estados inicial e final da transição como das mensagens é opcional, devendo ser definido pelo menos um destes elementos de cada lado da regra. Os seguintes casos podem se apresentar:

- o estado do lado esquerdo da regra não é especificado: a regra é aplicada ao objeto em qualquer papel que ele se encontre, sempre que a mensagem for recebida;
- a mensagem do lado esquerdo não é especificada: a regra causa a transição de estados sempre que o estado inicial for alcançado;
- o estado do lado direito não é especificado: a regra causa somente o envio de uma mensagem;
- a mensagem do lado direito não é especificada: a regra causa somente uma transição de estados.

Regras de transição de estados descrevem as características do comportamento ativo de um objeto. As características do comportamento passivo (estático) são representadas através de regras de integridade. Uma regra de integridade $s_i \in S_i$ especifica um restrição sobre a evolução do objeto, devendo sempre ser satisfeita por todas as instâncias do papel em que for definida. Apresenta-se da seguinte forma:

constraint (<pré-condição> ⇒ <pós-condição>)

onde as pré e pós-condições são representadas por conjunções de predicados. Representa que, sempre que uma instância de um papel satisfizer a pré-condição, deverá também satisfazer a pós-condição. O predicados utilizados nestas condições são nomes de mensagens e predicados sobre valores de propriedades, além dos seguintes predicados pré-definidos:

• in-role ([Oid,] Rn)	verifica se um objeto identificado por *Oid* desempenha o papel *Rn*
• role ([Oid,] [Rid,] Rn)	testa se a instância identificada por *Rid* é instância do papel *Rn*
• state ([Oid,] [Rn,] S)	testa se o objeto identificado por *Oid* e que assume o papel *Rn* está no estado *S*.

Um objeto pode desempenhar diferentes papéis em tempos diferentes, pode desempenhar simultaneamente mais de um papel e pode apresentar diversas instâncias do mesmo papel. As instâncias dos papéis evoluem independentemente umas das outras, podendo haver interações através de trocas de mensagens. Considerando estas interações, três casos distintos podem se apresentar: (i) papéis independentes, quando não apresentam nenhum relacionamento; (ii) papéis coordenados, nos quais a coordenação é realizada

através de regras de transição de estados ou de integridade; e (iii) papéis que compartilham recursos, quando sua evolução é limitada por compartilhamento de recursos, sendo o controle destes recursos representado através de regras de integridade presentes em todos os papéis envolvidos.

5.1.3. Papel Básico

Todo objeto apresenta um *papel básico* R_0 que descreve as características iniciais de uma instância e as propriedades globais que controlam sua evolução. Enquanto os demais papéis de uma classe podem ser instanciados mais de uma vez, o papel básico somente pode apresentar uma instância. A criação de um objeto de uma classe instancia automaticamente o seu papel básico. As propriedades deste papel básico se aplicam a todos os demais papéis.

Na definição do papel básico de uma classe somente são definidas as propriedades e as regras. As mensagens e os estados são pré-definidos. As mensagens são utilizadas para manipular a criação e o cancelamento de instâncias de outros papéis e a sua suspensão temporária. O papel básico apresenta as seguintes mensagens pré-definidas:

- *create-object* - criação de uma instância de uma classe (um objeto);
- *suspend-object* e *resume-object* - suspensão e retomada do estado de uma instância;
- *kill* - destruição de uma instância;
- *add-role* - criação de uma instância de um papel;
- *suspend-role* e *resume-role* - suspensão e retomada do estado ativo de uma instância de um papel;
- *terminate-role* - cancelamento de uma instância de um papel;
- *forbid-role* e *allow-role* - suspensão e retomada da permissão de instanciar determinados papéis;
- *forbid-op* e *allow-op* - suspensão e retomada da permissão de receber e enviar determinadas mensagens.

Os estados do papel básico também são pré-definidos, podendo ser somente dois: (i) *active* (ativo) quando o objeto está ativo; e (ii) *suspended* (suspenso) quando está temporariamente desabilitado a enviar e a receber mensagens. A única mensagem que pode ser recebida no estado suspenso é a mensagem que o coloca novamente no estado ativo (*resume-object*). Ao ser criado, um objeto assume o estado de ativo.

As regras de transição do papel básico indicam o comportamento geral dos objetos da classe no que diz respeito à criação e ao término das instâncias da classe. São definidas as condições iniciais para os objetos recém criados, sendo especificados quais os papéis que devem ser instanciados para um novo objeto. As transições posteriores entre os papéis são definidas através destas regras. As restrições representam restrições globais do objeto, a serem respeitadas em todos os seus papéis.

5.1.4. Herança

Uma classe pode ser definida como subclasse de uma ou mais classes (herança múltipla). A subclasse herda todos os papéis especificados para as classes da qual deriva.

Novos papéis podem ser acrescentados à definição da subclasse de duas maneiras: (i) acrescentando especificações de novos papéis; e (ii) redefinindo papéis herdados. No segundo caso, a redefinição é restrita ao acréscimo de novos estados, novas mensagens e novas regras àqueles definidos na superclasse. O papel básico é definido pela união dos componentes dos papéis básicos das superclasses. O comportamento dos objetos na subclasse pode ser redefinido completamente. A desativação de determinados papéis herdados pode ser realizada através da utilização de mensagens que não permitem a criação de instâncias daquele papel. Também o conjunto de mensagens herdadas em um papel pode ser diminuído através de mensagens especiais que as inibem.

Não é permitido nenhum conflito de nomes entre papéis de superclasses. Caso isto ocorra devem ser redefinidos os nomes dos papéis nas superclasses.

5.1.5. Exemplo de Especificação em ORM

Como exemplo, consideremos uma aplicação que represente uma agência de locação de fitas de vídeo (estudo de caso do capítulo 8). A classe que representa uma fita de vídeo apresenta como propriedades do papel básico o código da fita, o nome do filme e o tipo do filme (drama, comédia, etc.). Os seguintes papéis podem ser identificados para esta classe: (1) locações (*Rentals*), representado as possíveis locações desta fita; (2) o tempo de vida de uma fita (*Life-time*), representando as ações a serem executadas para comprar uma fita, cadastrá-la na agência, colocá-la em disponibilidade para locação durante um certo período de tempo e vendê-la após este período; e (3) a perda de uma fita (*Tape-loss*), representando as ações a serem tomadas quando uma fita for extraviada.

Considerando o papel *Rentals*, algumas das propriedades que devem ser definidas são o código do cliente e as datas de início e de final de uma locação. Possíveis estados deste papel são disponível (*available*) e alugada (*rented*). As mensagens que podem ser enviadas e recebidas são as seguintes: pedido de locação (*rental*), devolução de uma fita (*tape_devolution*) e tempo de locação (*rented_time*) que este papel envia para o controle de locações. A definição desta classe é a seguinte:

```
class (
 TAPE,
 R0 = < Base_role,
        properties = {  (tape_number, integer),
                        (tape_film, string),
                        (film_type, string) },
        rules = {
                r1: msg( ←create-object ) ⇒ msg( →add-role(Life_time)),
                r2: msg( ←add-role(Tape-loss)) ⇒ msg( →terminate-role(Rentals)),
                r3: msg( ←add-role(Life-time)) ⇒ msg( →allow_role(Rentals)),
                r4: constraint( in-role(Tape-loss) ⇒ not in-role(Life-time)),
                ... }
      >,
 R1 = < Life_time,
        ...
      >,
```

```
R2 = < Rentals,
       properties = {  (client_code, integer),
                       (beginning_date, date),
                       (end_date, date) },
       messages = {    ← rental(Tape:integer, Client:integer),
                       ← tape_devolution(Tape:integer),
                       → rented_time(Time:integer, Client:integer) },
       states = { available, rented },
       rules = {
             r1 : msg(← add_role) ⇒ state(available),
             r2 : state(available), msg(← rental(T,C)) ⇒ state(rented),
             r3 : state(rented), msg(← tape_devolution(T)) ⇒
                                  msg(→ rented_time(T,C)), state(available) }
     >,
R3 = < Tape_loss,
     ...
     > )
```

5.2. F-ORM

O modelo F-ORM (*Functionality in Object with Roles Model*) [DEA 91a,b,c] é uma extensão do modelo ORM com o objetivo de ser utilizado em especificações de sistemas de informação de escritórios. As extensões feitas visam incorporar conceitos apropriados à especificação das funcionalidades deste tipo particular de sistemas de informação. Este modelo foi também desenvolvido no âmbito do projeto ITHACA.

As classes são enquadradas em dois tipos distintos, pré-definidos: *classes de recurso* e *classes de processo*. Uma classe de recursos define a estrutura de um recurso (agente, dado ou documento) em termos dos papéis que este recurso pode apresentar durante seu ciclo de vida, com propriedades, mensagens permitidas, estados internos abstratos e correspondentes regras de transição. As classes de processos integram as classes de recursos, permitindo a descrição do trabalho realizado no escritório em termos de sua organização e da cooperação entre os agentes envolvidos. Representam a manipulação de informações através de procedimentos executados no escritório. Uma classe de processo também é descrita através de papéis com propriedades, mensagens, estados e regras. Cada um destes componentes corresponde às seguintes informações manipuladas:

- cada *procedimento* executado no escritório é descrito através de uma *classe de processo*;
- os procedimentos são divididos em *tarefas*, as quais são representadas através de *papéis* da classe de processo correspondente; cada papel, com exceção do papel básico, corresponde a uma tarefa;
- as *operações* das quais se constitui uma tarefa são descritas através de *mensagens* que o papel pode enviar e receber;
- *estados pré e pós-operações* são representados através de *estados* dos papéis.

Vemos, portanto, que o conceito de papel é utilizado para especificar as diferentes tarefas executadas no processo (procedimento) e seus relacionamentos em termos de regras de comunicação e de cooperação, juntamente com os recursos envolvidos.

O processo de especificação de uma aplicação através do modelo F-ORM é decomposto em duas fases: (i) a definição das classes de recursos, independentemente das atividades nas quais estarão envolvidas; e (ii) a integração destes recursos na especificação das classes de processos.

A definição de mensagens é mais detalhada do que no modelo anterior. As mensagens enviadas e recebidas por uma classe de recurso são denominadas "mensagens de comunicação de recursos" (*resource communication messages - rc-messages*). Usualmente as mensagens enviadas pelas classes de recursos são mensagens que determinam a instanciação de papéis de classes de processos.

Para as classes de processos são identificados dois tipos de mensagens: (i) "mensagens de comunicação entre processos" (*process communication messages - pc-messages*) utilizadas para estabelecer comunicação com outros processos; e (ii) "mensagens de manipulação de recursos" (*resource manipulation messages - rm-messages*) para realizar operações sobre os recursos. Na definição das mensagens que podem ser enviadas e recebidas pelos diferentes papéis das classes de processos o modelo F-ORM estende o modelo ORM ao definir também os papéis de destino e de origem. Estes papéis podem ser do mesmo objeto ou de outro objeto. Deste modo é possível representar completamente as classes de processos, sendo especificadas a comunicação e a cooperação entre todas as tarefas representadas e definindo como os recursos são envolvidos nas mesmas.

No modelo F-ORM o conceito de especialização de classes de ORM é um pouco estendido. Uma subclasse herda todos os papéis das superclasses, podendo ocorrer as seguintes situações:

- novos papéis podem ser acrescentados aos herdados; no caso de classes de processos isto significa que o processo descrito em uma superclasse apresenta na subclasse outras tarefas além daquelas descritas na superclasse; nas classes de recursos isto significa que este recurso está envolvido na execução de outros procedimentos além daqueles descritos na superclasse.

- um papel pode ser estendido através do acréscimo de novas propriedades, novos estados, novas mensagens e novas regras; nas classes de processos isto significa uma extensão dos procedimentos descritos pelo papel; nas classes de recursos significa que este recurso está envolvido no processo descrito pelo papel executando outras tarefas além das que estava envolvido na superclasse.

- um papel pode ser totalmente redefinido; nas classes de processos uma redefinição de um papel significa uma nova definição da tarefa que está sendo descrita; para a classe de recurso significa que este recurso participa do processo descrito de uma maneira diferente daquela representada na superclasse.

A abstração de agregação é permitida somente para as classes de recursos.

No exemplo da agência de locação de fitas de vídeo apresentado em 5.1.1 as classes que representam as pessoas (clientes e funcionários) e as fitas constituem classes de recursos enquanto que o controle das locações representa uma classe de processo. A seguir apresentamos a representação parcial da classe de recurso que representa uma fita e da classe de processo que especifica os procedimentos a serem executados quando de uma locação:

```
resource class (
 TAPE,
 < Base_role,
   properties = {
                (tape_number, integer),
                (tape_film, string),
                (film_type, string) },
   rules = {    r1: msg( ←create-object ) ⇒ msg( →add-role(Life_time)),
                r2: msg( ←add-role(Tape-loss)) ⇒ msg( →terminate-role(Rentals)),
                r3: msg( ←add-role(Life-time)) ⇒ msg( →allow_role(Rentals)),
                r4: constraint( in-role(Tape-loss) ⇒ not in-role(Life-time)),
                ... }
 >,
 < Life_time,
    ...
 >,
 < Rentals,
   properties = {
                (client_code, integer),
                (beginning_date, date),
                (end_date, date) },
   rc-messages = {
                rental(Tape:integer, Client:integer) from Rent.Renting_control,
                tape_devolution(Tape:integer) from Rent.Renting_control,
                rented_time(Time:integer, Client:integer) to Rent.Accountance },
   states = { available, rented },
   rules = {    r1 : msg(← add_role) ⇒ state(available),
                r2 : state(available), msg(← rental(T,C)) ⇒ state(rented),
                r3 : state(rented), msg(← tape_devolution(T)) ⇒
                        msg(→ rented_time(T,C)), state(available) }
 >,
 < Tape_loss,
   ...
 > )

process class (
 RENT,
 < Base_role,
   properties = { ... },
   rules = {    r1: msg( ←create-object) ⇒ msg( →add-role(renting_control)),
                ... }
 >,
```

```
< Renting_control,
   properties = { ... },
   rm-messages = {
                rental_request(Tape:tape, Client:client) from Person.Client,
                rental(Tape:integer, Client:integer) to Tape.Rentals,
                tape_devolution(Tape:integer) to Tape.Rentals },
   pc-messages = {
                check_client(Client:client) to Client_control,
                client_ok from Client_control,
                client_nok from Client_control },
   states = { active, waiting_client_check,  },
   rules = {    r1 : msg(← add-role)  ⇒ state(active),
                r2 : state(active), msg(← rental_request(Tape,Client)) ⇒
                        msg( →check_client(Client)), state(waiting_client_check),
                r3 : state(waiting_client_check), msg(← client_ok) ⇒
                        msg( → rental(Tape, Client)), state(active),
                r4 : state(waiting_client_check), msg(← client_nok) ⇒ state(active)
                ... }
>,
< Accountance,
  ...
>,
< Client_control,
  ...
> )
```

6. MODELO DE DADOS TF-ORM

O principal objetivo do trabalho apresentado neste documento é a extensão do modelo de dados F-ORM [DEA 91c], apresentado no capítulo anterior, com o objetivo de representar ainda melhor as funções das aplicações a serem especificadas. Entre outras extensões realizadas, especial atenção foi dada à representação dos aspectos temporais de uma aplicação, ampliando desta maneira o poder de expressão do modelo [EDE 93a,b]. A seguir são apresentadas as extensões efetuadas no modelo F-ORM; o modelo de dados resultante denomina-se TF-ORM (*Temporal Functionality in Objects with Roles Model*). Quatro diferentes conceitos para definições temporais foram acrescentados ao modelo F-ORM: (i) pontos de tempo (*timestamps*) associados a instâncias dos objetos e a propriedades dinâmicas; (ii) um valor especial *null* representando o valor de propriedades fora do período de validade; (iii) um conjunto de tipos de dados temporais juntamente com suas funções e operações associadas, para ser utilizado na definição de domínios de propriedades; e (iv) condições temporais definidas em regras, escritas em uma linguagem de lógica temporal. Nas extensões feitas no modelo F-ORM, instantes de tempo associados às propriedades dinâmicas e o valor *null* permitem a representação dos tempos de transação e tempos válidos; a definição de um conjunto de tipos de dados temporais tem por objetivo diminuir a necessidade de definição de tempos definidos pelo usuário; e as condições temporais adicionadas às regras restringem o conjunto de possíveis transações, agindo como regras de integridade estática e dinâmica. As extensões realizadas são detalhadas nas seções a seguir. Inicialmente são apresentadas algumas extensões que têm por objetivo permitir a representação de trabalho pouco estruturado e que ampliam os mecanismos de abstrações presentes no modelo. Em seguida são detalhadas as extensões temporais.

6.1. Classes de Agentes e Decisões

Conforme visto na seção 5.2, o modelo de dados F-ORM divide as classes em dois tipos distintos: classes de recursos e classes de processos. Analisando as diversas formas que as classes de recurso podem apresentar identificamos os agentes como tendo uma funcionalidade diferente das demais, funcionalidade esta que representa a parcela de trabalho não estruturado dos sistemas de informação de escritórios: o poder de decisão humana. Para permitir a representação formal deste trabalho não estruturado foi definido um terceiro tipo de classe, denominado de *classe de agentes*.

Uma classe de agente é representada de maneira semelhante à uma classe de recurso, sendo incluída a definição de um conjunto de decisões que este agente pode tomar e estendida a forma de representação das regras de transição de estados. Uma *decisão* representa o resultado de um processo de tomada de decisão efetuado por um agente. No conjunto de decisões são simplesmente listados todos os nomes relativos a decisões que este agente pode tomar. As decisões são representadas no lado esquerdo das regras de transição de estados, condicionando um determinado estado à possibilidade de tomar uma decisão e indicando quais as conseqüências desta tomada de decisão - mudança de estado e/ou envio de mensagens. Vemos, portanto, que as regras de transição de estados de classes de agentes podem apresentar, em seu lado esquerdo, além do estado inicial da transição, ou uma mensagem recebida, ou uma decisão tomada.

Como exemplo deste tipo de classe (ver um exemplo completo no capítulo 9) consideremos uma pessoa como representada através de uma classe de agente. Um dos possíveis papéis desta pessoa é o de funcionário de uma empresa; o conjunto de decisões que esta pessoa pode apresentar inclui uma decisão sobre a escolha de um pedido de férias, resultando no período em que este funcionário gostaria de gozar suas férias. Uma regra de transição de estados processa o resultado desta decisão enviando uma mensagem ao processo que trata de férias de funcionários da seguinte maneira:

```
decisions = { ...
    vacation_request(Begin: date, End: date), ... },

messages = { ...
    ask_vacations (Emp: employee, Begin: date, End: date)
            to EMPLOYEE_CONTROL.General_Employee_Control, ... },

rules = { ...
    vacation_request:
      state(employed), decision(vacation_request(Begin, End)) ⇒
            msg(→ ask_vacations(Emp, Begin, End)), state(employed), ... }
```

6.2. Identificador de Instâncias

Tratando-se de um modelo de dados orientado a objetos, diversas instâncias de uma mesma classe (diversos objetos) podem estar presentes simultaneamente. Como este modelo utiliza o conceito de papéis, para um mesmo objeto diversas instâncias de um mesmo papel podem também estar presentes em um determinado momento. A manipulação destas diferentes instâncias, tanto de classes como de papéis, requer uma identificação de instâncias. A identificação de instâncias é necessária em condições associadas às regras de transição de estados e nas condições das regras de integridade (seção 6.9) assim como na linguagem de recuperação de dados (capítulo 7).

Ao ser criado um objeto, o sistema gerenciador do banco de dados, que implementa o modelo de dados, associa a este objeto um identificador interno, único no sistema. Cada instância de um papel deste objeto também terá associado um identificador. Assumimos que os identificadores de instâncias de papéis também apresentam valores únicos no sistema, sendo assim diferentes dos identificadores de objetos.

Embora o valor dos identificadores de instâncias seja inacessível ao usuário, ele pode ser utilizado para fins de comparações em expressões lógicas. Com a finalidade de facilitar estas comparações foram definidas duas propriedades especiais, pré-definidas no modelo:

- *oId* - propriedade presente no papel básico de todas as classes, vai armazenar o valor do identificador de cada um dos objetos desta classe; e

- *rId* - propriedade presente em todos os papéis das classes, com exceção dos papéis básicos, vai armazenar os identificadores de cada um dos papéis instanciados.

6.3. Abstrações de Especialização e de Agregação

As possibilidades de definição de abstrações presentes nos modelos preliminares não foram alteradas. Foi, entretanto, ampliado o mecanismo de herança de papéis para permitir uma maior flexibilidade na representação. Além disso foi criada uma superclasse onde estão definidas as propriedades e as mensagens pré-definidas.

6.3.1. Superclasse OBJECT

TF-ORM apresenta uma classe pré-definida denominada *OBJECT*, que desempenha o papel de superclasse para todas as demais classes definidas. As propriedades desta superclasse são herdadas por todas as suas subclasses, sejam classes de agentes, de recursos ou de processos, não podendo ser redefinidas. O papel básico de *OBJECT* contém três propriedades: uma propriedade estática (ver seção 6.4) *oId* e as propriedades dinâmicas *class_instance* e *class_end*, destinadas a armazenar respectivamente o identificador de instância da classe, o início da vida da instância da classe e seus períodos de validade, e o momento em que deixa de existir. Todas as subclasses apresentam estas propriedades no papel básico por mecanismo de herança.

A superclasse *OBJECT* apresenta um papel, denominado *ROLE*, no qual estão definidas também três propriedades: a propriedade estática *rId* e as propriedades dinâmicas *role_instance* e *role_end*. A primeira tem por finalidade armazenar o identificador de instância de um papel; o momento de criação de uma instância de um papel e seus períodos de validade são armazenados na propriedade *role_instance*; o momento a partir do qual a instância do papel deixa de existir é registrado em *role_end*. É importante notar que a existência de uma instância de um papel está condicionada à existência da instância da classe na qual é definida. Todos os papéis das classes definidas, com exceção do papel-base, herdam estas três propriedades, as quais não podem ser redefinidas.

As mensagens pré-definidas citadas na seção anterior são também definidas nesta superclasse, sendo herdadas por todas as demais classes. Podem ser enviadas por qualquer papel de qualquer subclasse. As mensagens *create_object, resume_object, suspend_object* e *kill* podem ser recebidas por qualquer papel básico das classes derivadas desta superclasse; as mensagens *add_role, resume_role, suspend_role* e *terminate_role* podem ser recebidas por qualquer papel que não seja um papel básico de classes derivadas; e qualquer uma das mensagens acima pode ser enviada por qualquer papel (básico ou não) das classes derivadas.

6.3.2. Especialização

Como nos modelos preliminares, uma classe em TF-ORM pode ser definida como *subclasse* de uma ou mais classes (herança múltipla). No modelo ORM todos os papéis das superclasses são herdados, podendo ser alterados através da definição de novos estados, mensagens e regras. Além disso, novos papéis podem ser acrescentados na subclasse. No modelo F-ORM ainda é possível redefinir um papel herdado. A diferença entre um papel que é somente estendido e um que é totalmente redefinido, entretanto, não fica explícita na definição da subclasse.

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 exceção 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; no 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(m) válido(s) 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.

Nos modelos preliminares o **papel básico** é definido pela união dos componentes dos papéis básicos das superclasses. Como, entretanto, alguns dos papéis herdados podem ter sido redefinidos, outros desabilitados e novos papéis acrescentados à subclasse, o conjunto de regras necessário para controlar estes papéis pode ser bastante diferente daquele originado pela simples união das regras dos papéis básicos das superclasses. Decidiu-se então que 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-inherit*, 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).

6.3.3. Agregação

O conceito de agregação no modelo TF-ORM é válido para as classes de recursos e para as classes de agentes. Um exemplo de agregação de classes de agentes seria uma classe representando a diretoria de uma empresa, formada da agregação de diversas pessoas, cada uma delas representada por uma classe de agente.

Entre os componentes de uma classe de recurso formada por agregação podemos ter classes de agentes. Como exemplo deste último caso suponhamos a definição de uma classe de recurso composta denominada "escritório"; os componentes desta classe podem ser a classe de agente "proprietário" e a classe de recurso "localização". É importante notar que estamos utilizando aqui a possibilidade de definir classes através de um conceito amplo de agregação, o qual semanticamente é mais abrangente do que o conceito de objeto composto como definido a partir das partes que o constituem [MOT 93]. Este último caso poderia ser exemplificado através da definição de um automóvel como composto de "rodas", "carroceria" e "motor".

A abstração de agregação não envolve herança. Deve ser tomado algum cuidado, entretando, nos nomes fornecidos às propriedades e papéis de uma classe composta de outras classes - se forem utilizados nomes iguais aos presentes em alguma das classes componentes, estes estarão inacessíveis às instâncias da classe composta.

O modelo de dados considerado permite que um objeto seja 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.

6.4. Propriedades Estáticas e Dinâmicas

O modelo F-ORM não considera a passagem do tempo e a conseqüente variação de valores de propriedades em função do tempo. Dois tipos diferentes de propriedade podem ser identificados em aplicações que evoluem com o passar do tempo. Algumas propriedades nunca mudam de valor, como por exemplo, a cor dos olhos de uma pessoa. Estas propriedades são denominadas no modelo TF-ORM de *propriedades estáticas*. Somente um valor pode ser definido para uma propriedade estática, permanecendo o mesmo constante durante toda a existência do objeto. As propriedades pré-definidas introduzidas na seção anterior - *oId* e *rId* - são representadas por propriedades estáticas.

Outras propriedades, entretanto, podem apresentar diversos valores diferentes durante a existência de um objeto. Para representar propriedades cujos valores podem variar com a passagem do tempo foi definido um novo tipo de propriedade, denominado de *propriedade dinâmica*. Todos os diferentes valores definidos para uma propriedade dinâmica devem ficar armazenados permanentemente, pressupondo a implementação de um banco de dados temporal. A história completa de uma instância pode ser recuperada através dos valores de suas propriedades dinâmicas. Informações temporais referentes à definição e à validade de seus valores são associadas aos valores destas propriedades. Uma propriedade dinâmica consiste portanto de um conjunto de mapeamentos associando as informações temporais aos valores definidos para a propriedade [CLI 88a]. O domínio das propriedades dinâmicas é composto pela concatenação de domínios temporais e de domínios referentes aos valores das propriedades. Na linguagem de representação do modelo TF-ORM definimos somente o domínio dos valores das propriedades dinâmicas, ficando os domínios temporais implícitos.

Como exemplo, propriedades estáticas e dinâmicas são definidas no papel *empregado* da classe *PESSOA*:

```
Employee,
    static properties = { (gender, {F, M}) },
    dynamic properties = { (name, STRING), (salary, REAL) }
```

Para representar os períodos de tempo durante os quais as propriedades apresentam valores indefinidos foi introduzido um valor especial *null*. No momento de criação de uma instância todas as suas propriedades recebem um valor default *null*. Às propriedades estáticas é permitida somente uma alteração deste valor - o novo valor definido será mantido durante toda a existência da instância. Para as propriedades dinâmicas, o valor *null* é mantido até o momento em que um novo valor é definido. Em uma aplicação podem ocorrer períodos de tempo em que uma propriedade dinâmica apresenta um valor indefinido. Estes períodos são representados associando novamente o valor *null* a esta propriedade. O valor especial *null* está implicitamente definido como fazendo parte de todos os domínios definidos para propriedades.

6.5. Representação Temporal e Elemento Temporal Primitivo

No modelo de dados TF-ORM consideramos o tempo linearmente ordenado, variando de forma discreta.

A variação de valores armazenados em um banco de dados que implemente este modelo é considerada da forma de escada [SEG 88a,b] - uma vez definido um valor, este permanece válido até que outro valor seja definido.

O elemento temporal primitivo adotado para a representação de aspectos temporais em TF-ORM foi o *ponto no tempo*. Todos os pontos pertencem a uma ordenação total. Intervalos serão representados através de um par de pontos no tempo, os quais definem seus limites. O instante atual é considerado como um ponto especial no tempo, variando constantemente ao longo do eixo deste tempo.

Uma análise do domínio de sistemas de informação de escritórios identificou o *minuto* como a menor granularidade temporal adequada para atividades humanas. Este foi, portanto, escolhido como *chronon* do modelo de dados - menor intervalo de tempo entre quaisquer dois pontos no tempo. A definição completa de um ponto no tempo é dada, portanto, pela definição de uma data (ano, mês e dia) e por um horário dentro desta data (hora e minuto).

6.6. Tipos de Dados Temporais

A definição dos domínios das propriedades no modelo F-ORM é feita através da utilização de nomes de classes de objetos ou de tipos pré-definidos. Os tipos pré-definidos apresentam um determinado domínio pré-definido para seus valores, como por exemplo os inteiros (*integer*).

Uma das extensões realizadas no modelo F-ORM foi a definição de um conjunto de classes pré-definidas para serem utilizadas como domínios temporais, classes estas que têm associada alguma semântica temporal. Estas classes são denominadas de *tipos de dados temporais*. Através dos tipos de dados temporais podem ser representados os requisitos temporais incondicionais bem-definidos (seção 3.2.2.1.). Estes tipos de dados apresentam diferentes granularidades temporais, tais como hora, ano e intervalo. As diferentes granularidades são necessárias para que se possa especificar a realidade de uma maneira natural através de conceitos utilizados na prática.

O modelo F-ORM apresenta os seguintes domínios pré-definidos: *boolean*, *date*, *image*, *integer*, *place*, *string*, *text*, *time*, *title*. Dois tipos de dados temporais, a data (*date*) e o tempo (*time*), já se encontram definidos neste conjunto. Entretanto, dependendo da aplicação a ser especificada, outros tipos temporais se fazem necessários.

Quatro diferentes tipos de dados podem ser identificados [ADI 85, 86b, 87]: pontos no tempo, intervalos, duração e período. Decidimos introduzir somente os três primeiro tipos no modelo, considerando que o quarto tipo pode ser definido através de regras de restrições aplicadas a intervalos.

6.6.1. Pontos no Tempo

O tipo temporal básico é composto por:

instant ::= <ano> <mês> <dia> <hora> <minuto>

Foi definido um conjunto de tipos temporais para representar informações temporais específicas, derivados do tipo básico através de um mecanismo baseado em restrições. Os tipos de dados temporais pré-definidos do modelo F-ORM, *date* (data correspondente ao calendário gregoriano) e *time* (tempo do relógio interno do computador) são considerados aqui como tipos derivados:

date ::= <ano> <mês> <dia>
time ::= <hora> <minuto>

Os demais tipos temporais derivados são os seguintes:

year	-	ano
month	-	mês
day	-	dia
hour	-	hora
minute	-	minuto
week	-	semana
semester	-	semestre
century	-	século
weekday	-	dia da semana - domingo a sábado

Cada um destes tipos apresenta restrições implícitas de valores permitidos - por exemplo, $0 \leq$ *minute* ≤ 60.

O valor *now* é um ponto de tempo especial que se desloca constantemente ao longo do eixo dos tempos. Corresponde ao momento atual. Qualquer dos tipos definidos aceita este valor. Pode ser utilizado em comparações, nas regras de integridade e nas condições utilizadas nas regras. Assume o tipo para o qual está sendo utilizado, podendo portanto corresponder ao instante atual, ao mês atual, ao ano atual.

Um exemplo de definição de classe utilizando um tipo de dados temporal é o seguinte:

```
agent class (
    PERSON,
    < base_role,
            static properties = { (birth_date, date), ... }, ...
    >
```

6.6.2. Intervalos

Intervalos de tempo são utilizados para definir todos os instantes entre dois pontos de tempo. É suposta uma distribuição uniforme de pontos no tempo dentro do intervalo, devendo o primeiro limite ser anterior ao segundo no tempo. Os limites devem apresentar a mesma granularidade temporal, a qual define o *chronon* para pontos internos do intervalo. Os tipos temporais que podem ser utilizados como limites de intervalos temporais são *instant*, *date*, *time*, *year*, *month*, *day*, *hour* e *minute*.

Seis tipos diferentes de intervalos podem ser definidos, dependendo da pertinência ou não dos pontos limites ao intervalo: *intervalos fechados* quando ambos os limites pertencem ao intervalo, *intervalos semiabertos abaixo* e *intervalos semiabertos acima* quando somente um dos limites pertence ao intervalo, *intervalos abertos* quando nenhum dos limites pertence ao atual; e *intervalos flutuantes abaixo* e *intervalos flutuantes acima* nos casos em que um dos limites é representado pelo momento atual (*now*).

A definição de um domínio como sendo do tipo *interval* requer, portanto, a definição complementar do tipo de seus limites e do tipo de intervalo considerado.

Exemplos de definições de propriedades com domínio intervalo podem ser encontradas na definição do papel de funcionário de uma pessoa:

```
< Employee,
        dynamic properties = {
                (admission_date, date),
                (hours_week, day),
                (vacation, interval(date, closed)), ... }, ...
>
```

6.6.3. Duração

Outro tipo de dado muito utilizado na especificação de sistemas é a duração de uma atividade. Esta informação é representada por um número inteiro seguido de uma unidade de tempo apropriada - dias, horas, semanas. Um exemplo deste tipo de dado é o tempo que uma fita é mantida em uma locadora antes de ser colocada à venda:

```
< Life_time,
        dynamic properties = {
                (time, span(year))
                ...
```

6.6.4. Tipos de Dados para Informações Temporais Incompletas

Em algumas aplicações é necessária a representação de informações temporais incompletas. Para estas TF-ORM apresenta alguns tipos temporais específicos. Quando um requisito apenas especifica que um evento deve ocorrer antes ou depois de um determinado instante, data ou hora, um dos seguintes tipos pode ser utilizado:

<data limite> ::= *after* "(" <tipo do limite> ")" | *before* "(" <tipo do limite> ")"
<tipo do limite> ::= *instant* | *date* | *time* | *year* | *month* | *day* | *hour*| *minute*

Este tipo de dado tem o mesmo significado que aquele definido por um intervalo com um limite igual a infinito, considerando válido somente um ponto no intervalo, ponto este não definido.

6.6.5. Funções e Operações sobre os Tipos Temporais

A utilização de tipos de dados com diferentes granularidades provoca algumas dificuldades na manipulação e operação com tempos diferentes [CLI 88b, WIE 91]. Para que esta manipulação seja possível foi definido um conjunto de *funções* (predicados).

As **funções** podem ser classificadas em:

a) funções que mudam a granularidade temporal de uma informação, convertendo o valor para a granularidade requerida. São as seguintes:

to_minutes (<instante | horário>)	converte o horário total em minutos
to_months (<instante | data>)	converte a data em meses
to_days (<instante | data>)	converte a data em dias

b) funções que retornam uma informação temporal, calculada a partir de um valor temporal fornecido. São as seguintes:

month (<instante | data>)	extrai o mês de um ponto no tempo (truncamento)
day (<instante | data>)	extrai o dia do ponto no tempo (truncamento)
hour (<instante | horário>)	extrai as horas (truncamento)
minute (<instante | horário>)	extrai os minutos (truncamento)
weekday (<instante | data>)	dia de semana correspondente à data
lower_bound (<intervalo>)	limite inferior de um intervalo
upper_bound(<intervalo>)	limite superior do intervalo
duration (<intervalo>)	calcula a duração de um intervalo

c) funções booleanas que comparam a posição temporal relativa de dois pontos no tempo ou de dois intervalos, tais como os 13 relacionamentos da lógica intervalar de Allen [ALL 83]. Foram definidas as seguintes funções:

before (<instante>:<instante>)
verdadeiro quando o primeiro instante for anterior ao segundo
equal (<instante>:<instante>)
verdadeiro quando os dois instantes forem iguais
belongs (<instante>,<intervalo>)
verdadeiro quando o instante pertencer ao intervalo
contains (<intervalo>,<intervalo>)
verdadeiro quando o primeiro intervalo estiver contido no segundo

Foram definidas **operações** envolvendo valores temporais, podendo estas ser classificadas em:

a) operações aritméticas, tais como soma e subtração, podem ser aplicadas em alguns casos particulares, sumarizados na tabela 6.1: (i) quando um dos operandos representa um ponto no tempo representado por uma data, um horário ou um instante, e o outro uma duração, resultando um valor do mesmo tipo correspondente ao ponto no tempo; (ii) quando os dois operandos são do tipo duração, sendo que estas durações são

medidas através da mesma unidade, resultando em uma duração medida nesta mesma unidade; (iii) quando os dois operandos são do tipo *week*, *semester* e *century*, sendo o resultado deste mesmo tipo; e (iv) quando os dois operandos são intervalos de mesma granularidade temporal, resultando um outro intervalo ou um resultado indefinido.

b) operações aritméticas tais como multiplicação e divisão, aplicáveis quando um dos operandos é um valor numérico e o outro for: (i) uma duração, sendo o resultado definido pela granularidade em que é expressa a duração; ou (ii) um valor de granularidade *minute*, *hour*, *day*, *month*, *year*, *week*, *semester* ou *century*, sendo o resultado dado pela granularidade deste segundo operando;

c) operações lógicas determinadas pelos operadores relacionais "<" (menor), ">" (maior), "=" (igual), "≤" (menor do que), "≥" (maior do que) e "≠" (diferente) que podem ser utilizadas para comparar dois pontos de tempo ou dois valores do tipo duração (*span*) de diferentes granularidades, sendo os valores convertidos internamente para a menor granularidade; as relações resultam em um valor lógico; para pontos no tempo são permitidos os tipos *instant*, *date*, *time*, *year*, *month*, *day*, *hour* e *minute*;

c) operações sobre conjuntos para serem aplicadas a intervalos, tais como (i) *union* (união) e *intersection* (intersecção), resultando intervalos ou valores indefinidos; e (ii) *contains*, resultando valores lógicos.

Tabela 6.1 : Operações de Soma e Subtração

DOMÍNIO DO PRIMEIRO OPERANDO	DOMÍNIO DO SEGUNDO OPERANDO	DOMÍNIO DO RESULTADO
date	span (day | month | year)	date
time	span (hour | minute)	time
instant	span (day | month | year | hour | minute)	instant
span (x)	span (x)	span (x)
week	week	week
semester	semester	semester
century	century	century
interval (x,y)	interval (x,y)	interval (x,y)

6.7. Tempo de Transação e Tempo de Validade

Dois diferentes conceitos de tempo devem ser representados em uma aplicação - o tempo de transação e o tempo de validade (seção 3.3.8). O tempo de transação corresponde ao tempo em que uma informação é armazenada em um banco de dados, e o tempo de validade ao tempo em que a informação modela a realidade. A definição do tempo de transação é feita implicitamente, pelo SGBD. O tempo de validade deve ser fornecido pelo usuário do sistema, podendo ser igual ou diferente ao tempo de transação. Como exemplo de aplicação na qual o tempo de transação pode ser diferente do tempo de validade consideremos a atualização de salário de um empregado. Suponhamos, por exemplo, que no dia 5 de maio (tempo de transação) foi definido um novo salário para um

funcionário. Este valor de salário corresponde ao mês de maio todo, ou seja, já era válido no dia 1 de maio (tempo de validade).

No modelo TF-ORM são considerados tanto o tempo de transação como o tempo de validade. Ambos devem ser armazenados no banco de dados que modela a aplicação, constituindo-se este de um banco de dados bitemporal (seção 3.3.9).

Consideramos como tempo de validade o tempo correspondente ao início da validade da informação, a qual continuará válida até o início da validade de outro valor, definido como tempo de validade deste. A validade de uma informação é definida pela última transação realizada. Consideremos, por exemplo, três tempos de transação t_1, t_2 e t_3 tais que $t_1 < t_2 < t_3$. Suponhamos que no instante t_1 seja definido um valor a com tempo de validade (início de sua validade) correspondente a v_1 e que este valor a deverá valer somente no intervalo temporal entre v_1 e v_2. Para indicar o final do período de validade deste valor basta definir um novo valor para a propriedade considerada, indicando que seu valor é *null* a partir do instante de tempo v_2 (com este tempo de validade). Esta segunda definição é feita no instante de tempo correspondente ao tempo de transação t_2. O valor da propriedade considerada permanecerá sendo *null* até que um novo valor seja definido para ela - por exemplo em um instante de tempo t_3 é definido o valor b com tempo de validade v_3. Nas figuras 6.1 e 6.2 são representadas duas possíveis situações para estas definições, conforme a posição temporal relativa dos tempos de validade: na primeira temos $v_1 < v_2 < v_3$ e na segunda, $v_1 < v_3 < v_2$.

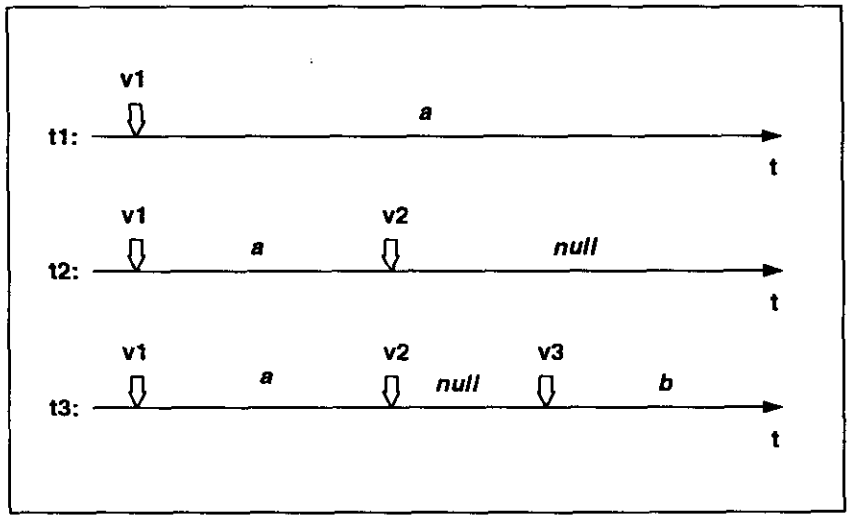

Figura 6.1: Tempo de Transação e Tempo de Validade - $v_1 < v_2 < v_3$

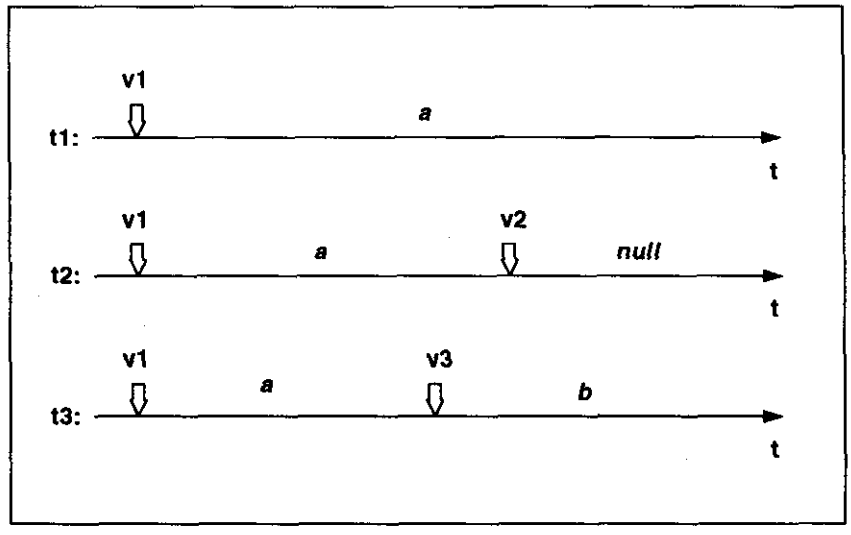

Figura 6.2: Tempo de Transação e Tempo de Validade - $v_1 < v_3 < v_2$

6.8. Associação de Tempo no Paradigma de Orientação a Objetos

Os modelos de dados temporais associam informações temporais às informações contendo dados. Esta associação pode ser feita em dois níveis diferentes: (i) a nível de conjunto de informações, como por exemplo, de tuplas; e (ii) a nível de atributos. Na primeira forma de representação, cada vez que uma nova informação é armazenada, todo o conjunto que contém esta informação com suas informações temporais correspondentes deve ser novamente armazenado. Na segunda, variações de atributos isolados são armazenadas com as suas informações temporais correspondentes, permitindo que diferentes granularidades temporais sejam utilizadas para os diferentes atributos. Formas mistas são muitas vezes utilizadas, representando algumas informações temporais a nível de conjunto de informações e outras, a nível de atributos.

Considerando o paradigma de orientação a objetos, duas formas podem ser utilizadas para representar o tempo: associar o tempo ao objeto como um todo, ou a cada uma de suas propriedades. O modelo de dados OSAM* [SU 91] associa as informações temporais aos objetos; nos modelos OODAPLEX [WUU 93] e TOM [SCH 91] o tempo é asssociado a nível de objeto e de propriedades. Na extensão temporal realizada neste trabalho ambas as formas foram consideradas importantes e necessárias para a completa representação das informações. O tempo associado às propriedades permite a definição dos diferentes valores que as propriedades assumem com a passagem do tempo. A associação do tempo aos objetos é necessária para representar as informações temporais referentes à criação do objeto, sua destruição e eventuais suspensões de suas atividades. Ambas as formas são utilizadas no modelo TF-ORM.

6.8.1. Associação de Tempo às Propriedades

Duas formas diferentes podem ser utilizadas para armazenar os tempos de validade: (i) definir as propriedades dinâmicas com três dimensões, representando respectivamente o tempo de transação, o tempo válido e o domínio da informação; e (ii)

utilizar duas dimensões na definição de propriedades dinâmicas, representando pares tempo e domínio da informação, e utilizando propriedades dinâmicas especiais para guardar o tempo válido.

No caso de utilização da segunda opção acima, as propriedades referentes ao tempo de validade estariam definidas somente nos casos em que este diferisse do tempo de transação. Neste caso, para cada propriedade dinâmica da modelagem existe a definição implícita da propriedade dinâmica de validade. Como exemplo de utilização desta segunda opção, consideremos a definição do papel empregado, onde as propriedades representando valores temporais de validade são representados pelo prefixo *valid_*:

```
< Employee,
    dynamic properties = {
            (salary, REAL), (hire_date, DATE),
            (end_date, DATE), (function, INTEGER)   },
                        ...
```

Implicitamente esta definição corresponde a:

```
< Employee,
    dynamic properties = {
            (salary, REAL), ( valid_salary, REAL),
            (hire_date, DATE), (valid_hire_date, DATE),
            (end_date, DATE), (valid_end_date, DATE),
            (function, INTEGER), (valid_function, INTEGER)   },
                        ...
```

Analisando as aplicações do domínio considerado, verificamos que a grande maioria das propriedades dinâmicas pode apresentar tempos de validade diferentes dos de transação, gerando um esquema conceitual bastante extenso. Para que isto não aconteça optou-se, no modelo TF-ORM, pela associação dos dois tempos à informação. Cada valor armazenado para uma propriedade dinâmica é representada, portanto, através de uma tripla:

< TEMPO DE TRANSAÇÃO, TEMPO DE VALIDADE, INFORMAÇÃO >

Quando não for definido o tempo de validade de uma informação, este é considerado como sendo igual ao de transação. O tempo de validade utilizado corresponde ao início da validade da informação, podendo ser anterior ou posterior ao tempo de transação.

Os valores armazenados no banco de dados são fornecidos através de argumentos em mensagens enviadas e recebidas por papéis. Um argumento especial é utilizado na linguagem do modelo TF-ORM para definir tempos válidos: *Valid_Time*. Cada mensagem pode apresentar somente um destes argumentos, correspondendo ao tempo de validade de todas as definições a serem efetuadas pela mensagem. O domínio deste argumento pode ser dos tipos INSTANT, DATE ou HOUR. Este argumento é opcional, sendo utilizado somente quando o tempo válido é diferente do tempo de transação. Quando existente, deverá ser o último argumento da lista de argumentos. Duas mensagens possíveis para o exemplo de controle de um funcionário são:

```
messages = {
    modify_salary( Value:REAL, Valid_Time:DATE) from employee_control,
    end_employment (Valid_Time:DATE) from employee_control }
```

6.8.2. Associação de Tempo às Instâncias

A associação de informações temporais às instâncias tem por objetivo representar a manipulação da instância na passagem do tempo - sua criação, eventuais suspensões e reativações e sua destruição. O modelo considerado apresenta instâncias de classes e instâncias de papéis - a ambas as formas deve ser feita a associação de tempo. Estas informações são armazenadas em propriedades especiais, presentes em todas as classes do modelo.

As ***instâncias de classes*** são manipuladas através de mensagens especiais, pré-definidas. A mensagem *create_object* determina a criação de uma instância de uma classe. No momento de criação de uma instância, o seu identificador interno, definido pelo sistema gerenciador do banco de dados, é armazenado na propriedade estática *oId*. O valor contido nesta propriedade pode ser utilizado para identificar a instância em condições lógicas e na linguagem de consulta. As informações temporais correspondentes ao tempo de criação (tempo de transação e tempo de validade) desta instância são armazenadas na propriedade dinâmica pré-definida *object_instance*. Esta propriedade está armazenada no papel básico de todas as classes e tem por objetivo armazenar o início da vida de um objeto e seus períodos de validade. Pode apresentar somente dois valores especiais para representar a vida de uma instância - *null* e *nonull*. A existência de um objeto inicia no momento de criação de uma instância de uma classe. O início da vida do objeto é representado na propriedade *object_instance* cujo valor passa a ser *nonull*; o tempo de transação corresponde ao início da vida deste objeto. Um objeto pode apresentar períodos disjuntos de validade, dependendo do recebimento das mensagens *suspend_object* (suspende temporariamente a existência da instância) e *resume_object* (retorna a validade à instância suspensa). Sempre que um objeto receber uma mensagem de suspensão (*suspend_object*) o valor da propriedade passa a ser *null* indicando que a instância está suspensa, não podendo enviar nem receber mensagens (com exceção do recebimento das mensagens de reativação ou do término da vida do objeto); o valor do tempo de transação corresponde ao instante em que a mensagem de suspensão foi recebida e o tempo de validade ao início da validade desta suspensão. Se não for fornecido o tempo de validade da suspensão, esta será efetivada no mesmo momento de recebimento da mensagem. O recebimento de uma mensagem de reativação do objeto (*resume_object*) faz com que o valor da propriedade volte a ser *nonull*, tendo associado os tempos de recebimento da mensagem e o tempo de validade desta reativação.

O recebimento da mensagem *kill* termina o tempo de vida de uma instância. Como o banco de dados correspondente a este modelo será um banco de dados bitemporal, todos os valores serão sempre mantidos, não havendo nenhuma remoção de informações definidas. A mensagem *kill* desativa a vida útil de uma instância, porém seus valores serão mantidos no banco de dados. Para manter a informação do final da vida de uma instância foi criada a propriedade dinâmica pré-definida *end_object*, a qual armazena o instante de tempo em que a instância foi descontinuada. Também esta propriedade está definida em todo papel básico das classes. O tempo de transação associado a esta

propriedade representa o instante de recebimento da mensagem de *kill*. Também esta mensagem pode definir um tempo de validade para o final da vida da instância diferente do tempo de recebimento da mensagem. O valor contido na propriedade não é relevante, somente os tempos de transação e de validade associados. Cada instância de classe pode apresentar somente uma definição desta propriedade, pois a ela corresponde a finalização da vida do objeto. Quando for finalizada a vida de um objeto, todas as instâncias de papéis deste objeto também são finalizadas pelo sistema gerenciador.

Na figura 6.3 estão representadas as mensagens de manipulação de instâncias de classes e sua atuação sobre a propriedade *object_instance*.

As ***instâncias de papéis*** são manipuladas através das mensagens pré-definidas *add_role*, *resume_role*, *suspend_role* e *terminate_role*, de forma análoga às instâncias dos objetos. Além destas, duas outras mensagens pré-definidas colocam um papel em duas condições distintas: (i) a mensagem *allow_role* enviada pelo papel básico a um papel não cria instância deste papel mas permite que instâncias deste papel sejam criadas por outros papéis desta ou de outras classes; e (ii) a mensagem pré-definida *forbid_role* impede temporariamente a criação de instâncias deste papel. Todo papel, com exceção do papel básico, apresenta duas propriedades dinâmicas pré-definidas que têm por objetivo armazenar os tempos de criação, suspensão e reativação e do final de vida de instâncias de papéis. A propriedade dinâmica pré-definida *role_instance* armazena o instante de criação e as eventuais suspensões e reativações da instância do papel no qual está definida; e a propriedade dinâmica *role_end* armazena os tempos de transação e de validade do final da vida desta instância de papel.

A figura 6.4 representa as mensagens de manipulação de instâncias de papéis e sua atuação sobre a propriedade *role_instance*.

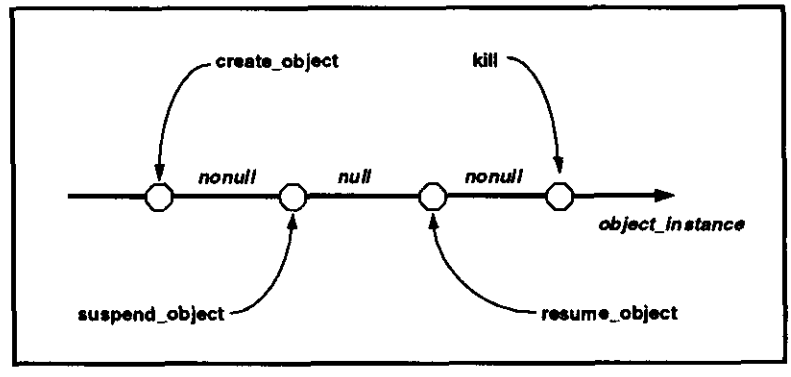

Figura 6.3 : Mensagens de Manipulação de Instâncias de Classes

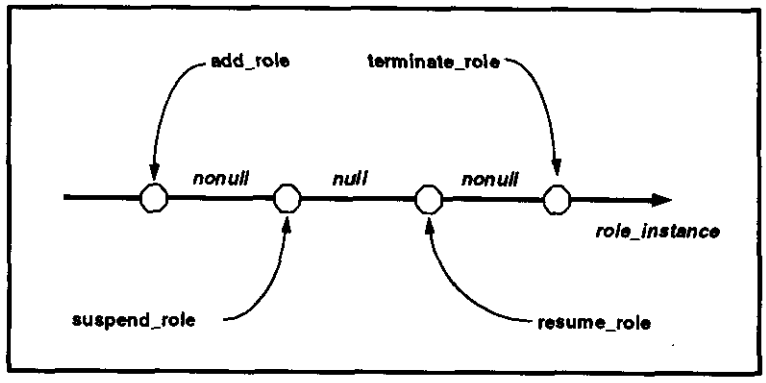

Figura 6.4 : Mensagens de Manipulação de Instâncias de Papéis

As mensagens que manipulam as instâncias de papéis estão condicionadas à atividade da instância da classe considerada - elas somente serão atendidas se a instância da classe correspondente estiver ativa.

6.9. Regras e Condições Temporais

O principal objetivo da extensão realizada no modelo de dados F-ORM, a qual deu origem ao modelo TF-ORM, foi permitir a representação de todos os possíveis aspectos temporais de uma aplicação. Dentre estes, especial atenção foi dada a restrições temporais.

Um modelo de dados convencional pode ser estendido para representar restrições temporais de duas maneiras distintas: (i) acrescentando uma condição à definição de transição de estados de um objeto, restringindo desta maneira as possíveis transições; e (ii) representando definições de integridade específicas as quais devem ser sempre satisfeitas (em todos os momentos) por todas as instâncias dos papéis.

Como visto na seção 5.1, o modelo F-ORM apresenta dois tipos de regras; regras de transição de estados e regras de integridade. Uma transição de estados é realizada sempre que a instância estiver em um estado inicial e receber uma determinada mensagem. O modelo TF-ORM estende as regras de transição de estados através da possibilidade de definição de uma condição associada a cada uma delas; somente se a condição for satisfeita é que a transição, dadas as condições de estado inicial e de mensagem recebida, será executada.

As condições a serem especificadas nas definições de integridade podem ser temporais, sendo avaliadas sobre todo o conteúdo passado do banco de dados. Uma linguagem de lógica temporal é utilizada no modelo TF-ORM para expressar condições dos dois tipos de regras do modelo de dados - das regras de transição de estados e das condições que constituem as regras de integridade. Estas condições podem ser utilizadas para representar restrições temporais respectivamente às transições de estado do banco de dados e ao estado corrente. A utilização de linguagens de lógica para expressar as condições temporais pode ser encontrada em diversos sistemas e linguagens publicados [ARA 91, BOL 83, CAS 88, COR 91, GRE 86, LIP 87, LOU 91a,b, MAI 92, SCH 83].

A definição das condições em TF-ORM utiliza uma linguagem de lógica temporal de primeira ordem. Lógica de primeira ordem permite manipular somente elementos individuais de um domínio, sendo que nas lógicas de ordens mais elevadas são consideradas construções complexas, tais como conjuntos, relações e funções. Em termos de bancos de dados, através de lógica de primeira ordem somente entidades elementares podem ser dados. Em lógicas mais elevadas, conjuntos, funções e relações também podem ser dados. O motivo pelo qual é geralmente utilizada a lógica de primeira ordem em lugar de uma variante mais elevada (e portanto mais expressiva) é que esta apresenta uma axiomatização precisa e completa; por exemplo, na lógica de primeira ordem as denotações de "provável" e "verdadeiro" coincidem, propriedade esta que falha na lógica de segunda ordem [BEE 90].

A seguir apresentamos uma análise detalhada dos dois tipos de regras presentes em TF-ORM; em seguida é apresentado um conjunto de predicados e de funções a serem utilizados nas condições destas regras; para finalizar, é definida a sintaxe e a semântica da linguagem de lógica temporal de primeira ordem através da qual as condições devem ser expressas.

6.9.1. As Regras de TF-ORM

TF-ORM apresenta dois tipos de regras: regras de transição de estados e regras de integridade.

Regras de transição de estados de TF-ORM definem, para cada papel, as possíveis transições de estados que podem ocorrer. A forma mais geral de uma regra de transição de estados é a seguinte:

rn: state(s1), ←msg(m1) ⇒ →msg(m2), state(s2); (<condição de transição>)

A regra expressa que quando o papel considerado do objeto está no estado *s1* (estado inicial), recebendo a mensagem *m1* e satisfazendo a condição de transição, é executada a transição para o estado *s2* (estado final), causando o envio de uma ou mais mensagens *m2*. Uma regra de transição não precisa apresentar todos os elementos acima. Formas alternativas de representação destas regras são as seguintes:

(i) rn: ←msg(m1) ⇒ →msg(m2), state(s2) [; (<condição de transição>)]
(ii) rn: state(s1), ←msg(m1) ⇒ →msg(m2) [; (<condição de transição>)]
(iii) rn: state(s1) ⇒ →msg(m2), state(s2) [; (<condição de transição>)]
(iv) rn: state(s1), ←msg(m1) ⇒ state(s2) [; (<condição de transição>)]

Quando o estado inicial não for definido (i), a regra é válida para todos os estados, dependendo sua execução somente da recepção da mensagem e da condição. Quando o estado final não for definido (ii), a transição se efetua através da troca de mensagens e o estado permanece o mesmo. Quando a mensagem do lado esquerdo da regra não for definida (mensagem recebida) (iii), a transição é efetuada independentemente da chegada de alguma mensagem, sempre que a condição for satisfeita. E quando a mensagem do lado direito não for definida (mensagem enviada) (iv), a transição para outro estado ocorre mas nenhuma mensagem é enviada. A condição de transição também é

opcional - quando não definida, a transição é executada dependendo somente do estado inicial e da mensagem de chegada. A existência da condição de transição restringe as possíveis transições - uma transição somente será efetuada quando a condição for satisfeita.

Quando a regra representa o caso em que não é feita a definição do estado s1, a chegada da mensagem m1 causará a transição para o novo estado independentemente de qual o estado atual. Isto torna possível a modelagem de *objetos ativos*, objetos que apresentam um comportamente definido quando do recebimento de um mensagem com argumentos temporais pré-definidos. Nestes casos, um objeto do tipo relógio envia mensagens (eventualmente mensagens virtuais) a intervalos de tempo definidos para todos os objetos ativos. O acréscimo de uma condição a este tipo de regra de transição limita o comportamento destes objetos - a regra será executada no momento em que esta condição seja satisfeita, sem considerar o estado atual.

Regras de integridade devem ser sempre (em todos os momentos) satisfeitas, por todas as instâncias do papel. Apresentam-se da seguinte maneira:

constraint (<pré-condição> ⇒ <pós-condição>)

Sempre que uma instância do papel satisfaz a pré-condição, deve também satisfazer a pós-condição. As regras de integridade representam um comportamento passivo na evolução dos papéis - não causam transições de estado, mas restringem as possíveis evoluções dos objetos uma vez que não permitem transições que gerem conteúdos do banco de dados que não satisfazem as condições. Considerando um banco de dados correspondente a uma aplicação, as regras de integridade avaliam o conteúdo atual deste banco de dados. O conjunto de condições de integridade representa o conteúdo válido do banco de dados. Conteúdos inconsistentes não serão permitidos.

Sempre que uma regra de integridade não é satisfeita, o último processo executado deve ser desfeito para retornar o banco de dados ao último estado consistente. Este processo corresponde à execução de algum método referente a uma mensagem recebida, representada em alguma regra de transição executada. O momento exato da avaliação das regras de integridade deve ser, portanto, a cada vez que uma regra de transição de estados foi habilitada, logo antes do envio da mensagem de saída correspondente. Se as restrições de integridade do papel considerado forem todas satisfeitas ocorre o envio da mensagem de saída. Caso alguma restrição de integridade não seja satisfeita, a transição será desfeita através de um processo de recuperação de erro, permanecendo deste modo o banco de dados consistente - não haverá mudança de estado, os valores alterados pelo método correspondente à mensagem de chegada são restaurados aos seus valores anteriores e a mensagem de saída não será enviada. Uma mensagem de *NAck* (não atendimento da mensagem de chegada) deverá ser enviada ao papel que desencadeou a transição (que enviou a mensagem de entrada). A recuperação de erros é parte fundamental da implementação do sistema que utilize o modelo TF-ORM, podendo ser efetuada de diversas maneiras. Uma forma possível é através de um mecanismo de especificação de mensagens denominado MPS (Message Pattern Specification) [PUR 90], o qual associa a cada classe métodos a serem ativados para tratar erros.

Uma das principais diferenças entre uma condição de transição e uma condição de integridade é o momento de sua avaliação. As condições de integridade devem ser avaliadas a cada vez que um novo estado do banco de dados é criado - a cada vez que a instância alcance um novo estado - correspondendo ao momento logo após a execução de uma transição. Se neste momento for detectada a violação de uma restrição, a última transição deve ser desfeita, devendo o estado e os valores da instância serem restaurados para a situação anterior à transição. O momento de avaliação de uma condição de transição, no entanto, é anterior à execução da transição. Conseqüentemente, as condições de integridade detectam a ocorrência de uma violação de integridade enquanto que as condições de transição não permitem a ocorrência de violações.

Outra diferença fundamental entre os dois tipos de condição se refere à representação de restrições de integridade. Condições de transição podem ser utilizadas para reforçar restrições de integridade - restrições que devem ser satisfeitas para permitir a transição de estados. Mas uma condição de transição somente reforça uma restrição de integridade quando uma transição específica é considerada, enquanto que as regras de integridade se aplicam a todas as transições de estado do papel. Quando uma regra de integridade é definida, parte-se da suposição que diferentes formas de violação desta integridade podem ocorrer, causadas por diferentes transições de estado. Sempre é possível substituir uma regra de integridade por um conjunto de regras de transição, associadas a todas as transições de estado que podem causar a violação, mas o contrário nem sempre é possível. Existem condições de transição especiais para as quais não é possível definir uma regra geral. É possível, no entanto, representar todas as intâncias válidas do banco de dados utilizando somente regras de transição de estados.

Diversas restrições de integridade podem ser representadas por qualquer uma das duas formas de condições - as duas formas representam um nível diferente de representação: condições de transição de estados representam o ponto de vista operacional, enquanto que condições de integridade representam um nível de abstração de modelagem mais elevado. O gerenciador de banco de dados utilizado para implementar a aplicação é que vai definir qual dos dois tipos de condição deve ser utilizado - se estiver disponível um monitor para as condições de transição ou um monitor de condições de integridade, ou ambos.

Restrições de integridade dinâmica também podem ser representadas. Consideremos um banco de dados implementando uma aplicação segundo o modelo de dados TF-ORM. O comportamento deste banco de dados com a passagem do tempo pode ser caracterizado por uma seqüência de conteúdos do banco de dados, cada um representando o conteúdo do banco de dados em um instante isolado. Restrições de integridade dinâmica são utilizadas para especificar seqüências admissíveis destes conteúdos do banco de dados [LIP 87]. Isto pode ser efetuado através de uma condição escrita em lógica temporal, a qual compara o conteúdo atual do banco de dados com algum (conjunto de) conteúdo passado. Quando são utilizadas condições de transição, o conteúdo dos argumentos da mensagem recebida permite inferir os próximos conteúdos do banco de dados. Através destes valores a condição pode simular os próximos conteúdos do banco de dados, permitindo assim comparar os conteúdos atuais com os próximos.

Considerando os tipos de classes do TF-ORM - classes de agentes, de recursos e de processos - as regras são utilizadas para representar informações específicas a cada classe. Nas classes de recursos e de agentes as regras modelam as possíveis transições dos diferentes papéis que os recursos podem desempenhar durante sua vida, respeitando as restrições de integridade dos recursos. Nas classes de processos as regras representam como as diferentes tarefas se comportam mediante a recepção e o envio de mensagens (regras de transição de estados) e as restrições de integridade que devem ser mantidas durante a evolução das tarefas (condições de transição e regras de integridade). As regras que representam os processos de decisão humana das classes de agentes se enquadram nesta segunda forma de interpretação.

Um ponto importante a ser analisado na construção das condições de transição e nas condições das regras de integridade é quais as informações que podem ser referenciadas. O banco de dados implementando uma aplicação segundo o modelo TF-ORM deve ser bitemporal - cada informação será rotulada com dois tempos, o tempo de transação e o tempo de validade (seção 6.7). Assumimos que a referência ao valor de uma propriedade em uma condição se refere ao valor válido no momento da avaliação da condição.

6.9.2. Linguagem de Lógica Temporal

As condições utilizadas nas regras de transição de estados e de integridade são escritas em uma linguagem de lógica temporal. A lógica temporal é uma especialização da lógica modal - enquanto o domínio de interpretação da lógica modal é um conjunto genérico de estados e as relações entre estes estados, a lógica temporal requer que estes estados formem uma seqüência discreta linear [MAN 81]. Uma seqüência discreta de dois estados pode ser utilizada para descrever as mudanças dinâmicas em instantes discretos.

Através da utilização de lógica temporal podemos representar situações que variam com a passagem do tempo. Assumimos que a variação do tempo é discreta, apresentando um passado linear e permitindo ramificações no futuro. Formalismos temporais têm sido largamente utilizados para expressar requisitos que envolvem tempo e para especificar aplicações dinâmicas. [CAR 88b, CAS 82, FIN 91, GAB 91, LIP 87, SEG 88a]. Uma das vantagens da utilização deste formalismo é a possibilidade de representar informações incompletas e indefinidas através da utilização de operadores temporais tais como *desde* e *até*.

Os símbolos que poderão ser utilizados nas fórmulas das condições temporais são: (i) proposições atômicas, fazendo referência a valores de propriedades estáticas e dinâmicas; (ii) operadores relacionais; (iii) conectivos lógicos *and*, *or* e *not*; (iv) quantificadores existencial *exists* e universal *forall*; (v) valores transmitidos como argumento pelas mensagens recebidas; e (vi) um conjunto de operadores lógicos temporais.

As proposições atômicas envolvem possíveis predicados e funções temporais; estes e os operadores lógicos temporais serão detalhados nas seções seguintes, seguido da formalização da linguagem.

6.9.2.1. Predicados e Funções Temporais a Serem Utilizados nas Condições

As condições definidas nas regras TF-ORM avaliam três tipos de informação: (i) valores de propriedades armazenados no banco de dados; (ii) valores de rótulos temporais associados a estas informações; e (iii) estados de papéis. Foi definido um conjunto de predicados e funções para ser utilizado nestas condições.

Os identificadores de instâncias de classes e de instâncias de papéis, armazenados respectivamente nas propriedades *oId* e *rId*, devem ser utilizados para identificá-las, a menos que esta identificação seja feita pelo contexto - quando a condição se refere a uma propriedade definida no mesmo papel onde a condição foi definida. Quando a propriedade é definida no papel-base da classe, ela é identificada através do identificador da classe. O identificador de papel é utilizado para identificar propriedades definidas dentro de papéis.

A identificação de estados também é feita através dos identificadores de instâncias. Quando for utilizado o identificador da classe, a referência é ao estado do papel-base. O identificador de papel identifica uma referência ao estado do respectivo papel.

Foi definido um conjunto de ***predicados*** a serem utilizados nas condições. Estes predicados são descritos a serguir.

- *has_class_instance*(NomeClasse, Id)
  Verifica se existe pelo menos uma instância da classe denominada *NomeClasse*, sendo o identificador desta instância devolvido na variável *Id*.

- *has_role_instance*(IdClasse, NomePapel, IdPapel)
  has_role_instance(NomePapel, IdPapel)
  O predicado é verdadeiro quando existe pelo menos uma instância do papel de nome *NomePapel* para a instância de classe identificada por *IdClasse*; o identificador deste papel é retornado em *IdPapel*. A segunda forma é utilizada quando a instância da classe é identificada pelo contexto.

- *active_class*(Id)
  active_role(Id)
  Uma instância pode estar ativa ou não. As propriedades *class_instance* e *role-_instance* apresentam esta informação - se seu valor for *null* a instância está inativa. Os predicados acima podem ser utilizados para verificar esta informação. *Id* corresponde ao identificador da instância da classe ou do papel considerado.

- *active_class_at*(Id, Tempo)
  active_role_at(Id, Tempo)
  Verdadeiro quando a classe ou papel identificado por *Id* estava ativa no instante de tempo definido por *Tempo*, anterior ao momento atual.

- *is_valid*(Id, NomePropriedade, Valor)
  is_valid(NomePropriedade, Valor)
  Utilizado para verificar se o valor atualmente válido para a propriedade de nome *NomePropriedade* é o valor definido por *Valor*. Se *Id* corresponde ao identificador de uma classe, a propriedade é definida no papel-base desta classe; quando definida em um outro papel, é utilizado o identificador da instância deste papel. A segunda forma é utilizada quando a instância é identificada pelo contexto.

- *is_valid_at*(Id, NomePropriedade, Valor, Tempo)
  is_valid_at(NomePropriedade, Valor, Tempo)
  O predicado é verdadeiro se o valor fornecido for o valor válido da propriedade no instante de tempo definido por *Tempo*.

- *out_role*(Id, NomePapel)
  Este predicado é verdadeiro quando a instância de classe identificada por *Id* não possui nenhuma instância do papel de nome *NomePapel*.

- *role*(IdClasse, IdPapel)
  Verifica se a instância de papel identificada é uma instância desta classe.

- *before*(Instante, Instante)
  Utilizado para comparar dois instante de tempo - verdadeiro quando o primeiro instante de tempo é anterior ao segundo.

Foi também definido um conjunto de ***funções***, descritas a seguir.

- *value*(Id, NomePropriedade)
  value(NomePropriedade)
  A função retorna o valor atualmente válido da propriedade referenciada, sendo a instância identificada por *Id*. A segunda forma é utilizada quando a instância é identificada pelo contexto.

- *value_at*(Id, NomePropriedade, Tempo)
  value_at(NomePropriedade, Tempo)
  Retorna o valor válido da propriedade no instante de tempo definido por *Tempo*.

- *valid_time*(Id, NomePropriedade)
  valid_time(NomePropriedade)
  Retorna o tempo de validade (instante de tempo em que começa a validade) associado ao último valor definido para a propriedade (último tempo de transação).

- *transaction_time*(Id, NomePropriedade)
  transaction_time(NomePropriedade)
  Retorna o instante de tempo em que foi feita a última transação referente à propriedade definida.

- *class_creation_time*(Id)
  role_creation_time(Id)
  Esta função retorna o instante de tempo de criação da instância da classe ou do papel considerado.

- *class_end_time*(Id)
  role_end_time(Id)
  Retorna o instante de tempo correspondente à destruição da instância da classe ou do papel.

- *state*(Id)
  Retorna o estado atual do papel-base da classe (quando utilizado o identificador de classe) ou do papel (no caso de utilizar identificador de papel).

- *state_at*(Id, Tempo)
  Retorna o estado em que se encontrava o papel base da instância da classe ou o papel da instância do papel, no instante de tempo definido por *Tempo*.

Além destas funções que tratam especificamente de informações referentes a instâncias de classes e de papéis, todas as funções temporais definidas na seção 6.6.5 também podem ser utilizadas.

6.9.2.2. Operadores Lógicos Temporais

Foi definido um conjunto de operadores lógicos temporais para avaliar condições em determinados momentos, listados na tabela 6.2 a seguir. Inicialmente são apresentados seis operadores monádicos, os quais selecionam o(s) momento(s) de tempo em que a expressão sobre a qual estão agindo deve ser avaliada. Quando não é utilizado nenhum destes operadores, as expressões são avaliadas no momento atual. Três diferentes momentos podem ser identificados: (i) considerando o instante imediatamente anterior ou posterior ao momento atual, instante este definido pela menor granularidade temporal da expressão envolvida - geralmente o minuto, a menor granularidade do modelo; (ii) considerando todos os momentos do passado (ou futuro) e verificando se a expressão é válida em *algum* destes momentos; e (iii) considerando também todos os momentos do passado (ou futuro) e verificando se a expressão é válida para *todos* os momentos considerados. Como estamos considerando um ambiente de orientação a objetos, os operadores que referenciam o passado se referem aos momentos a partir da criação da instância considerada; quanto ao futuro, este é limitado pelo tempo de vida da mesma instância.

A tabela 6.2 apresenta ainda quatro operadores diádicos os quais limitam ainda mais os intervalos de tempo sobre os quais as expressões devem ser avaliadas: (i) *since* verifica se o primeiro argumento é válido em todos os momentos desde que o segundo argumento se tornou verdadeiro; (ii) *until* verifica se o primeiro argumento é verdadeiro em todos os momentos desde a criação das instâncias envolvidas até o momento em que o segundo argumento se tornou verdadeiro; (iii) *before* verifica se o primeiro argumento se

tornou verdadeiro pela primeira vez em um momento anterior àquele em que o segundo argumento se tornou verdadeiro; e (iv) *after* verifica se o primeiro argumento é verdadeiro em algum momento posterior àquele em que o segundo argumento é verdadeiro.

Tabela 6.1 : Operações de Soma e Subtração

Operador	Semântica
sometime past A	*A* valeu em algum momento do passado
immediately past A	*A* valeu no momento imediatamente anterior
always past A	*A* valeu em todos os momentos do passado
sometime future A	*A* será válido em algum momento do futuro
immediately future A	*A* será válido no momento imediatamente seguinte
always future A	*A* será válido em todos os momentos do futuro
A since B	*A* valeu em todos os momentos desde que *B* valeu
A until B	*A* vale em todos os momentos até que *B* valha
A before B	*A* valeu em algum momento anterior ao em que *B* valeu
A after B	*A* vale em algum momento depois que *B* valeu

6.9.2.3. A Linguagem de Lógica Temporal das Condições

As condições a serem definidas nas regras de transição de estados e nas regras de integridade devem ser expressas como sentenças de uma linguagem de lógica temporal de primeira ordem (LTP). Considerando o banco de dados que implementa o modelo de dados TF-ORM, caracterizamos como estado deste banco de dados o conjunto de seus conteúdos válidos em um determinado instante. Lógica temporal requer que os diferentes estados do banco de dados formem uma seqüência linear discreta. Estas seqüências são utilizadas para descrever os processos, ocorrendo mudanças neste processos em instantes discretos. Conseqüentemente, lógica temporal é apropriada para raciocínios a respeito de processos dinâmicos e seu comportamento no tempo [MAN 81]. Assumimos que o tempo seja discreto e linearmente ordenado.

Os *símbolos* utilizados nas sentenças de uma LTP são variáveis individuais, constantes, predicados e funções. Os símbolos são considerados como sendo globais, apresentando a mesma interpretação sobre todo o universo e não apresentando variação de significado de um estado do banco de dados para outro.

Os símbolos são particionados em *sortes*, correspondendo cada sorte a um diferente domínio. Alguns símbolos podem apresentar uma assinatura não homogênea, isto é, podem apresentar diferentes sortes associados com diferentes posições dos argumentos.

A LTP considerada consiste de uma linguagem polisortida [END 72], com os seguintes sortes:

nomes de estados
nomes de propriedades
instâncias
valores aritméticos
valores de seqüências de caracteres

valores de instantes de tempo
valores de intervalos de tempo
valores de durações temporais

Os domínios dos nomes de estados e de propriedades consistem dos nomes considerados no modelo de aplicação, acrescentando um conjunto de nomes de estados e de propriedades pré-definidos. Instâncias são representadas por variáveis e por nomes de propriedades específicos. Valores aritméticos e de seqüências de caracteres correspondem a valores normais para estes domínios. Valores de instantes de tempo correspondem a valores de datas (dia, mês e ano) e de horas (hora e minuto). A linguagem apresenta um conjunto de símbolos constantes dos sortes nomes de estados e de propriedades, valores aritméticos, de seqüências de caracteres e de instantes de tempo.

O *alfabeto A* da linguagem contém os símbolos lógicos normais de qualquer linguagem de primeira ordem (operadores aritméticos, operadores lógicos, quantificadores aplicados a variáveis individuais), mais um conjunto de operadores temporais e um conjunto especial de símbolos funcionais e predicativos não-lógicos:

- operadores temporais:

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

- símbolos predicativos:

has_class_instance	de sorte <nome classe> <instância>
has_role_instance	de sorte <instância> <nome papel> <instância>
active_class	de sorte <instância>
active_role	de sorte <instância>
active_class_at	de sorte <instância> <instante de tempo>
active_role_at	de sorte <instância> <instante de tempo>
is_valid	de sorte <instância> <nome de propriedade> <valor>
is_valid_at	de sorte <instância> <nome de propriedade> <valor> <instante de tempo>
out_role	de sorte <instância> <nome de papel>
role	de sorte <instância> <instância de papel>
before	de sorte <instante de tempo> <instante de tempo>
begin	de sorte <nome propriedade>
contains	de sorte <nome de propriedade> <nome de propriedade>

- símbolos funcionais:

value	de sorte <instância> <nome de propriedade>
value_at	de sorte <instância > <nome de propriedade> <instante de tempo>
valid_time	de sorte <instância> <nome de propriedade>
transaction_time	de sorte <instância> <nome de propriedade>
class_creation_time	de sorte <instância>
role_creation_time	de sorte <instância>
class_end_time	de sorte <instância>
role_end_time	de sorte <instância>
state	de sorte <instância>
state_at	de sorte <instância> <instante de tempo>
year	de sorte <instante de tempo>
month	de sorte <instante de tempo>
day	de sorte <instante de tempo>
hour	de sorte <instante de tempo>
minute	de sorte <instante de tempo>
weekday	de sorte <instante de tempo>
lower_bound	de sorte <instervalo temporal>
upper_bound	de sorte <instervalo temporal>
duration	de sorte <duração temporal>

Considerando o alfabeto A, o conjunto de *termos de primeira ordem* sobre este alfabeto satisfaz as seguintes condições [END 72]:

- cada variável em A é um termo no alfabeto;
- toda constante em A é um termo no alfabeto;
- se $t_1 , \ldots , t_n$ são termos em A e f é um símbolo funcional de aridade n em A, então $f(t_1 , \ldots , t_n)$ é um termo em A.

Fórmulas atômicas utilizando este alfabeto são construídas conforme as seguintes condições:

- se $t_1 , \ldots , t_n$ são termos em A e p é um símbolo predicativo de aridade n de A, então $p(t_1 , \ldots , t_n)$ é uma fórmula atômica sobre A;
- se t_1 e t_2 são termos em A e "=" é um símbolo em A, então $(t_1 = t_n)$ é uma fórmula atômica sobre A.

As *fórmulas* (*sentenças*) sobre este alfabeto são formadas da seguinte maneira:

- uma fórmula atômica sobre A é uma fórmula sobre A;
- se f_1 e f_2 são fórmulas sobre A, então a aplicação de conectivos booleanos, operadores lógicos, operadores temporais e quantificadores sobre f_1 e f_2 também são fórmulas sobre A.

A sintaxe completa das regras está detalhada no apêndice 1, na BNF da linguagem TF-ORM.

6.10. Reutilização no Modelo TF-ORM

A especificação de sistemas de informação de escritórios através da utilização do modelo TF-ORM pode ser feita reutilizando informações armazenadas em uma biblioteca de classes.

Para facilitar a recuperação de informações, a biblioteca de classes TF-ORM pode ser estruturada de duas maneiras diferentes:

- armazenando classes completas, associando a cada classe um domínio específico, de modo que a recuperação seja feita pelo nome da classe mas considerando somente um subconjunto de todas as classes armazenadas, definido pelo nome do subdomínio;

- armazenando somente papéis que representem comportamentos-padrão; a classe a ser definida seria montada a partir da reutilização destes papéis.

A separação entre os três tipos de classes - agentes, recursos e processos - contribui para a diminuição do domínio de busca das classes no primeiro caso. Além disso, nas duas formas de estruturação da biblioteca descrições textuais podem ser utilizadas para auxiliar na identificação de: (i) classes, descrevendo o que a classe realmente representa; (ii) papéis, descrevendo o comportamento modelado por cada um dos papéis; e (iii) condições lógicas, esclarecendo em linguagem natural o significado de cada uma delas.

O ambiente de apoio a uma especificação através do modelo TF-ORM, apresentado no capítulo 8 deste documento, utiliza uma biblioteca de classes. As classes são armazenadas de forma completa, cada uma com todos os seus papéis. Metainformações são associadas a cada classe e a cada papel, com a finalidade de auxiliar na identificação das classes.

7. RECUPERAÇÃO DE INFORMAÇÕES

A possibilidade de recuperar informações armazenadas em um banco de dados é fundamental. Cada modelo de dados deve apresentar uma linguagem de recuperação de informações associada, para permitir esta funcionalidade. Consultas temporais são um tipo especial de consultas. O fator tempo pode ser envolvido de três formas diferentes nas consultas: (i) recuperar valores de propriedades cujo domínio é temporal; (ii) se referir a um determiando instante ou intervalo temporal; e (iii) recuperar valores com base em restrições temporais.

Algumas linguagens de consulta para modelos de dados orientados a objetos foram propostas [CAR 88a, CHE 93, DAY 93, DEU 91, KÄF 92, ROS 93, STO 86, SU 93]. A maioria destas linguagens se baseia na linguagem SQL [DAT 87], como por exemplo TOOSQL de Rose e Segev, a linguagem de recuperação de dados do modelo funcional OODAPLEX de Dayal e Wuu, OOTempSQL de Cheng e Gadia e a linguagem MQL do modelo de dados T-MAD de Käfer. As linguagens Quel de Stonebraker [STO 76] e TQuel de Snodgrass [SNO 87] também influenciaram bastante o desenvolvimento de linguagens de consulta de modelos temporais orientados a objetos, como por exemplo a linguagem EXCESS do modelo de dados EXTRA de Carey.

Neste capítulo são abordados alguns conceitos envolvidos com a recuperação de informações. São analisados os diferentes aspectos envolvidos na recuperação de informações quando utilizado um modelo de dados orientado a objetos e qual a influência da dimensão temporal nesta mesma recuperação. Com base nos diferentes parâmetros das consultas e nas diferentes histórias que podem ser consideradas em um banco de dados é proposto um esquema de classificação para consultas.

7.1. Consultas e Tipos de Bancos de Dados

As possíveis consultas temporais dependem não só da especificação da informação buscada mas também do tipo de banco de dados implementado pelo modelo de dados utilizado e da história considerada. Analisemos quais as informações que podem ser recuperadas para cada um dos quatro diferentes tipos de bancos de dados (seção 3.3.10).

Os *bancos de dados instantâneos* não apresentam suporte para informações temporais, não permitindo portanto consultas temporais.

Para os *bancos de dados de tempo de transação*, além da recuperação dos valores atuais das informações armazenadas, podem ainda ser recuperadas informações definidas em algum instante no passado, uma vez que todas as informações passadas estão armazenadas, associadas sempre ao instante de sua definição (tempo de transação).

Os *bancos de dados de tempo de validade* representam mais a semântica da realidade do que os bancos de dados de tempo de transação, que são orientados à implementação. Considerando o tempo de validade associado às informações podem ser

recuperadas informações válidas em momentos presentes, passados e futuros, de acordo com a atual percepção da história dos dados.

Os *bancos de dados bitemporais* combinam as duas representações anteriores, trazendo para cada valor tanto o tempo de transação como o de validade. Isto permite que sejam feitas consultas a respeito de valores atuais, passados e futuros, tanto para o atual estado de validade das informações como para estados passados do banco de dados. As consultas que podem ser feitas nos três primeiros tipos de banco de dados podem todas ser feitas neste último tipo, o qual contém os anteriores.

7.2. Classificação de Consultas

Durante o ano de 1992 e o primeiro semestre de 1993 estabeleceu-se uma discussão através de uma lista de correio eletrônico com o objetivo de criar um conjunto de consultas temporais que pudesse servir de base para qualquer modelo de dados temporal. Esta discussão culminou com a realização de um *workshop* em Arlington, Estados Unidos, em junho de 1993, onde os resultados foram avaliados e reunidos em um documento [JEN 93b]. Uma taxonomia para as consultas foi proposta, baseada na qual as consultas foram elaboradas por diferentes pesquisadores da área. Entretanto, a taxonomia proposta resultou por demais complexa, comprometendo os resultados obtidos, o que pode ser comprovado pelas opiniões discrepantes colocadas na referida lista.

Com base nesta experiência, desenvolvemos neste trabalho um nova taxonomia para consultas temporais, considerando apenas alguns dos fatores considerados na anterior e incluindo as diferentes histórias que podem ser identificadas em um banco de dados temporal. Consideramos somente bancos de dados bitemporais, nos quais tanto o tempo de transação como o tempo de validade são associados a cada informação armazenada.

7.2.1. Seleção e Saída

Uma consulta apresenta dois componentes ortogonais: um componente de seleção e um de saída (projeção). O componente de **seleção** é representado através de uma condição lógica. Considerando consultas temporais, condições podem ser estabelecidas sobre:

- valores de dados; e
- informações temporais associadas a dados (*timestamps*) representando tanto tempo de transação como tempo de validade.

Dependendo do componente de seleção, as consultas podem ser classificadas em: *consultas de seleção sobre dados*, *consultas de seleção temporal* e *consultas de seleção mista*.

Uma consulta é classificada como sendo uma *consulta de seleção sobre dados* quando as condições são estabelecidas somente sobre valores de dados, como por exemplo na consulta "selecione os objetos que são instâncias de uma classe denominada departamento e que se referem ao empregado denominado João". É importante ressaltar

que quando forem utilizados tipos de dados temporais, tais como datas e horas, a utilização destes na condição de seleção representa uma seleção sobre dados e não uma seleção temporal. Um exemplo deste último caso é a consulta "selecione o nome dos empregados que apresentam data de nascimento posterior a 1/jan/1980".

Consultas de seleção temporal são as consultas nas quais somente informações temporais associadas aos dados (tempo de transação e/ou tempo de validade) são analisadas pela condição de seleção, como no exemplo "selecione todos os empregados da empresa durante o período de 1/jan/90 a 1/jan/91".

Nas *consultas de seleção mista* a condição de seleção atua não somente nos dados mas também nas informações temporais associadas a eles, como no exemplo "selecione todos os empregados do departamento denominado entregas que estavam habilitados para dirigir automóveis durante o período de 1/jan/90 a 1/jan/91".

Analisando somente as possíveis **saídas** de uma consulta, diferentes tipos de valores podem ser identificados: um objeto (identificado através de um identificador único), objetos complexos, valores de propriedades de objetos, informações temporais e conjuntos de valores de propriedades e de informações temporais. De acordo com o componente de saída de uma consulta, estas podem ser classificadas em: *consultas de saídas de dados*, *consultas de saídas temporais* e *consultas de saídas mistas*.

Em uma *consulta de saídas de dados* as informações selecionadas correspondem exclusivamente a valores de dados. Um exemplo deste tipo de consulta é o seguinte: "selecione todos os empregados do departamento denominado entregas que estavam habilitados para dirigir automóveis durante o período de 1/jan/90 a 1/jan/91". O resultado desta consulta será um conjunto de empregados.

Consultas de saídas temporais recuperam informações abstraídas das informações temporais associadas aos dados. Deste modo podem ser recuperados pontos no tempo, intervalos temporais e durações temporais. Um exemplo deste tipo de consulta é o seguinte: "selecione todos os períodos nos quais qualquer empregado do departamento de entregas estava habilitado a dirigir automóveis". O resultado desta consulta é um conjunto de intervalos.

As *consultas de saída mista* recuperam simultaneamente valores de dados e valores temporias associados a estes dados, como no exemplo "selecione os valores de salário com os respectivos tempos de validade para o empregado chamado João entre 1/jan/90 e 1/jan/91".

Analisando as possíveis combinações entre os componentes de seleção e de saída de uma consulta observamos que a única combinação que não pode ser utilizada é a de seleção temporal com saída temporal - devemos ter algum dado envolvido em pelo menos um dos componentes. As combinações possíveis são, portanto:

- *seleção sobre dados, saída de dados*
- *seleção sobre dados, saída temporal*
- *seleção sobre dados, saída mista*
- *seleção temporal, saída de dados*
- *selecão temporal, saída mista*
- *seleção mista, saída de dados*
- *seleção mista, saída temporal*
- *seleção mista, saída mista*

7.2.2. Histórias de Bancos de Dados

Considerando bancos de dados bitemporais, a história presente é definida por todos os valores que são válidos no momento presente, que eram válidos em momentos passados e que estão definidos como válidos para momentos futuros. Esta história corresponde ao conhecimento presente a respeito do presente, a respeito do passado e a respeito do futuro. Durante a evolução de um banco de dados diferentes histórias são definidas, de acordo com modificações efetuadas no conhecimento. Tomando como base um momento no passado, existia naquele momento um determinado conhecimento a respeito de valores daquele momento, de momentos anteriores àquele momento e de momentos posteriores a ele. A história conhecida naquele momento não pode levar em consideração eventuais modificações efetuadas nos valores válidos, modificações estas que podem, inclusive, alterar valores passados (ver figura 6.2). Os dados conhecidos naquele momento constituem a correspondente história de valores válidos, identificados através do tempo de transação associado a cada valor - são considerados os valores cujos tempos de transação associados são iguais ou anteriores ao momento considerado.

Cinco diferentes histórias podem ser identificadas em um banco de dados bitemporal, representando diferentes interprestações dos dados armazenados:

- *dados instantâneos atuais*, representados por todas as informações válidas no momento presente;
- *dados instantâneos passados*, representados pelos dados válidos em um determinado instante do passado, de acordo com a atual percepção da história do banco de dados;
- *dados instantâneos de história passada*, considerando todas as informações de um determinado momento no passado, de acordo com a história válida naquele momento;
- *dados históricos*, nos quais estão incluídas todas as informações armazenadas (presentes, passadas e futuras) de acordo com a presente história de dados válidos; e
- *dados históricos de história passada*, análogos aos anteriores porém considerando uma história anterior à atual, definida por um determinado tempo de transação.

A recuperação de valores de uma determinada história do banco de dados depende da condição estabelecida no componente de seleção. Por exemplo, para recuperar informações relativas a dados instantâneos do passado é necessário que o componente de selecão apresente alguma seleção temporal - a seleção do instante passado considerado. Sempre que forem considerados dados históricos podem ser recuperados tempos de

transação, como por exemplo ao recuperar o instante em que foi definido um novo salário para um funcionário.

Na figura 7.1 são apresentados os diferentes conjuntos de dados que podem ser recuperados, ao longo do eixo do tempo. Nas três primeiras linhas podem ser recuperados todos os dados no ponto marcado; nas duas primeiras é considerada a história atual dos dados e na terceira, a história conhecida no ponto marcado com um símbolo retangular. As duas últimas linhas correspondem a dados históricos, podendo ser recuperados todos os dados ao longo da linha de tempo (presentes, passados e futuros), sendo diferente somente a história considerada - o mesmo símbolo retangular assinala em cada uma delas o ponto de tempo referente ao conhecimento da história.

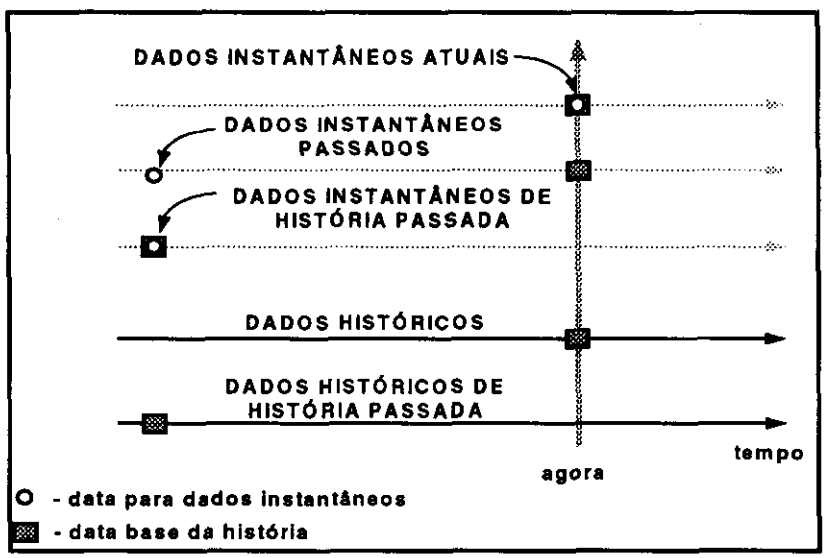

Figura 7.1 : Informações Recuperadas e Histórias de Bancos de Dados

7.2.3. Relacionamento entre Componentes da Consulta e a História do Banco de Dados

Com base nas possíveis combinações entre componentes de uma consulta e as histórias do banco de dados foi feita uma análise das possíveis combinações que podem ocorrer. Os resultados desta análise estão apresentados na tabela 7.1.

Tabela 7.1 : Componentes da Consulta e Recuperação de Dados

	DADOS INSTANTÂNEOS ATUAIS	DADOS INSTANTÂNEOS PASSADOS	DADOS INSTANTÂNEOS DE HISTÓRIA PASSADA	DADOS HISTÓRICOS	DADOS HISTÓRICOS DE HISTÓRIA PASSADA
SELEÇÃO EM DADOS SAÍDA DE DADOS	A1	A2	A3	A4	A5
SELEÇÃO EM DADOS SAÍDA TEMPORAL	B1	B2	B3	B4	B5
SELEÇÃO EM DADOS SAÍDA MISTA	C1	C2	C3	C4	C5
SELEÇÃO TEMPORAL SAÍDA DE DADOS	D1	D2	D3	D4	D5
SELEÇÃO TEMPORAL SAÍDA MISTA	E1	E2	E3	E4	E5
SELEÇÃO MISTA SAÍDA DE DADOS	F1	F2	F3	F4	F5
SELEÇÃO MISTA SAÍDA TEMPORAL	G1	G2	G3	G4	G5
SELEÇÃO MISTA SAÍDA MISTA	H1	H2	H3	H4	H5

Na tabela 7.1, as combinações possíveis entre componentes de uma consulta e os conjuntos de dados que podem ser recuperados são representadas através das células brancas. Assim, a única consulta que permite recuperar dados da história relativa aos *dados instantâneos atuais* está representada na célula A1. Esta, na realidade, não correponde a uma consulta temporal. As célula D2 e F2 apresentam as possíveis combinações de componentes de uma consulta para *dados instantâneos passados* - é necessária uma seleção temporal para fixar o instante de tempo passado considerado, sendo que somente dados podem ser recuperados. D2 pode ser utilizado como um ponto de recuperação de dados, sendo recuperados todos os dados válidos naquele momento no passado, de acordo com a atual concepção da história. Para os *dados instantâneos de história passada* podem ser utilizadas as combinações das células D3 e F3, sendo necessária a seleção temporal não somente para fixar o instante no passado mas também para recuperar a história passada e possibilitando somente recuperação de dados. Todas as combinações dos componentes são possíveis para recuperar informações de *dados históricos*. E para os *dados históricos de história passada* é necessária uma seleção temporal, para fixar o instante no qual deve ser considerada a história passada.

A seguir são apresentados um exemplo de consultas para cada uma das possíveis combinações apresentadas na tabela 7.1:

A1: selecione o atual departamento em que trabalha um funcionário denominado João e que ganha mais do que CR$ 10.000,00;

A4: selecione todos os departamentos (atuais, passados e futuros) de um funcionário denominado João, com salário superior a CR$10.000,00;

B4: selecione todos os períodos em que o funcionário denominado João esteve qualificado para dirigir automóveis;

C4: selecione todos os salários do funcionário denominado João, juntamente com suas correspondentes datas de validade;

D2: selecione os nomes de todos os funcionários em 1/jan/90;

D3: selecione o nome de todos os funcionários em 1/jan/90, de acordo com o que se acreditava como verdadeiro em 1/jan90;

D4: selecione o nome de todos os funcionários durante o período de 1/jan/90 a 1/jan/91;

D5: selecione o nome de todos os funcionários no período de 1/jan/90 a 1/jan/91, de acordo com o que se acreditava em 30/mar/92;

E4: selecione os salários de todos os funcionários e suas correspondentes datas de validade, durante o período de 1/jan/90 e 1/jan/91;

E5: selecione ps sala'rios de todos os funcionários e suas correspondentes datas de validade, durante o período de 1/jan/90 a 1/jan/91, de acordo com o que se acreditava em 1/mar/92;

F2: selecione o departamento em que trabalhava o funcionário denominado João em 1/dez/91;

F3: selecione o departamento em que trabalhava o funcionário denominado João em 1/dez/91, de acordo com o que se acreditava em 1/dez/91;

F4: selecione todos os departamentos nos quais o funcionário denominado João trabalhou durante o período de 1/jan/90 e 1/jan/91;

F5: selecione todos os departamentos nos quais o funcionário denominado João trabalhou durante o período de 1/jan/90 e 1/jan/91, de acordo com o que se acreditava em 1/mar/92;

G4: selecione todos os períodos em que o funcionário denominado João esteve qualificado para dirigir automóveis, durante o período de 1/jan/90 a 1/jan/91;

G5: selecione todos os períodos nos quais o funcionário denominado João esteve qualificado para dirigir automóveis, durante o período de 1/jan/90 a 1/jan/91, de acordo com o que se acreditava em 1/mar/92;

H4: selecione todos os salários e seus correspondentes tempos de validade para o empregado denominado João, de 1/jan90 a 1/jan/91;

H5: selecione todos os salários e seus correspondentes tempos de validade para o empregado denominado João, de 1/jan90 a 1/jan/91, de acordo com o que se acreditava em 1/mar/92.

7.3. Consultas e o Paradigma de Orientação a Objetos

Modelos de dados orientados a objetos requerem propriedades especiais para a recuperação de informações. Os objetos apresentam atributos (propriedades) cujos valores são definidos em domínios específicos. Os domínios podem ser simples (como, por exemplo, *inteiros* e *reais*) ou complexos, representados por nomes de classes. Este último caso caracteriza classes denominadas complexas. Os valores assumidos por estas

propriedades são instâncias das classes especificadas como domínios. Além disso, classes complexas podem ser construídas através de construtores (conjunto, lista). A recuperação de valores de propriedades cujos domínios são classes deve ser considerada quando elaborada uma linguagem de consulta para um modelo orientado a objetos. Várias soluções podem ser adotadas para a recuperação destas propriedades, como por exemplo: (i) devolver o identificador do objeto recuperado, embora este identificador seja usualmente interno ao sistema e, portanto, não acessível ao usuário; (ii) listar os valores de todas as propriedades do objeto identificado, identificando os objetos referentes às propriedades recursivamente até que todas as propriedades sejam definidas em domínios simples; (iii) listar somente os valores referentes a propriedades simples do objeto identificado; ou (iv) fornecer o(s) valor(es) de alguma(s) propriedade(s) identificada(s) pelo modelo como especial para esta finalidade.

As propriedades cujo domínio são classes podem apresentar diversos objetos (instâncias desta classe) definidos. A recuperação de informações para estes casos requer que todos os objetos sejam devolvidos pela linguagem de consulta.

Outro problema envolvido na recuperação de informações em um modelo de dados orientado a objetos consiste na possível hierarquia de classes existente. Uma classe pode apresentar um conjunto de subclasses e um conjunto de superclasses. Quando o domínio de uma propriedade for uma classe, todas as subclasses desta (diretas ou indiretas) são também possíveis domínios desta propriedade. A recuperação de informações deve navegar através de toda a hierarquia para fornecer todos os valores definidos.

Analisemos agora a influência do paradigma de orientação a objetos em consultas temporais. Para tal vamos nos basear no exemplo de esquema orientado a objetos representado na figura 7.2. São representadas três classes - *pessoa*, *departamento* e *funcionário*. A classe *pessoa* apresenta três propriedades: *nome*, *data_de_nascimento* e *sexo*. A classe *funcionário* é uma subclasse da classe *pessoa* e apresenta também três propriedades: *salário*, *locação* e *habilidades*. O domínio da propriedade *locação* é a classe *departamento*. Esta, por sua vez, apresenta duas propriedades: *nome* e *chefe*. O domínio da propriedade *chefe* é a classe *funcionário*.

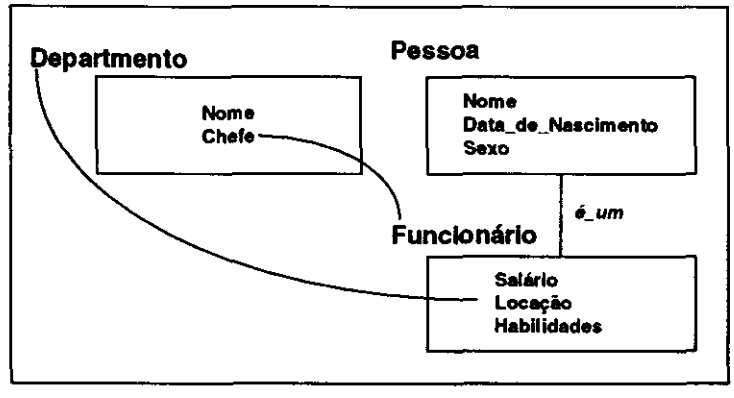

Figura 7.2 : Exemplo de Esquema Orientado a Objetos

Uma possível consulta não temporal neste esquema é a seguinte: "selecione todos os nomes de funcionários do departamento denominado contabilidade, que ganham mais de CR$100.000,00". Vimos que as possíveis consultas temporais que podem ser elaboradas dependem do tipo de banco de dados temporal implementado pelo modelo de dados utilizado. Quando for utilizado um banco de dados bitemporal, a consulta acima recupera os valores atualmente válidos para a propriedade citada.

Os valores temporais associados às informações (tempo de transação e tempo de validade) também podem ser recuperados através da linguagem de consulta. Conforme o modelo utilizado, estes tempos podem estar associados a cada propriedade ou a cada objeto, devendo a linguagem de consulta identificá-los de forma a deixar sua localização transparente ao usuário.

Duas formas de valores passados podem ser referenciados em uma consulta: (i) valores passados de acordo com a atual história do banco de dados; e (ii) valores que se acreditava como válidos em um momento passado. A linguagem de consulta deve permitir identificações diferentes para instantes no passado (ou futuro) dos quais queremos recuperar alguma informação, e o instante de tempo que marca uma história passada.

8. LINGUAGEM DE CONSULTA DO MODELO TF-ORM

A linguagem de consulta do modelo TF-ORM [EDE 94] se baseia na linguagem SQL [DAT 87], apresentando também alguma influência de TQuel [SNO 87]. A forma geral de uma consulta SQL é a seguinte:

SELECT <cláusula de especificação>
FROM <cláusula de identificação>
WHERE <cláusula de busca>

A cláusula de especificação define a estrutura do resultado solicitado; a cláusula de identificação identifica os objetos sobre os quais vai ser efetuada a busca; e a cláusula de busca apresenta em condição que deve ser satisfeita pelos objetos buscados.

A maior parte dos modelos de dados temporais utiliza esta linguagem como base, estendendo-a para captar os aspectos temporais. A estrutura geral do comando SQL - SELECT, FROM, WHERE - geralmente é mantida nestas linguagens, sendo acrescentadas condições temporais à cláusula de busca [DAY 93, GAB 91, KÄF 92, SAR 90]. Em [NAV 88], Navathe define uma linguagem SQL temporal - TSQL - na qual é acrescentada uma nova cláusula, a cláusula WHEN, para a especificação de condições que avaliam predicados temporais. Rose e Segev [ROS 93] também utilizam esta cláusula na sua linguagem de consulta TOOSQL; Cheng e Gadia [CHE 93] utilizam uma cláusula WHILE em lugar da cláusula WHEN.

Na linguagem de consulta TF-ORM ambas as cláusulas (WHERE e WHEN) são utilizadas, de uma forma independente, como acontece na linguagem TQuel [SNO 87]. A escolha de uma das cláusulas define o universo de busca. Quando for utilizada a cláusula WHERE a informação é buscada no banco de dados instantâneo (*snapshot*) correspondente ao instante de tempo considerado; pode se referir ao estado atual do banco de dados ou a um estado de uma história passada. A utilização da cláusula WHEN aumenta o universo de busca para todas as informações da história considerada (atual ou passada), incluindo dados passados e futuros além dos atuais.

As duas estruturas seguintes definem a linguagem de consulta TF-ORM:

SELECT <cláusula de especificação>
FROM <cláusula de identificação>
WHERE <cláusula de busca>
[**AS ON** <cláusula de instante temporal>]

e

SELECT <cláusula de especificação>
FROM <cláusula de identificação>
WHEN <cláusula de busca>
[**AS ON** <cláusula de instante temporal>]

A *cláusula de especificação* define o componente de saída da consulta. O componente de seleção é definido pela *cláusula de identificação*, a qual define o domínio de valores a serem analisados, e pela *cláusula de busca*, onde são especificadas as condições que devem ser satisfeitas pelos objetos recuperados. A *cláusula de instante temporal* define a data correspondente a uma história passada, mudando o referencial de tempo de execução da consulta para o momento definido e utilizando como base as informações conhecidas naquele momento.

A diferença básica entre as duas estruturas é a possibilidade de escolha entre as cláusulas WHERE e WHEN. Quando utilizadas as cláusulas SELECT, FROM e WHERE são recuperadas as informações atualmente válidas - informações válidas atualmente para a história atual do banco de dados. A utilização das cláusulas SELECT, FROM e WHEN recupera todas as informações definidas na atual história dos dados (informações presentes, passadas e futuras). O acréscimo da cláusula AS ON fixa uma história anterior à história atual, da qual os valores devem ser recuperados.

No Anexo 2 deste documento a sintaxe da linguagem de consulta TF-ORM é apresentada através de uma notação BNF. A seguir cada uma das cláusulas acima citadas será apresentada com mais detalhes.

8.1. Cláusula de Especificação

A cláusula de especificação define as saídas da consulta. Os três tipos de saídas podem ser obtidos (seção 7.2.1): saída de dados, temporal e mista.

Saída de dados são obtidas quando são especificadas as partes do objeto que devem ser mostradas pela consulta. Para obter este tipo de saída a cláusula de especificação pode ser composta: (i) pelo nome de uma propriedade; (ii) por uma lista com nomes de propriedades separadas por vírgulas; ou (iii) pelo símbolo especial "*", quando forem solicitados os valores de todas as propriedades do(s) objeto(s) identificado(s). Para as propriedades cujo domínio é o nome de uma classe são listadas todas as propriedades do objeto identificado nesta classe, associado ao nome da classe. Isto é repetido neste objeto que corresponde à propriedade, para todas as suas propriedades cujos domínios são classes, até que tenhamos somente propriedades definidas em domínios simples. Quando uma propriedade é definida como um conjunto ou uma lista de elementos, todos os possíveis elementos deste conjunto ou lista são recuperados.

Uma *saída temporal* é obtida quando a cláusula de especificação solicita uma data de transação, uma data de validade ou um período. As datas de transação e de validade estão associadas a cada uma das propriedades dinâmicas do modelo. As propriedades estáticas não possuem datas associadas, sendo válidas enquanto a instância estiver ativa. As propriedades especiais *class_instance* e *role_instance* associadas às instâncias trazem a informação dos períodos de validade da instância, sendo utilizados pela linguagem de consulta para deduzir os períodos de validade das propriedades estáticas. Datas de transação para propriedades estáticas não têm significado. As seguintes palavras especiais são utilizadas na cláusula de especificação para solicitar as saídas temporais:

- DATE, quando solicitada uma data de validade;
- TRANSACTION_DATE, quando solicitada uma data de transação;
- PERIOD, quando solicitado um período (intervalo) limitado por duas datas de validade. São considerados somente intervalos fechados - os limites pertencem ao intervalo; e
- TRANSACTION_PERIOD, quando solicitado um período limitado por duas datas de transação.

Para obter uma *saída mista* deve ser utilizada a palavra especial *and* unindo um dos elementos que definem a saída de dados com um dos elementos que definem a saída temporal (nesta ordem).

8.2. Cláusula de Identificação

Na cláusula de identificação são identificados uma classe e um papel desta classe. A qualificação do papel através do nome da classe a que corresponde não se faz necessária quando o esquema sobre o qual está sendo definida a consulta apresentar todos os papéis com nomes únicos (o que não é necessário no modelo - podemos ter classes diferentes com papéis de mesmo nome).

8.3. Cláusula de Busca

Uma classe pode apresentar diversas instâncias (objetos). Cada um destes objetos pode apresentar diversas instâncias do mesmo papel. A identificação das instâncias que devem ser consideradas na consulta é feita na cláusula de busca. Nesta cláusula são também especificadas as restrições de ordem temporal que se aplicam à busca efetuada, utilizadas para identificar datas e períodos de informações passadas.

A cláusula de busca é expressa através de uma fórmula lógica escrita na mesma linguagem de lógica temporal utilizada para expressar as condições no modelo TF-ORM (seção 6.9.1.), acrescida de dois predicados específicos para seleções temporais em consultas: (i) "PERIOD in [<data>, <data>]", o qual define o período (intervalo) no qual os dados históricos devem ser analisados; e (ii) "DATE = <data>", definindo uma data de transação a ser utilizada na seleção.

8.4. Cláusula de Instante Temporal

Na cláusula de instante temporal é fixado um determinado tempo de transação para servir de base para uma história passada do banco de dados. Transações ocorridas depois desta data não são consideradas na consulta. Esta data pode ser definida explicitamente ou através de algum predicado temporal que devolva uma data.

8.5. Exemplos de Consultas na Linguagem de Consulta TF-ORM

Nesta seção são apresentados alguns exemplos da utilização da linguagem de recuperação de informações TF-ORM. Para estes exemplos vamos supor que o esquema conceitual apresente uma classe denominada *Pessoa* e que dois dos papéis desta classe sejam *Funcionário* e *Gerente*. O papel básico de *Pessoa* apresenta a propriedade dinâmica *nome*. *Departamento*, *habilidades* e *salário* são propriedades dinâmicas de *Funcionário*.

Consideremos, inicialmente, algumas da consultas apresentadas na seção 7.2.3. Correspondendo à célula A1 temos a seguinte consulta: "selecione o atual departamento em que trabalha um funcionário denominado João e que ganha mais do que CR$ 10.000,00". Na linguagem TF-ORM esta consulta é expressa da seguinte maneira:

```
SELECT departamento
FROM Pessoa.Funcionário
WHERE nome = "João" and salário > 10000
```

Como o esquema pode apresentar outras classes com algum papel denominado *Funcionário*, é necessário especificar o nome da classe (*Pessoa*) para identificar o papel considerado. A consulta vai considerar em sua busca todas as instâncias da classe *Pessoa* e, para cada uma delas, todas as instâncias do papel *Funcionário*. O resultado será o nome do departamento em que o funcionário denominado João trabalha atualmente - considerando o departamento válido no momento atual. A consulta somente apresentará mais de um resultado se o banco de dados apresentar mais de um funcionário que atenda às condições impostas.

Vejamos o que muda se utilizarmos o WHEN em lugar do WHERE:

```
SELECT departamento
FROM Pessoa.Funcionário
WHEN ( nome = "João" or sometime past nome = "João")
        and salário > 100000
```

Esta consulta corresponde àquela que exemplificou a célula A4 na seção 7.2.3: "selecione todos os departamentos (atuais, passados e futuros) de um funcionário denominado João, com salário superior a CR$10.000,00". A utilização da cláusula WHEN faz com que sejam analisados todos os dados definidos no banco de dados - presentes, passados e futuro. Serão recuperados todos os departamentos nos quais o funcionário João trabalhou, trabalha e trabalhará com o salário estipulado.

A evolução do banco de dados deve ser considerada quando da elaboração de uma consulta, ao lembrar que as propriedades dinâmicas podem mudar de valor com a passagem do tempo. Assim, a condição do nome ser João deve considerar a possibilidade deste nome ter sido outro no passado, ou de algum funcionário que se chamava João ter alterado seu nome atualmente. O enunciado correto da consulta seria: "selecione todos os departamentos de um funcionário que atualmente se chama João ou que em algum momento do passado se chamava João, com salário superior a CR$10.000,00".

Como exemplo de uma consulta que apresenta seleção de dados e saída temporal sobre dados históricos, consideremos o exemplo da célula B4 da seção 7.2.3: "selecione todos os períodos em que o funcionário denominado João esteve qualificado para dirigir automóveis":

```
SELECT PERIOD
FROM Pessoa.Funcionário
WHEN (nome =  "João" or sometime past nome = "João")
        and habilidades contains "dirigir_automóveis"
```

A consulta seguinte também recupera dados históricos, utilizando agora seleção mista e apresentando saída temporal (G4 - "selecione todos os períodos em que um funcionário denominado João esteve qualificado para dirigir automóveis, durante o período de 1/jan/90 a 1/jan/91"):

```
SELECT PERIOD
FROM Person.Employee
WHEN (nome = "João" or sometime past nome = "João")
        and habilidades contains "dirigir_automóveis"
        and PERIOD in [01/01/90, 01/01/91]
```

Para selecionar informações de dados instantâneos passados devemos definir a data a ser considerada, como no exemplo correspondente à célula F2: "selecione o departamento em que trabalhava um funcionário denominado João, em 1/dez/91".

```
SELECT departamento
FROM Pessoa.Funcionário
WHERE (nome = "João" or sometime past nome = "João")
        and DATE = 12/01/91
```

A cláusula AS ON é utilizada para recuperar informações de histórias passadas. Como um exemplo consideremos a consulta correspondente à célula G5: "selecione todos os períodos nos quais o funcionário denominado João esteve qualificado para dirigir automóveis, durante o período de 1/jan/90 a 1/jan/91, de acordo com o que se acreditava em 1/mar/92". Na linguagem TF-ORM esta consulta é expressa por:

```
SELECT PERIOD
FROM Pessoa.Funcionário
WHEN (nome = "João" or sometime past nome = "João")
        and habilidades contains "dirigir_automáveis"
        and PERIOD in [01/01/90, 01/01/91]
AS ON 03/01/92
```

A identificação de instâncias de uma classe ou de um papel pode ser necessária em uma consulta. Como exemplo consideremos o caso em que se quer a recuperação do nome de todos os funcionários:

```
SELECT nome
FROM Pessoa
WHEN has_class_instance(pessoa, Oid) and
        has_role_instance(Oid, funcionário) and
        has_role_instance(Oid, gerente)
```

Nesta consulta todas as instâncias da classe *Pessoa* são analisadas; para cada uma delas é verificado se existe simultaneamente uma instância do papel *Funcionário* e uma do papel *Gerente*.

9. AMBIENTE DE APOIO PARA UMA ESPECIFICAÇÃO ATRAVÉS DO MODELO TF-ORM

Neste capítulo é apresentado um ambiente que apoia especificações através do modelo de dados TF-ORM, enfatizando a reutilização de classes armazenadas em uma biblioteca. O ambiente foi implementado em computador PC-compatível, na linguagem PROLOG [Sterling 86]. Uma ferramenta para apoio a especificações em F-ORM, também fazendo uso de uma biblioteca de classes, denominada RECAST [BEL 91], está sendo implementada no âmbito do projeto ITHACA [PRO 89].

Dois diferentes *usuários* são identificados no ambiente: o engenheiro de aplicações e o projetista de aplicações. O *engenheiro de aplicações* é um especialista encarregado da construção da biblioteca de classes. É também encarregado de posteriormente, a intervalos periódicos, fazer a manutenção da biblioteca com base nas informações referentes às especificações realizadas. Estas informações são armazenadas pelo ambiente em uma biblioteca auxiliar durante os processos de especificação. O *projetista de aplicações* é o usuário que utiliza o ambiente como apoio para a construção da especificação de uma aplicação. O próprio ambiente, através de uma ferramenta específica, guia esta especificação enfatizando a utilização de classes da biblioteca.

9.1. Bancos de Dados

Três diferentes bancos de dados são utilizados pelo sistema que implementa este ambiente: o banco de dados especificação, a biblioteca de classes e a biblioteca de manutenção.

No *banco de dados especificação* é armazenada uma especificação enquanto está sendo construída. Somente uma aplicação pode ser especificada de cada vez. Ao final do processo a especificação construída é armazenada em um arquivo externo e o banco de dados especificação é liberado. Cada especificação realizada é associada a um domínio de aplicação. As classes armazenadas na biblioteca de classes também estão associadas a domínios de aplicação específicos. A pesquisa na biblioteca a partir de domínios de aplicação mais restritos diminui o tempo de busca para reutilização de classes.

A *biblioteca de classes* armazena as classes a serem reutilizadas durante as especificações. A biblioteca é inicialmente povoada com um conjunto de classes abstraídas do domínio de aplicação considerado, um subdomínio da área de sistemas de informação de escritórios. São escolhidas aquelas classes que apresentam possibilidade de serem utilizadas em mais de uma aplicação deste domínio. Posteriormente esta biblioteca sofre processos de manutenção periódicos, com base nos processos de reutilização realizados durante as especificações. Nestas manutenções classes novas podem ser incluídas na biblioteca, classes que não mostraram utilidade para reutilização podem ser retiradas e classes já existentes podem ser modificadas.

Na *biblioteca de manutenção* são armazenadas as informações que podem ser úteis nos processo de manutenção. São armazenadas todas as informações referentes às reutilizações de classes efetuadas - nomes das classes reutilizadas, se foram reutilizadas

com ou sem modificações e as alterações efetuadas. Além disso, uma cópia das novas classes definidas também é armazenada.

9.2. Ferramentas de Apoio

O ambiente apresenta duas ferramentas independentes, uma para cada tipo de usuário previsto. Somente uma das ferramentas pode ser executada de cada vez. O objetivo das ferramentas é apoiar as tarefas dos usuários, apresentando uma interface interativa coloquial. Na figura 9.1 estão representados os bancos de dados utilizados pelo ambiente, as ferramentas disponíveis para manipular as informações referentes a estes bancos de dados e os fluxos de informações efetuados pelas ferramentas entre os bancos de dados.

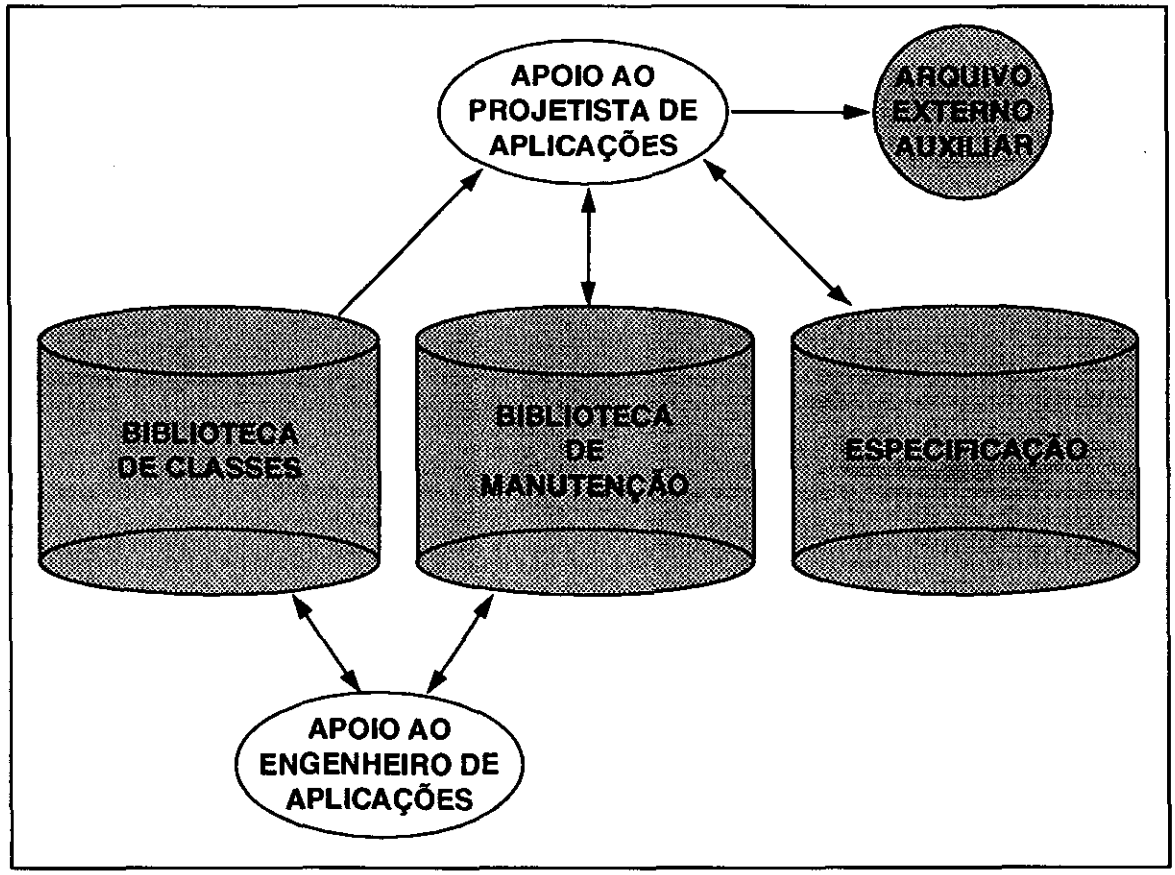

Figura 9.1 : Bancos de Dados e Ferramentas do Ambiente

9.2.1. Ferramenta de Apoio ao Engenheiro de Aplicações

Esta ferramenta apresenta duas funções independentes: (i) contrução inicial de uma biblioteca de classes, e (ii) manutenção de uma biblioteca já existente, com base nas informações contidas na biblioteca de manutenção.

A construção de uma biblioteca é efetuada pela definição de um conjunto de classes e de suas metainformações. A ferramenta não apresenta apoio à análise do domínio de aplicação, somente armazena as classes definidas pelo engenheiro de aplicações. As informações referentes a estatísticas de posteriores utilizações das classes são inicializadas pela ferramenta.

A manutenção da biblioteca é feita apresentando ao engenheiro de aplicações as informações da biblioteca de manutenção e da biblioteca de classes, para sua análise. As informações referentes a reutilizações efetuadas durante especificações construídas são armazenadas na biblioteca de manutenção pela ferramenta de apoio ao projetista de aplicações, podendo ser visualisadas através de opções fornecidas pela ferramenta. Esta oferece ainda diversas opções de listagem de nomes de classes já definidas na biblioteca de classes, sinônimos destes nomes, apresentação de toda uma classe definida. Além de apresentar as informações destas duas bibliotecas, a ferramenta apoia a manutenção da biblioteca de classes e a atualização da biblioteca de manutenção após o processo de manutenção, retirando informações não mais necessárias e atualizando indicadores de estatísticas de reutilização. A manutenção da bilioteca de classes pode significar:

- acréscimo de novas classes;
- remoção de classes que não foram reutilizadas;
- alterações em classes já armazenadas.

O nome de cada classe da biblioteca deve ser único. Para facilitar a identificação de uma classe com vistas à sua posterior reutilização são definidas metainformações associadas a cada classe: sinônimos do nome da classe, além de uma descrição textual de sua aplicação. Tanto os sinônimos como a descrição textual são fornecidos pelo projetista de aplicações. O mesmo acontece com os nomes de papéis. Além disso, descrições textuais explicando as condições definidas em regras são também fornecidas pelo projetista.

9.2.2. Ferramenta de Apoio ao Projetista de Aplicações

A ferramenta de apoio ao projetista de aplicações tem por objetivo ser utilizada durante a especificação de uma aplicação. O projetista de aplicações é guiado na definição de classes de objetos segundo o modelo TF-ORM. A definição de classes é atemporal, isto é, o esquema é estático, sendo entretanto representados os aspectos temporais da aplicação. A definição de classes é feita apoiada na reutilização de classes anteriormente definidas, contidas na biblioteca de classes. Inicialmente é identificado o domínio de aplicação do sistema a ser especificado. Poderá ser utilizado um dos domínos já existentes na biblioteca de classes ou definido um novo domínio.

Após a identificação do domínio, as classes da biblioteca associadas a este domínio podem ser reutilizadas. A ferramenta permite reutilizar classes também de outros domínios, quando assim desejado. Várias opções de pesquisa das classes disponíveis na biblioteca são oferecidas - todas as classes da biblioteca, classes de um determinado domínio de aplicação, classes de agentes, classes de recursos, classes de processos, sinônimos de um nome de classe, descrição textual de uma determinada classe, etc.

Uma classe pode ser definida como:

- nova classe, independente das classes da biblioteca;
- cópia de uma classe encontrada na biblioteca de classes;
- modificação de uma classe da biblioteca.

Quando não for encontrada nenhuma classe na biblioteca, adequada à reutilização na aplicação em desenvolvimento, o projetista de aplicações opta por definir uma nova classe. A ferramenta de apoio guia esta definição, de acordo com o modelo utilizado, sendo a classe armazenada na base de dados da especificação e uma cópia feita para a biblioteca de manutenção.

O nome de uma classe deve ser único em uma especificação. Nada impede que uma nova classe seja criada com nome igual a outra contida na biblioteca de classes, sem se basear nela. Cabe ao engenheiro de aplicações, posteriormente, analisar a possibilidade de introduzir esta nova classe na biblioteca, devendo então seu nome ser alterado.

Quando uma classe for identificada na biblioteca de classes, duas formas de reutilização podem ser empregadas: com ou sem modificações. No caso de ser utilizada uma classe sem modificações, as informações correspondentes são copiadas para a base de dados. As informações referentes a estatísticas e as metainformações não são copiadas para a base de dados, embora o projetista de aplicações tenha acesso às mesmas para informação. Em seguida é incluída na biblioteca de manutenção uma indicação da reutilização e das modificações introduzidas.

Quando uma classe é definida com base em uma classe encontrada na biblioteca, necessitando de alterações, a ferramenta faz uma cópia, com controle do especificador, das informações contidas na biblioteca para a base de dados especificação. Uma vez copiada a classe, a mesma pode ser modificada conforme as necessidades. Podem ser acrescentados novos atributos, novos papéis, modificados ou suprimidas características da classe.

Uma especificação em TF-ORM pode ser realizada em vários níveis de detalhamento [DEA 91c]. O nível de detalhamento considerado para uma determinada informação é armazenado junto a esta informação na base de dados especificação. O nível inicial de detalhamento de uma classe é sempre considerado como sendo de número 1, recebendo os detalhamentos posteriores desta classe números incrementados de uma unidade cada um. O maior nível de detalhamento associado à informações de uma classe representa a especificação final desta classe. São estas as informações referentes à classe copiadas para a biblioteca de manutenção.

Diversos testes de consistência são realizados por esta ferramenta, em três momentos diferentes: (1) durante a especificação de uma classe, (2) logo ao final da especificação de uma classe, e (3) ao final da especificação da aplicação. Entre os testes de consistência realizados durante a especificação de uma classe podem ser citados

- nomes únicos para classes de objetos, para nomes de papéis dentro de uma determinada classe de objetos, e para nomes de estados e de mensagens dentro de um determinado papel de uma classe de objetos;
- referência a propriedades e mensagens dentro de expressões lógicas, pertencentes às propriedades e às mensagens definidas para o papel considerado, para o papel básico da classe considerada ou para o papel básico de uma superclasse da classe considerada;
- consistência nas mensagens constantes das regras de transição compatível com sua localização na regra;
- consistência de número de argumentos em mensagens de expressões lógicas.

Logo após o término da especificação de uma classe são realizados os seguintes testes:

- se em cada conjunto de regras de transição dos diferentes papéis aparece o estado inicial, alcançado pelo recebimento da mensagem de criação de uma instância do papel considerado;
- se cada estado definido aparece pelo menos em alguma regra de transição de estados;
- se algum dos estados é inatingível a partir da regra inicial;
- se todas as mensagens aparecem nas regras de transição de estados.

Ao final de uma especificação a ferramenta realiza testes de consistência complementares, verificando a especificação como um todo. Neste momento são verificados:

- se todas as classes definidas como superclasses e como classes componentes de novas classes estão definidas;
- se todas as classes de objetos utilizadas nos domínios de propriedades estão definidas;
- se todos os papéis de classes utilizados na definição de mensagens estão definidos;
- se as mensagens enviadas por um determinado papel para outro aparece listada no conjunto de mensagens recebidas por este segundo papel e se apresenta os mesmos argumentos.

Quando o projetista dá por encerrada uma especificação, após a realização dos testes de consistência finais, a ferramenta faz uma cópia da versão final das classes para a biblioteca de manutenção para posterior análise do engenheiro de aplicações. São copiadas as novas classes definidas e as classes reutilizadas porém modificadas. Indicações sobre reutilização de classes são também armazenadas - nomes das classes reutilizadas sem modificações, nomes anteriores de classes reutilizadas caso o nome tenha sido alterado e nomes das classes reutilizadas com modificações. Neste último caso, são também

armazenadas algumas informações fornecidas pelo projetista de aplicações, referentes às modificações efetuadas, para posterior análise por parte do engenheiro de aplicações e resultando eventualmente na modificação da classe constante da biblioteca de classes. Algumas metainformações relativas às classes são também incluídas na biblioteca de manutenção, tais como sinônimos dos nomes das classes e dos papéis, explicações textuais sobre classes e papéis, e explicações textuais das condições lógicas.

9.3. Modelos de Dados Internos

Durante um processo de especificação os três bancos de dados são utilizados simultaneamente. Como o ambiente é implementado em PROLOG, um só banco de dados físico é utilizado. Os modelos internos para representar as informações referentes a cada um dos bancos de dados é por isso diferente, de modo a permitir a individualização das informações referentes a cada um deles [EDE 93c]. A seguir apresentamos os modelos internos correspondentes a cada um dos bancos de dados utilizados.

9.3.1. Modelo Interno do Banco de Dados Especificação

A biblioteca de classes associa cada uma das classes definidas a um determinado domínio de aplicação. Isto permite uma mais efetiva recuperação de informações a serem reutilizadas durante um processo de especificação. Por sua vez, uma especificação em andamento também é associada a um domínio de aplicação. Deste modo é possível acrescentar à biblioteca de classes aquelas definidas em uma determinada aplicação, o que é feito durante o processo de manutenção. A seguir apresentaremos os predicados utilizados internamente para armazenar a especificação; utilizaremos como convenção de representação as palavras reservadas iniciando por letras minúsculas e as variáveis, cujos valores são fornecidos pelo projetista da aplicação, iniciando por letras maiúsculas.

Cada especificação é associada a um só domínio de aplicação. A definição do domínio de aplicação no banco de dados especificação é feito através do seguinte predicado:

aplication (Nome_Aplicação).

Considerando a metodologia de aplicação do modelo F-ORM [DEA 91c] que também pode ser aplicada ao modelo TF-ORM, uma especificação pode ser construída em diversos níveis de detalhamento, processo este que pode ser *top-down* ou *bottom-up*. Para o modelo TF-ORM consideramos somente o desenvolvimento *top-down*, detalhando a definição das classes em refinamentos sucessivos. Todos os diversos níveis de detalhamento de especificação de uma classe ficam armazenados no banco de dados especificação durante o processo, podendo ser consultado. Para isto foi criada uma indicação do nível que está sendo considerado. É adotado o maior número de ordem para o nível mais detalhado da especificação, o qual constitui a especificação final. O nível inicial recebe o número 1 pelo sistema. A definição do nível em que uma classe se encontra é associada à definição de cada uma das informações relativas a uma classe. A reutilização de classes da biblioteca em uma especificação se dá somente no nível inicial de definição de uma classe.

Conforme o tipo de classe definido (classe de agente, de recurso ou de processo), o seu nome é definido através de um dos seguintes predicados:

agclass(Nome_Classe).
resclass(Nome_Classe).
proclass(Nome_Classe).

O nome de cada classe deverá ser único na especificação. Os diversos papéis definidos para uma classe são associados ao nome da classe através de predicados da seguinte forma:

role(Classe, Nível, Nome_Papel).

O nome de cada papel deve ser único para uma mesma classe de objetos. O papel básico recebe internamente o nome de *baserole*.

Na definição das propriedades é feita a associação do nome da propriedade ao domínio dos valores que a mesma poderá receber. O nome de uma propriedade deve ser único dentro de um papel da classe. Como domínios podem ser utilizados tipos de dados, nomes de outras classes de objetos, além de conjuntos e listas de objetos. Os tipos de dados permitidos para domínios são pré-definidos no sistema: **integer**, **real**, **boolean**, **string**, **text**, **instant**, **date**, **year**, **month**, **day**, **time**, **hour**, **minute**, **week**, **semester**, **weekday**, **interval**, **span**.

Nos casos particulares de definição utilizando os tipos *interval* e *span*, informações complementares deverão ser fornecidas. No caso de *intervalos*, devem ser definidos o tipo de seus limites e o tipo do intervalo. O tipo dos limites determina a unidade de variação temporal dentro do intervalo, podendo ser utilizados **integer**, **instant**, **date**, **year**, **month**, **day**, **time**, **hour** e **minute**. A pertinência dos pontos limites ao intervalo determina o tipo de intervalo - **closed** quando nenhum dos limites pertence ao intervalo, **upopen** quando somente o limite superior (na seqüencia temporal) não pertence ao intervalo, **downopen** quando o limite inferior não pertence ao intervalo, **open** quando nenhum dos limites pertence ao intervalo, **upfloat** quando o limite superior é o momento atual e **downfloat** quando o momento atual é o limite inferior. No caso de definição através do tipo duração (*span*), deverá ser definida a sua unidade de medida temporal, podendo ser utilizadas somente unidades simples - **year**, **month**, **day**, **hour**, **minute**, **week** e **semester**. Estas informações complementares são armazenadas juntamente com o tipo de domínio, sob forma de uma tupla.

As propriedades estáticas e dinâmicas são representadas internamente por:

statprop(Classe, Nível, Papel, Nome_Propriedade, Tipo_Domínio).
dynprop(Classe, Nível, Papel, Nome_Propriedade, Tipo_Domínio).

Quando o domínio de uma propriedade é uma classe, o nome desta é associado à propriedade através do seguinte predicado:

classdom(Classe, Nível, Papel, NomeProp, NomeClasse).

É importante ressaltar que a associação dos tempos de transação e de validade às propriedades dinâmicas se dá somente no momento de execução de uma aplicação, não durante a sua especificação.

Os estados devem apresentar nomes únicos dentro de um mesmo papel. Cada estado é armazenado da seguinte forma:

state(Classe, Nível, Papel, Nome_Estado).

O papel básico tem seus estados pré-definidos (*active* e *suspended*). Os predicados que definem os estados do papel básico são definidos implicitamente pelo sistema cada vez que uma nova classe é definida. Novos estados para o papel básico não são aceitos.

Na definição de uma mensagem é indicado, além do nome da mensagem, o nome do papel para o qual a mensagem pode ser enviada ou do qual a mensagem pode ser recebida. Este papel pode pertencer a esta mesma classe ou a outra classe da especificação. Os nomes de mensagens devem ser únicos dentro de um mesmo papel. Uma mensagem pode apresentar argumentos, através dos quais são trocados valores entre diferentes papéis. O tipo de cada argumento é também definido quando da construção da especificação, devendo ser um dos tipos pré-definidos anteriormente citados. O número de argumentos é armazenado junto à definição da mensagem correspondente, sendo os pares de nome do argumento e seu tipo armazenados separadamente, com um indicativo de sua ordem. As mensagens enviadas e recebidas e os argumentos são representadas respectivamente pelos predicados:

mess(Classe, Nível, Papel, Nome_Mensagem, **to**, Nome_Classe, Nome_Papel, Nro_Argumentos).
mess(Classe, Nível, Papel, Nome_Mensagem, **from**, Nome_Classe, Nome_Papel, Nro_Argumentos).
argum(Classe, Nível, Papel, Mensagem, Nro_Ordem, Nome, Tipo).

As mensagens do papel básico também são pré-definidas. Estas mensagens servem para: (1) criar, suspender, ativar e destruir uma instância de uma classe; (2) criar, suspender, terminar, permitir e proibir instâncias de um papel; (3) permitir e proibir o envio ou o recebimento de uma mensagem; e (4) ativar ou desabilitar uma regras de transição. Cada vez que uma nova classe é criada, estas mensagens são definidas pelo sistema, de forma análoga à das demais mensagens. Novas mensagens para o papel básico não são aceitas pelo sistema.

Dois tipos de regras são utilizados em TF-ORM: regras de *integridade* e regras de *transição de estados*. Uma regra de integridade é representada internamente através do seguinte predicado:

statintegr (Classe, Nível, Papel, Condicao1, Condicao2).

Atualmente nenhuma análise das condições fornecidas está sendo realizada, ficando sua sintaxe a cargo do projetista da aplicação. Uma regra de transição pode apresentar uma condição associada, condição esta que representa geralmente uma condição de integridade dinâmica. Uma transição de estados, os argumentos de suas

mensagens e sua condição, quando existente, são representadas através dos seguintes predicados:

statetrans(Classe, Nível, Papel, Nro_Regra, Estado1, Estado2, Mens1, Mens2).
argumento(Classe, Nível, Papel, Nro_Regra,Direção,Nro_Ordem,Nome).
dynintegr(Classe, Nível, Papel, Nro_Regra, Condição).

A numeração das regras de transição de estados de um papel é feita pelo sistema. A associação entre uma regra e sua condição é definida pelo número correspondente à regra. Também para esta condição não está sendo atualmente feita nenhuma análise sintática, ficando isto sob responsabiblidade do projetista.

As abstrações de especialização e agregação são representadas através dos seguintes predicados:

subclass(Classe, Nível, Superclasse).
agregation(Nova_Classe, Classe_Componente, Nível).

Em caso de herança múltipla, para cada superclasse é definido um destes predicados. Também para cada classe componente de uma agregação é definido um predicado associando-a à nova classe gerada.

9.3.2. Modelo Interno da Biblioteca de Manutenção

Quando o projetista de aplicações definir uma nova classe sem se basear em nenhuma das classes definidas na biblioteca da classes, uma cópia desta classe é incluída na biblioteca de manutenção. No caso de terem sido definidos vários níveis de detalhamento da classe, somente o nível mais detalhado é armazenado. A classe será armazenada através da mesma representação semelhante à utilizada na base de dados especificação. Diferenças básicas são: (i) cada classe é associada a um domínio de aplicação, uma vez que nesta biblioteca teremos classes de diversas aplicações desenvolvidas; (ii) são excluídas as informaçòes referentes ao nível de detalhamento, uma vez que somente o nível mais detalhado é incluído nesta biblioteca; e (iii) metainformações são solicitadas ao projetista de aplicações pelo sistema, para facilitar o processo de identificação de classe e de papéis na biblioteca de classes - sinônimos de nomes de classes e de papéis, explicações textuais.

Os nomes dos predicados são diferenciados daqueles utilizados na especificação através de um prefixo, uma vez que durante a execução da especificação a base de informações é única. Os predicados referentes à biblioteca de manutenção são os seguintes:

m_classnamesyn(Nome_Classe, Sinônimo).
m_rolenamesyn(Classe, Nome_Papel, Sinônimo).
m_classdescr(Classe, Descricão).
m_roledescr(Classe, Papel, Descrição).
m_conddescr(Classe, Papel, Regra, Descrição).
m_agclass(Aplicação, Nome_Classe).
m_resclass(Aplicação, Nome_Classe).
m_proclass(Aplicação, Nome_Classe).
m_role(Classe, Nome_Papel).

m_statprop(Classe, Papel, Nome_Propriedade, Tipo_Domínio).
m_dynprop(Classe, Papel, Nome_Propriedade, Tipo_Domínio).
m_classdom(Classe, Papel, Nome_Propriedade, Classe).
m_state(Classe, Papel, Nome_Estado).
m_mess(Classe, Papel, Mensagem, **to**, Nome_Classe, Nome_Papel, Nro_Parâmetros).
m_mess(Classe, Papel, Mensagem, **from**, Nome_Classe, Nome_Papel, Nro_Argumentos).
m_argum(Classe, Papel, Mensagem, Nro_Ordem, Nome, Tipo).
m_statetrans(Classe, Papel, Nro_Regra, Estado1, Estado2, Mens1, Mens2).
m_argumento(Classe, Papel, Nro_Regra,Direção,Nro_Ordem,Nome).
m_statintegr (Classe, Papel, Condicao1, Condicao2).
m_dynintegr(Classe, Papel, Nro_Regra, Condicao).
m_subclass(Classe, Superclasse).
m_agregation(Nova_Classe, Classe_Componente).

Informações a respeito da forma de reutilização de classes são também armazenadas na biblioteca de manutenção pela ferramenta de apoio à especificação. É feita a estatística de quantas vezes a classe foi reutilizada sem nenhuma modificação e de quantas vezes foi reutilizada e posteriormente modificada. Estas informações são armazenadas através dos seguintes predicados:

m_equal_reuse(Classe, NrReusos).
m_modif_reuse(Classe, NrReusos).

Quando uma classe da biblioteca de classes for modificada para sua reutilização em uma especificação, as seguintes informações são armazenadas na biblioteca de manutenção: (i) uma cópia completa da classe final (nível mais detalhado) constante da reutilização; (ii) uma indicação de qual a classe de origem, pois o nome da classe pode ter sido modificado durante sua adaptação à especificação; e (iii) sugestões do projetista de aplicações para o engenheiro de classes. As informações relativas aos itens (ii) e (iii) são armazenadas através dos predicados:

m_reused_class(Classe, Classe_Biblioteca).
m_hint(Classe_Biblioteca, Sugestão).

O software gerenciador da especificação impede a existência, na biblioteca de manutenção, de duas classes com nomes idênticos, o que geraria problemas para a identificação de seus componentes. Caso encontre uma classe com o nome daquela que está sendo definida, pede ao projetista de aplicações um sinônimo para evitar a duplicação.

9.3.3. Modelo Interno da Biblioteca de Classes

As informações armazenadas nesta biblioteca apresentam forma semelhante àquelas dos dois bancos de dados anteriormente apresentados. Um prefixo específico faz a diferenciação entre os predicados que têm a mesma forma.

Cada classe definida nesta biblioteca é associada a um determinado domínio de aplicação. A análise das classes associadas a um determinado domínio de aplicação diminui bastante o tempo de busca de uma classe para reutilização. A representação interna das informações da biblioteca de classes é a seguinte:

b_aplication(Nome_Aplicação).
b_classnamesyn(Nome_Classe, Sinônimo).
b_rolenamesyn(Classe, Nome_Papel, Sinônimo).
b_classdescr(Classe, Descricão).
b_roledescr(Classe, Papel, Descrição).
b_agclass(Aplicação, Nome_Classe).
b_resclass(Aplicação, Nome_Classe).
b_proclass(Aplicação, Nome_Classe).
b_role(Classe, Nome_Papel).
b_statprop(Classe, Papel, Nome_Propriedade, Tipo_Domínio).
b_dynprop(Classe, Papel, Nome_Propriedade, Tipo_Domínio).
b_classdom(Classe, Papel, Propriedade, Classe).
b_state(Classe, Papel, Nome_Estado).
b_mess(Classe, Papel, Nome_Mensagem, **to**, Nome_Classe, Nome_Papel, Nro_Argumentos).
b_mess(Classe, Papel, Nome_Mensagem, **from**, Nome_Classe, Nome_Papel, Nro_Argumentos).
b_argum(Classe, Papel, Mensagem, Nro_Ordem, Nome, Tipo).
b_statetrans(Classe, Papel, Nro_Regra, Estado1, Estado2, Mens1, Mens2).
b_argumento(Classe, Papel, Nro_Regra, Direção, Nro_Ordem, Nome).
b_statintegr (Classe, Papel, Condição1, Condição2).
b_dynintegr(Classe, Papel, Nro_Regra, Condição).
b_subclass(Classe, Superclasse).
b_agregation(Classe, Classe_Componente).

São ainda definidos alguns predicados para fins de estatísticas de reutilização de classes:

b_equal_reuse(Classe, Nro_Reutilizações).
b_modif_reuse(Classe, Nro_Reutilizações).

10. ESTUDO DE CASO

Com o objetivo de avaliar a adequação do modelo de dados TF-ORM como ferramenta de especificação de sistemas de informação de escritórios, foi desenvolvido um estudo de caso completo, apresentado a seguir. Inicialmente é apresentada a aplicação a ser especificada, sendo em seguida detalhadas algumas das soluções adotadas na solução. A especificação completa é apresentada no Anexo 3 deste documento.

10.1. Apresentação da Aplicação

A aplicação a ser especificada consiste de uma agência locadora de fitas de vídeo e de jogos. Em um processo de análise dos requisitos desta aplicação foram identificados: (i) os agentes envolvidos (pessoas); (ii) os recursos manipulados; e (iii) os procedimentos executados.

10.1.1. Agentes

Cinco tipos de *agente* (pessoas com poder de decisão) são envolvidos nesta aplicação: (i) funcionários, (ii) proprietário(s) da locadora, (iii) clientes, (iv) fornecedores de fitas e de material de consumo e (v) compradores de fitas.

Um *funcionário* é identificado por um código único, apresentando propriedades complementares tais como: nome, endereço, data de nascimento, CPF, data de sua contratação e dispensa, salário, aptidões, função em que se enquadra, número de horas trabalhadas por semana, períodos de férias gozadas, etc. Conforme o sexo do funcionário, informações complementares são necessárias, tais como licenças-gestação para as mulheres. Diversos tipos de funcionário são identificados: um gerente geral, funcionários de atendimento, funcionários da contabilidade e funcionários de apoio (serviços gerais). Uma mesma pessoa pode apresentar dois empregos na locadora (por exemplo, um período no atendimento e um período na contabilidade) desde que o tempo total trabalhado não ultrapasse 40 (quarenta) horas semanais.

O *gerente geral* é um funcionário com funções próprias, as quais o diferenciam dos demais funcionários. Sua contratação é feita diretamente pelo proprietário principal da locadora. Deve trabalhar na locadora em tempo integral nesta função, não podendo acumular dois cargos diferentes. As principais funções do gerente geral são as seguintes:

- contratar e despedir funcionários;
- definir a alteração de salários dos demais funcionários, obedecendo sempre à lei de que nenhum salário pode diminuir;
- autorizar períodos de férias e de licenças especiais de funcionários, períodos estes solicitados pelos funcionários correspondentes;
- definir períodos de férias de funcionários, mesmo que não haja pedidos por parte destes;

- decidir sobre a compra de novas fitas e de quais as fitas a serem escolhidas;
- decidir quais as fitas que devem ser postas a venda e por quais preços;
- autorizar todos os pagamentos a fornecedores;
- autorizar o pagamento de salários a funcionários;
- definir dos valores de locação diária, da multa por fita perdida, do valor a ser cobrado por créditos (diárias em haver) de clientes e de qual a quantidade mínima de diárias que deve ser adquirida no sistema de créditos.

A locadora pode apresentar um ou mais *proprietários*. Um deles, entretanto, deve ser o proprietário principal, com poder de decisão. É identificado através de seu nome, seu endereço e seu CPF. O proprietário principal da locadora apresenta somente as seguintes funções:

- contrar e dispensar o gerente geral;
- mudar a função de um funcionário já existente para a função de gerente geral;
- definir alterações de salário do gerente geral, também obedecendo à lei de que um salário nunca pode diminuir;
- autorizar períodos de férias e de licenças especiais para o gerente geral.

Um *cliente* é identificado através de um código único. Informações adicionais necessárias a respeito de clientes são seu nome e seu endereço. Todo cliente pode autorizar até 3 (três) pessoas a alugar fitas em seu nome. Para cada uma destas pessoas fica registrado somente o nome. Informações adicionais necessárias a respeito de um cliente são a data de sua inscrição na agência de locação, quantas fitas estão atualmente em seu poder, quais as fitas que já alugou, número de créditos (diárias de locações em haver) que apresenta e o total de seu débito para com a locadora.

Outro tipo de agente envolvido com a locadora é representado pelos *fornecedores* de fitas e de materiais de consumo (de escritórios, limpeza, cafezinho, etc). São identificados através do CGC da empresa, do nome da empresa e de seu endereço. Dados a respeito dos produtos fornecidos devem ser registrados, tais como nome do produto, valor da última compra, data da última compra, etc.

Após um determinado tempo e por decisão do gerente geral, com base no número de locações apresentadas, algumas das fitas são colocadas a venda para impedir que a locadora fique com um acervo ultrapassado. O último tipo de agente identificado nesta aplicação é o dos *compradores* de fitas - empresas que adquirem estas fitas. É necessário conhecer tão somente o nome de cada comprador, seu CGC e endereço.

10.1.2. Recursos

As *fitas* são identificadas por um código único. Para cada fita devem ser conhecidos o nome de seu fornecedor, a data em que foi adquirida, quem a está alugando no momento e a data de início desta locação. Dois tipos de fitas estão presentes na locadora: fitas de filmes de vídeo e fitas de jogos. Informações relativas a cada um destes tipos de fitas devem ser registradas; para fitas de vídeo, o nome do filme, o gênero deste filme (comédia, drama, aventura, etc.), nome do diretor e dos atores principais, se o filme está em uma ou mais fitas e se o filme é legendado; para os jogos o nome do jogo, seu tipo e qual o equipamento necessário para poder utilizá-lo.

A locadora apresenta um *cadastro de todos os clientes*, um *cadastro de fitas* com os dados de todas as fitas existentes, um *cadastro de funcionários* e um *cadastro de fornecedores*. Existe um *cadastro de clientes desautorizados* o qual contém informações relativas a clientes para os quais não devem ser alugadas fitas por algum motivo particular (falta de pagamento, perda de fita sem ressarcimento da mesma, danos efetuados em fitas, etc.). A inclusão e retirada do nome de um cliente deste cadastro somente é efetuada com autorização do gerente geral.

Outro importante recurso existente na locadora é o *livro-caixa*. Neste são registrados todos os débitos e créditos com seus respectivos valores e identificação de devedor ou credor, além do movimento em moeda.

10.1.3. Processos

Os principais processos executados dentro da locadora podem ser divididos em processos de controle de donos da locadora, processos de controle de funcionários, processos de controle de clientes, processos de controle de fitas e atividades relativas à contabilidade.

O processo de *controle de donos* trata do registro das informações relativas aos donos da locadora e da alteração de algum de seus dados.

O *controle de funcionários* efetua o registro de novos funcionários, controla a alteração de alguma de suas características (tais como nome ou função), trata de pedidos de férias e de licensas especiais e de afastamento de funcionários da locadora. Novos dados para um funcionário são informados pelo interface representado pelo *mundo exterior*, enquanto que pedidos de férias, de licenças ou de afastamento são solicitados diretamente pelo funcionário envolvido. Para os funcionários em geral estes últimos pedidos são julgados e autorizados ou não pelo gerente geral. Para pedidos efetuados pelo gerente, o julgamento é feito pelo dono principal da locadora.

O processo de *controle de clientes* trata do registro de novos clientes com a definição das informações relativas a este cliente, da alteração dos dados referentes a um cliente já cadastrado e da inclusão e retirada de um cliente do conjunto de clientes não autorizado a alugar fitas. Pedidos de registro de novos clientes devem ser autorizados pelo gerente geral. Informações relativas a dados dos clientes são fornecidos pelo interface do mundo exterior. Nomes de pessoas autorizadas a alugar fitas em nome de um cliente

somente podem ser fornecidas pelo cliente envolvido. A inclusão e retirada de clientes do conjunto de clientes não autorizados é efetuada diretamente pelo gerente geral.

O processo de *controle de fitas* apresenta como principais atividades o registro de novas fitas adqüiridas e o controle de locações. Os dados relativos a novas fitas adqüiridas são fornecidos diretamente pelo gerente geral.

A locação de uma fita somente pode ser efetivada se todas as seguintes condições forem satisfeitas: (i) o pedido foi efetuado por um cliente cadastrado ou uma pessoa autorizada por um cliente cadastrado; (ii) o cliente que solicita a fita não está no cadastro de clientes para os quais não podem ser alugadas fitas; (iii) a fita está disponível para locação; (iv) no máximo 5 fitas podem ser alugadas por cliente; e (iv) o cliente não está no momento de posse de alguma fita por um período superior a 1 (um) mês. Na devolução de uma fita são calculadas as diárias a serem cobradas, descontando estas do total de diárias em haver caso existam. Informações relativas à locação são registradas no cadastro do cliente e no livro-caixa.

As principais atividades da *contabilidade* da locadora são referentes a valores (pagos ou a pagar) de relativos a locações, a compra de fitas ou de outros produtos (produtos de escritório, de limpeza, cafezinho, etc.), a multas aplicadas a clientes em caso de extravio de fitas, a venda de fitas e ao pagamento de salários a funcionários.

Novas fitas são adquiridas de tempos em tempos, a intervalos decididos pelo gerente geral. A decisão de aquisição de uma determinada fita é tomada pelo gerente. Ele fornece ao setor de compra os dados da fita e o fornecedor. Uma vez efetuado o pedido de compra, o seu pagamento pode ser efetuado de uma vez só ou debitado. A nova fita é cadastrada e está disponível para locação.

As fitas não permanecem indefinidamente na locadora. Uma fita pode sair do acervo da locadora por venda, destruição ou extravio. A decisão de quais as fitas que devem ser colocadas a venda e por quais preços é de responsabilidade do gerente geral. Uma fita é destruída por determinação do gerente geral sempre que apresentar algum defeito. E uma fita pode ser estraviada por algum cliente ou na própria locadora. No caso de estravio por cliente, este será penalizado.

A locadora apresenta atendimento a clientes todos os dias do ano, inclusive sábados, domingos e feriados, sendo cobradas diárias normais também para estes dias.

10.2. Especificação através do Modelo TF-ORM

Na modelagem realizada são definidas classes de agentes para cada um dos agentes identificados: as pessoas, os fornecedores e os compradores de fitas. As pessoas podem desempenhar diferentes papéis: funcionário, dono e cliente. Os funcionários apresentam uma propriedade que distingüe sua função, sendo uma destas funções a de gerente. O gerente é representado, portanto, por um funcionário com funções especiais, obedecendo às regras gerais de um funcionário - somente seu salário e sua contratação é que dependem do dono principal. Clientes não autorizados a alugar fitas constituem uma

classe especial, subclasse da classe pessoa, herdando somente o papel de cliente. A inclusão de algum cliente nesta subclasse depende de decisão do proprietário principal da locadora.

Classes de recursos são definidas para as fitas e para o caixa da locadora. Fitas de filmes e fitas de jogos são modelados como subclasses da classe genérica fita, apresentando suas características próprias. Os tipos de filmes de vídeo e de jogos também são representados através de classes de recursos. O caixa da locadora apresenta os valores em moeda e os débitos e créditos que a locadora possui.

Os procedimentos principais da locadora são representados através de classes de processos. São procedimentos a serem executados com fins de controle de pessoas, de fitas e procedimentos relativos à contabilidade da locadora. Para o controle de pessoas foram criadas três classes de processos diferentes: uma para os donos, a segunda para os funcionários e a terceira para os clientes.

Procedimentos relativos aos donos da locadora são executados quando da definição de um novo dono com os valores de suas propriedades e da alteração de alguma de suas propriedades.

No caso da classe de controle dos funcionários, diferentes papéis representam as atividades que podem ser executadas dentro do processo de controle: (i) controle geral - definição dos valores iniciais de propriedades de um funcionário, alteração de alguma de suas propriedades, pedidos de férias, de licenças especiais, e de saída do emprego; (ii) cadastro de um novo funcionário, com a geração de um código único para ele; (iii) controle de início e fim de férias e de licenças especiais; e (iv) atividades que são executadas quando o funcionário sai do emprego.

O controle de clientes constitui-se do tratamento de pedidos de novas inscrições, da alteração dos valores de alguma das propriedades de algum cliente e da criação de uma nova instância de cliente, associando a ele um código único.

Os procedimentos relativos ao controle de fitas também são subdivididos em diversos papéis, modelando diferentes atividades a serem executadas; (i) controle geral de fitas, compreendendo pedidos de registro de uma nova fita, atividades a serem executadas quando for informada a perda de uma fita dentro da locadora ou por algum cliente e o tratamento da destruição de uma fita; (ii) o cadastramento de uma nova fita, associando-lhe um código único; e (iii) o controle de locações.

A geração dos códigos únicos para funcionários, clientes e fitas é representada através de uma classe de processo que fornece este código através de um papel específico para cada um dos casos considerados.

A contabilidade da locadora é representada através de uma classe de processo única. As diferentes atividades que envolvem a contabilidade são agrupadas nos seguintes papéis: (i) contabilidade relativa a locações; (ii) aquisições de fitas; (iii) aquisição de outros materiais, tais como materiais de escritório, de limpeza, etc; (iv) atividades relativas a perda de fitas; (v) atividades a serem executadas quando uma fita é posta a venda; (vi)

pagamento de salários a funcionários; e (vii) controle de pagamentos em geral - valores recebidos como pagamentos de locações ou de compra de fitas e pagamentos efetuados pela locadora, pela compra de fitas.

Uma última classe de processo é definida para representar informações de um relógio, para fins de controle de datas de início e de final de férias e de licensas especiais.

Na modelagem apresentada não foram considerados processos de listagem, tais como listagem de: (i) todos os sócios; (ii) dos sócios que alugaram filmes em um determinado período de tempo; (iii) sócios inadimplentes; (iv) de todos os filmes; (v) de filmes por gênero; (vi) de filmes mais locados; (vii) dos filmes em poder de um cliente; (viii) de todos os filmes alugados por um cliente; (ix) de todos os filmes atualmente alugados, com informação de quem os está alugando; etc. Estas listagens são necessárias para, por exemplo, fornecer relatórios, editar jornais de propaganda e para estudos estatísticos para embasar decisões do gerente geral. Consultas com a finalidade de fornecer informações a clientes também não estão sendo modeladas, tais como quando uma determinada fita deverá ser devolvida, se um determinado filme já foi alugado pelo cliente, etc. As informações necessárias a estas listagens e consultas estão todas representadas junto às informações de filmes, clientes e locações.

Quando utilizados nomes de classes associados ou não a nomes de papéis como argumentos de mensagens, a passagem de um argumento na realidade significa a passagem do identificador da classe (*oId*) ou do papel (*rId*) relativo ao objeto considerado. Esta forma de passagem de valores simplifica sobremaneira a passagem de valores entre os diferentes processos a serem executados.

Na modelagem foram utilizadas as seguintes **convenções**:

- definição dos nomes de classes com letras maiúsculas;
- definição dos nomes de papéis iniciando por letras maiúsculas;
- nomes de classes em letras maiúsculas quando referenciadas como classe de origem ou de destino na lista das mensagens de um papel;
- nomes de papéis iniciando por letras maiúsculas quando referenciados como papéis de origem ou de destino na lista de mensagens;
- variáveis em listas de argumentos iniciando sempre por letra maiúscula;
- nomes de propriedades, de classes e de papéis em listas de argumentos iniciando sempre por letras minúsculas.;
- comentários delimitados pelos símbolos **/*** e ***/**.

A especificação completa do estudo de caso está no Anexo 3 deste documento.

10.3. Comentários a Respeito da Escolha do Estudo de Caso

Foi escolhida esta aplicação para exemplificar a utilização do modelo de dados TF-ORM em uma especificação por apresentar, de uma forma não muito complexa, as características principais que se quer representar, ou seja:

- os mais diversos tipos de propriedades quanto ao número de valores que podem apresentar (multivaloradas ou monovaloradas), e quanto a sua variação com o tempo; as diversas combinações que se apresentam são: (i) propriedades monovaloradas em um instante mas multivaloradas com a passagem do tempo, como por exemplo a propriedade "salário"; (ii) propriedades multivaloradas em um instante e também com a passagem do tempo, como no caso das "aptidões" de um funcionário - em um momento pode ter mais de uma aptidão e as aptidões podem mudar com o passar do tempo; (iii) propriedades monovaloradas em um mesmo instante e que não mudam com o passar do tempo, como no caso do sexo de uma pessoa; e (iv) propriedades multivaloradas em um instante mas que não mudam com o passar do tempo, como por exemplo, o tipo do filme contido na fita, que pode ser, por exemplo, guerra e aventura simultaneamente;

- restrições de integridade dinâmicas, ou seja, que comparam mais de um estado do banco de dados, como no caso de uma alteração de salário, não sendo permitido que um salário posteriormente (em relação à ordem temporal) definido seja inferior a um anteriormente definido;

- a possibilidade de existirem simultaneamente instâncias de papéis diferentes para um mesmo objeto, como no caso de uma pessoa que pode ser simultaneamente funcionário e cliente da locadora;

- a possibilidade de se apresentar simultaneamente mais de uma instância de um mesmo papel para um mesmo objeto, como quando uma pessoa apresenta dois contratos de emprego na mesma locadora (por exemplo, um turno em atendimento e outro na contabilidade);

- a existência de processos de decisão humana, caracterizando procedimentos não bem estruturados; e

- a utilização de condições associadas às regras de transição de estados com a finalidade de identificar a instância que deve efetuar a transição - por exemplo, uma mensagem de modificação do nome de um empregado é recebida por todas as instâncias do papel *empregado* de todos os objetos *pessoa*; através da condição, somente a instância de papel cujo identificador coincide com aquele passado como parâmetro é que efetua a transição;

- a possibilidade de definir subclasses com alguns dos tipos previstos de herança de papéis - herdados, estendidos e novos; na definição da classe WOMAN são herdados os papéis de cliente e de dono, sendo estendido o papel de funcionário; já nas classes VIDEO e GAME é herdado um papel e é definido um novo papel.

11. CONCLUSÃO

O principal objetivo deste trabalho é a definição de um método para ser utilizado em especificações de requisitos de sistemas de informação de escritórios. Foi selecionado como método de especificação a definição do *modelo de dados* a ser implementado no banco de dados que representa o sistema.

O desenvolvimento de especificações é uma tarefa bastante complexa, devendo todas as características (estáticas e comportamentais) da aplicação ser representadas. A especificação de uma aplicação geralmente é muito extensa, sendo dispendido um tempo considerável em sua elaboração. Um aspecto importante no desenvolvimento de especificações é a verificação da corretude destas, devido à complexidade que apresentam. Uma maneira de gerar especificações mais corretas e em menor espaço de tempo é através da reutilização de especificações anteriormente construídas e já validadas através de implementações.

Neste trabalho optou-se por utilizar um *modelo de dados orientado a objetos* para especificar os sistemas de informação de escritórios. Este paradigma permite a definição de uma biblioteca de classes de objetos definidas em especificações, as quais podem ser reutilizadas de maneira bastante eficiente em especificações posteriores. A biblioteca deve ser criada por um especialista, sendo inicialmente povoada com um conjunto de classes identificadas no domínio de sistemas de informação de escritórios. A manutenção desta biblioteca deve ser feita periodicamente, sempre por um especialista. Novas classes são acrescentadas à biblioteca e as existentes são modificadas ou eliminadas pelo especialista como resultado de uma análise das especificações realizadas com base na biblioteca.

A representação das características dinâmicas, tais como a evolução dos objetos dentro do escritório, requer a possibilidade de representação de propriedades temporais. O modelo de dados utilizado na especificação deve permitir a representação de aspectos temporais tanto para definição de dados definidos em um domínio temporal como para representar a evolução dos valores assumidos pelos objetos durante sua existência. O modelo de dados utilizado para especificar sistemas de informação de escritórios proposto neste trabalho será, portanto, um *modelo de dados orientado a objetos temporal.*

Para o desenvolvimento deste trabalho foi selecionado um modelo de dados já existente, o modelo F-ORM (*Functionality in Objects with Roles Model*) [DEA 91a,b,c], para servir de base para o modelo proposto. O modelo F-ORM foi desenvolvido com a finalidade de captar as funcionalidades de sistemas de informação de escritórios. Trata-se de um modelo de dados orientado a objetos que utiliza o conceito de papéis para representar os diferentes comportamentos de um objeto. Aspectos temporais, entretanto, não são representados no modelo F-ORM. Também não é possível a representação de parcelas de trabalho pouco estruturado, características importante da influência exercida por pessoas neste tipo de aplicação. O modelo F-ORM foi estendido para incorporar não somente os aspectos temporais, mas também outros aspectos importantes identificados nas aplicações de sistemas de informação de escritórios, em especial o trabalho pouco

estruturado. O modelo resultante foi denominado modelo TF-ORM (*Temporal Functionality in Objects with Roles Model*).

Devido às características das aplicações consideradas, identificou-se que a possiblidade de representação de aspectos temporais é de fundamental importância. Uma análise das formas de representação temporal identificou que não existe unanimidade na forma de representação de aspectos temporais. Alguns métodos orientados a objetos foram estendidos para incorporar esta forma de representação, mas a influência da representação temporal neste paradigma ainda não foi estudada com profundidade. A principal contribuição deste trabalho é a proposta de uma forma uniforme de representação de aspectos temporais em modelos orientados a objetos, captando tanto a evolução de cada instância durante seu tempo de vida como a variação de suas propriedades dinâmicas.

Quatro diferentes conceitos para representar informações temporais foram utilizados: (1) um conjunto de tipos de dados temporais juntamente com funções e operações associadas, para serem utilizados na definição de propriedades; (2) a associação do tempo de definição a instâncias e a propriedades dinâmicas; (3) um valor especial para propriedades, *null*, representando períodos em que uma propriedade não apresenta valor válido; e (4) condições temporais associadas às regras de transição de estados e a regras de integridade, escritas em uma linguagem de lógica temporal. O modelo resultante utiliza dois rótulos de tempo para valores de propriedades dinâmicas: o tempo de transação (instante em que o valor foi introduzido no banco de dados) e o tempo de validade (intante correspondente ao início da validade deste valor no mundo real, instante este que pode ser anterior, igual ou posterior ao tempo de transação).

Os tipos de classes existentes no modelo F-ORM foram ampliados, sendo definida no modelo TF-ORM a classe "agente" na qual pode ser representada uma parcela de trabalho tipicamente humano: o resultado de uma decisão. O trabalho pouco estruturado foi representado através de mensagens resultantes de processos de decisão, mensagens estas que são tratadas em regras de transição de estados apropriadas. A representação dos resultados de processos de decisão em uma especificação representa um importante passo no sentido de permitir uma melhor representação de uma aplicação. Esta parcela de trabalho não estruturado não é representada na grande maioria dos modelos de dados existentes para fins de especificações de requisitos, sendo somente o trabalho estruturado considerado.

O modelo TF-ORM resultante permite, portanto, a representação de dois aspectos considerados como fundamentais na especificação de SIEs - dos aspectos temporais e do efeito de decisões humanas no trabalho desenvolvido. Estes dois aspectos não são considerados no maior parte dos modelos utilizados para a especificação de aplicações nesta área. É o caso, por exemplo, do ambiente TODOS [HEI 88] e dos sistemas IRIS [DER 86, FIS 87, LYN 84, 86a,b] e OFFIS [BRA 84, KON 82], os quais consideram somente trabalho estruturado e não permitem a representação de aspectos temporais.

A possibilidade de recuperação de informações de um banco de dados que implemente o modelo de dados proposto foi também considerada. Foi definida uma linguagem de consulta para o modelo de dados TF-ORM. Especial atenção foi dada às consultas temporais, que envolvem informações relativas a: (i) dados definidos em domínios temporais; (ii) dados referentes a intervalos temporais ou a datas diferentes da data atual; e (iii) valores que satisfaçam condições temporais. Foi realizado um estudo a respeito da influência do paradigma de orientação a objetos em relação a este tipo de comandos, considerando a navegação ao longo das possíveis hierarquias de classes e a possibilidade de existência de diversas instâncias de uma classe e de papéis satisfazendo as condições da recuperação.

Como no modelo proposto são armazenados tanto o tempo de transação como o de validade, a linguagem de recuperação pode ser utilizada para recuperar informações a referentes ao estado atual do banco de dados (informações atualmente válidas), a respeito de estados passados e futuros (informações válidas no passado e que serão válidas em estados futuros, de acordo com o atual conhecimento dos dados) e referentes a histórias passadas do banco de dados (informações que se acreditava como válidas em algum momento do passado).

A recuperação de informações temporais é um assunto ainda em aberto nas pesquisas atuais, em especial quando considerado o paradigma de orientação a objetos. O presente trabalho apresenta uma taxonomia para as possíveis consultas sobre um banco de dados temporal na qual são consideradas as diferentes histórias que podem ser identificadas neste banco de dados. Esta taxonomia constitui uma contribuição importante para o campo de recuperação de informações, tendo em vista as últimas tentativas da comunidade de modelagem temporal de estabelecer uma taxonomia semelhante e que não obtiveram muito sucesso [JEN 93b].

O modelo TF-ORM aumenta a possibilidade de reutilizar partes de especificações através do emprego de papéis. O paradigma de orientação a objetos, por si só, possibilita uma ênfase na reutilização, através da utilização de bibliotecas de classes montadas a partir de especificações realizadas. Estas bibliotecas podem ser consultas e suas classes reutilizadas em diversas especificações. A representação dos diferentes comportamentos através do conceito de papéis permite que na reutilização de uma classe sejam aproveitados somente alguns dos papéis representados. A adaptação das classes reutilizadas é também facilitada ao ser permitida a definição de novos papéis representando novos comportamentos, e a nova definição de algum papel já definido.

Um estudo de caso exaustivo foi desenvolvido com o objetivo de validar o modelo de dados proposto e sua eficiência na especificação deste tipo de sistema de informação. A especificação resultante, embora bastante extensa, mostrou que o modelo de dados representa de maneira muito clara e suficientemente completa a aplicação proposta. Entretanto, novos trabalhos deverão ser desenvolvidos nesta área.

Para complementar a validação do modelo de dados proposto foi implementado o protótipo de um ambiente de apoio a especificações que faz uso de uma biblioteca de classes. Através deste ambiente a especificação de uma aplicação é construída

gradualmente, estando disponíveis opções de listagem das partes já definidas e de informações da biblioteca de classes para serem reutilizadas. As especificações geradas através deste ambiente apresentam pouca possibilidade de erro, uma vez que são efetuados vários procedimentos de verificação das informações fornecidas.

Como prosseguimento deste trabalho está sendo implementado um compilador que reconhece a linguagem TF-ORM e gera um esquema conceitual do SGBD O_2 [DEU 91]. Em paralelo está sendo realizado o desenvolvimento de um mapeamento do modelo TF-ORM para o relacional, em cooperação com a UFMG - DCC. Além disso, o ambiente implementado está sendo ampliado e será objeto de futuros trabalhos para aperfeiçoamento de sua interface e de suas funções.

ANEXO 1

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, sparada 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 símbolbos "[" 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> ")"
<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> [**with** <roles>]
| <message> to **EXTERNAL_WORLD** [**with** <roles>]
| <message> **to itself** [**with** <roles>]
| <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> ::= <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** ["(" [<role 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

ANEXO 2

SINTAXE DA LINGUAGEM DE CONSULTA TF-ORM

<query> ::=
SELECT <target clause>
FROM <object identification clause>
<search specification>
[**AS ON** <temporal instant clause>]
<search specification> ::= **WHERE** <search clause> | **WHEN** <search clause>
<target clause> ::=
<data output>
| <temporal output>
| <data output> **and** <temporal output>
<data output> ::= <property name list> | "*"
<temporal output> ::=
DATE | TRANSACTION_DATE | PERIOD | TRANSACTION_PERIOD
<property name list> ::= <property name> ["," <property name list>]
<property name> ::= <identifier>
<object identification clause> ::=
[<oid variable> ","] <class name>
| [<oid variable> "," [<role variable> ","]] <role name>
| [<oid variable> ","] <class name> "." [<role variable> ","] <role name>
| <class name> "." <role name>
<class name> ::= <identifier>
<role name> ::= <identifier>
<rule name> ::= <identifier>
<search clause> ::= <query logical expression>
<query logical expression> ::= <logical expression>
| <logical expression> **and** <predefined query predicate>
<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> "," <property value> ")"
| **is_valid_at** "(" [<id variable> ","]
<property name> "," < property value> ","<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 name argument> ")"
| **before** "(" <function argument> "," <function argument> ")"
| **equal** "(" <function argument> "," <function argument> ")"

<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> ")"

<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> ")"
| **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>
<temporal instant> ::= <number> "/" <number> "/" <number> ","
<number> ":" <number>
<id variable> ::= <oid variable> | <role variable>
<oid variable> ::= <variable>
<role variable> ::= <variable>
<variable> ::= <identifier>
<function argument> ::= [<id variable> ","] <property name>
| <temporal instant> | <variable>
<function name argument> ::=
[<id variable> ","] <property name> | <variable>
<predefined query predicate> ::=
PERIOD "=" "[" <date value>, <date value>"]"
| **DATE** "=" <date value>
<temporal instant clause> ::= <date value><predefined state predicate> ::= **state** "(" [
<oid variable> ","] <predefined state> ")"
<predefined state> ::= **active** | **suspended**
<state predicate> ::= <defined state predicate> | <predefined state predicate>
<defined state predicate> ::= **state** "(" [<id variable> ","] <state name> ")"
<state name> ::= <identifier>
<property value> ::= <variable> | """ <string> """
<string> ::= "'" { <any character including blanck> }+ "'"

ANEXO 3

ESPECIFICAÇÃO DO ESTUDO DE CASO

```
/* ---------------------- */
/* AGENT CLASSES */
/* ---------------------- */

agent class (
  PERSON,

  < Base_role,
    static properties = {
        (gender, {"F", "M"} ),
        (cpf, integer),
        (date_birth, date) },
    dynamic properties = {
        (name, string),
        (address, string),
        (telephone, set_of integer) },
    rules = {
        r1: msg(← create_object) ⇒ state(active),
        r2: state(active) ⇒ msg(→ allow_role(employee)),
        r3: state(active) ⇒ msg(→ allow_role(owner)),
        r4: state(active) ⇒ msg(→ allow_role(client)) }
  >,

  < Employee,
    static properties = {
        (code, integer) },
    dynamic properties = {
        (salary, real),
        (admission_date, date),
        (demission_date, date),
        (skills, set of string),
        (function, {"manager", "attendance", "accountance", "support"} ),
        (hours_week, day),
        (vacation, interval(date, closed)),
        (license, interval(date, closed)) },
    messages = {
        initial_values(Rid: integer, Code: integer, CPF: integer, Name: string, Addr: string,
              Gender: {F, M}, D-birth: date, AdmissionDate: date, Salary: real,
              Function: {"manager", "attendance", "accountance", "support"},
              HourWeek: hours )
              from EMPLOYEE_CONTROL.New_Employee_Registration,
        change_name (Emp: employee, Name: string)
              from EMPLOYEE_CONTROL.General_Employee_Control,
        change_address (Emp: employee, Address: string)
              from EMPLOYEE_CONTROL.General_Employee_Control,
        change_phone (Emp: employee, Old: integer, New: integer)
              from EMPLOYEE_CONTROL.General_Employee_Control,
        add_phone (Emp: employee, Number: integer)
              from EMPLOYEE_CONTROL.General_Employee_Control,
```

new_salary (Emp: employee, Value: real, Valid_Time: date)
    to EMPLOYEE_CONTROL.General_Employee_Control,
modify_salary (Emp: employee, NewValue: real, Valid_Time:date)
    from EMPLOYEE_CONTROL.General_Employee_Control,
add_skill (Emp: employee, NewSkill: string)
    from EMPLOYEE_CONTROL.General_Employee_Control,
remove_skill (Emp: employee, SkillName: string)
    from EMPLOYEE_CONTROL.General_Employee_Control,
new_function (Emp: employee,
    Function: {"manager", "attendance", "accountance", "support"},
    Valid_Time: date) to EMPLOYEE_CONTROL.General_Employee_Control,
change_function (Emp: employee,
    NewFunction:{"manager", "attendance", "accountance", "support"})
    from EMPLOYEE_CONTROL.General_Employee_Control,
mgr_change_HW(Emp: employee,HWValue: hours,ValidDate:date)
    from EMPLOYEE_CONTROL.General_Employee_Control,
new_hours_week (Emp: employee, HWValue: hours, ValidTime: date)
    to EMPLOYEE_CONTROL.General_Employee_Control,
change_hours_week(Emp: employee, NewHWValue: hours, Valid_Time: date)
    from EMPLOYEE_CONTROL.General_Employee_Control,
add_employee(Name: string, CPF: integer, Addr: string, Gdr: {"F","M"},
    Dbirth: date, Date: date, Salary: real,
    Function: {"manager", "attendance", "accountance", "support"},
    HoursWeek: hours) to EMPLOYEE_CONTROL.General_Employee_Control,
add_employment (Oid: integer, Date: date, Salary: real,
    Function: {"manager", "attendance", "accountance", "support"},
    HoursWeek: hours) to EMPLOYEE_CONTROL.General_Employee_Control,
emp_values(Oid: integer, Date: date, Salary: real,
    Function:{"manager", "attendance", "accountance", "support"},HWeek: hours)
    from EMPLOYEE_CONTROL.General_Employee_Control,
ask_vacations (Emp: employee, Begin: date, End: date)
    to EMPLOYEE_CONTROL.General_Employee_Control,
confirm_vacations (Emp: employee, Begin: date, End: date)
    from EMPLOYEE_CONTROL.General_Employee_Control,
ok_vacations (Emp: employee, Begin: date, End: date)
    to EMPLOYEE_CONTROL.General_Employee_Control,
nok_vacations (Emp: employee)
    to EMPLOYEE_CONTROL.General_Employee_Control,
record_vacations (Emp: employee, Begin: date, End: date)
    from EMPLOYEE_CONTROL.General_Employee_Control,
vacations_indefered (Emp: employee)
    from EMPLOYEE_CONTROL.General_Employee_Control,
begin_vacations (Emp: employee)
    from EMPLOYEE_CONTROL.Employee_Vacation_Control .
end_vacations (Emp: employee)
    from EMPLOYEE_CONTROL.Employee_Vacation_Control,
ask_license (Emp: employee, Type: string, Begin: date, End: date)
    to EMPLOYEE_CONTROL.General_Employee_Control,
confirm_license (Emp: employee, Type: string, Begin: date, End: date)
    from EMPLOYEE_CONTROL.General_Employee_Control,
ok_license (Emp: employee, Type: string, Begin: date, End: date)
    to EMPLOYEE_CONTROL.General_Employee_Control,
nok_license (Emp: employee, Type: string)
    to EMPLOYEE_CONTROL.General_Employee_Control,
record_license (Emp: employee, Type: string, Begin: date, End: date)
    from EMPLOYEE_CONTROL.General_Employee_Control,

```
    license_indefered (Emp: employee)
        from EMPLOYEE_CONTROL.General_Employee_Control,
    ask_end_employment(Emp: employee, valid_date: date)
        to EMPLOYEE_CONTROL.General_Employee_Control,
    analyze_emp_end (Emp: employee, EndTime: date)
        from EMPLOYEE_CONTROL.General_Employee_Control,
    ok_end_employment (Emp: employee, valid_time: date)
        to EMPLOYEE_CONTROL.General_Employee_Control,
    nok_end_employment (Emp: employee)
        to EMPLOYEE_CONTROL.General_Employee_Control,
    record_end_employment (Emp: employee, Time: date)
        from EMPLOYEE_CONTROL.General_Employee_Control,
    end_employment_indefered (Emp: employee)
        from EMPLOYEE_CONTROL.General_Employee_Control,
    mgr_analyze_new_client (CPF: integer, Name: string, Addr: string,
        Phone: set_of integer, Gdr: {"F","M"}, Dbirth: date)
        from CLIENT_CONTROL.General_Client_Control,
    add_client (CPF: integer, Name: string, Addr: string, Phone: set_of integer,
        Gdr: {"F","M"}, Dbirth: date)
        to CLIENT_CONTROL.General_Client_Control,
    new_client_nok (CPF: integer, Name: string, Addr: string)
        to CLIENT_CONTROL.General_Client_Control,
    mgr_analyze_former_client (Client: person.client)
        from CLIENT_CONTROL.General_Client_Control,
    add_former_client (Client: person.client)
        to CLIENT_CONTROL.General_Client_Control,
    former_client_not_accepted (Client: person.client)
        to CLIENT_CONTROL.General_Client_Control,
    record_bad_client (Client: person.client, Reason: string)
        to CLIENT_CONTROL.General_Client_Control,
    remove_bad_client_record (Client: person.client)
        to CLIENT_CONTROL.General_Client_Control,
    include_film_type (Type: string, Description: string) to TYPE. Film_Type,
    remove_film_type(Type: string) to TYPE.Film_Type,
    include_game_type (Type: string, Description: string) to TYPE. Game_Type,
    remove_game_type(Type: string) to TYPE.Game_Type,
    acquire_video (Date: date, Furn: furnisher, Film: string,
        Type:set_of type.film_type, Nrwomans: integer, Director: string,
        Actors: set_of string, Legend: {"Y", "N"}, Price: real)
        to ACCOUNTANCE.Tape_Acquisition,
    acquire_game (Date: date,Furn: furnisher,Name: string,
        Type:set_of type.game_type,Equip: string, Price: real)
        to ACCOUNTANCE.Tape_Acquisition,
    acquire_material(Material: string, Furnisher, furnisher, Price: real)
        to ACCOUNTANCE.General_Acquisition,
    tape_loose_fine(Value: real) to ACCOUNTANCE.Tape_Loose,
    sell_tape (Tape: tape, Price: real) to ACCOUNTANCE.Tape_Sale,
    destruct_tape (Tape: tape) to TAPE_CONTROL.General_Tape_Control,
    update_min_credits(Quantity: integer) to ACCOUNTANCE.Payment_Control,
    update_credit_price(Price: real) to ACCOUNTANCE.Payment_Control,
    update_rental_price (Value: real) to ACCOUNTANCE.Rentals,
    salary_payment_authorization (Employee:person.employee, Value: real)
        to ACCOUNTANCE.Salary,
    pay(Credor: furnisher, Value: real) to ACCOUNTANCE.Payment_Control },
decisions = {
    new_employee (Name: string, CPF: integer, Addr: string, Gdr:{"F","M"},
```

```
        Dbirth: date, BeginningDate: date, Salary: real,
        Function:{"manager", "attendance", "accountance", "support"},
        HoursWeek: hours ),
    new_employment (Emp: employee, Date: date, Salary: real,
        Function:{"manager", "attendance", "accountance", "support"},
        HoursWeek: hours ),
    dec_new_salary (Emp: employee, Value: real, Valid_Time: date),
    change_emp_function(Emp: employee,
        Function: {"manager", "attendance", "accountance", "support"},
        ValidTime: date),
    change_emp_hours_week(Emp: employee, HWValue: hours, ValidTime: date),
    vacation_request(Begin: date, End: date),
    vacation_decision (Decision:string, Emp: employee, Beginning:date, Ending:date),
    license_request(Type: string, Begin: date, End: date),
    license_decision(Decision: string, Emp: employee, Type: string,
        Beginning: date, Ending: date),
    end_emp_request (Date: date),
    end_emp_decision (Decision: string, Emp: employee, Date: date),
    decision_end_employment(Decision: string, Emp: employee, FinalDate: date),
    new_client_accepted (CPF: integer, Name: string, Addr: string,
        Phone: set_of integer, Gdr: {"F","M"}, Dbirth: date),
    new_client_not_accepted (CPF: integer, Name: string, Addr: string),
    former_client_accepted (Client: person.client),
    former_client_nok (Client: person.client),
    bad_client_record (Client: person.client, Reason: string),
    remove_bad_client (Client: person.client),
    new_film_type (Type: string, Descr: string),
    wrong_film_type (Type: string),
    new_game_type (Type: string, Descr: string),
    wrong_game_type (Type: string),
    buy_new_video(Date: date, Furn: furnisher,Film: string, Type: set_of type.film_type,
            NrTapes: integer, Director: string, Actors: set_of string,
            Legend: {"Y", "N"}),
    buy_new_game(Date: date, Furn: furnisher,Name: string,
            Type: set_of type.film_type, Equipment: string),
    buy_material(Material: string, Furnisher: furnisher, Price: real)
    new_rental_price(Price: real),
    new_credit_quantity(Quantity: real),
    new_credit_price(Price: real),
    change_loose_fine_value(Value: real),
    sell_a_tape (Tape: tape, Price: real),
    destruct_a_tape(Tape: tape),
    new_rental_price(Price: real),
    payment_employee(Employee: employee, Value: real),
    payment_authorization(Creditor: cash.creditor, Value: real) },
states = {
    beginning_employment,
    employed,
    in_vacations,
    waiting_end_employment },
rules = {
    begin:
        msg(← add_role) ⇒ state(beginning_employment),
    initialization:
        state(beginning_employment),
```

```
        msg(← initial_values(Rid, Code, CPF, Name, Addr, Gender, Dbirth,
            Adate, Salary, Func, Hours) ⇒
        state(employed),
changing_name:
        state(employed), msg(← change_name(Code, Name)) ⇒
        state(employed);
        (value(code) = Code),
changing_addr:
        state(employed), msg(← change_address(Emp, Address)) ⇒
        state(employed);
        (value(rId) = Emp),
changing_telephone:
        state(employed), msg(← change_phone(Emp, Old, New)) ⇒
        state(employed);
        (value(rId) = Emp),
adding_new_telephone:
        state(employed), msg(← add_phone(Emp, Number)) ⇒
        state(employed);
        (value(rId) = Emp),
manager_deciding_new_salary:
        state(employed), decision(dec_new_salary (Emp, Value, Valid_Time)) ⇒
        msg(→ new_salary (Emp, Value, Valid_Time)), state(employed);
        (value(function) = "manager" and value(rId) ≠ Emp),
        /* Only the manager can decide a new salary of employees; he cannot decide his
           own salary which is done by the owner. */
salary1:
        state(employed), msg(← modify_salary (Emp, NewValue, Valid_Time)) ⇒
        state(employed) ;
        (value(rId) = Emp and value(salary) < NewValue),
        /* The salary of an employee is only changed if the new value is bigger that the
           last one. */
salary2:
        state(in_vacations), msg(← modify_salary (Emp, NewValue, Valid_Time)) ⇒
        state(in_vacations) ;
        (value(rId) = Emp and value(salary) < NewValue),
        /* The salary of an employee is changed even if he is in vacatons. */
add_skill:
        state(employed), msg(← add_skill(Emp, NewSkill)) ⇒ state(employed);
        (value(rId) = Emp),
remove_skill:
        state(employed), msg(← remove_skill(Emp, SkillName)) ⇒ state(employed);
        (value(rId) = Emp),
manager_deciding_change_function:
        state(employed),
        decision(change_emp_function(Emp, Function, ValidTime)) ⇒
        msg(→ new_function (Emp, Function, ValidTime)), state(employed);
        (value(function) = "manager" and value(rId) ≠ Emp and
        exists Rid(value(Rid, rId) = Emp) and Function ≠ "manager" ),
        /* This rule is only executed by the manager; the manager cannot change his
           own function; there must exist an employee associated to this Code; and the
           new function cannot be manager. */
change_function:
```

```
        state(employed),
        msg(← change_function(Emp, NewFunction, Valid_Time)) ⇒
        state(employed);
        (value(rId) = Emp),
request_change_hours_week:
        state(employed),
        msg(← mgr_change_HW(value(rId) = Emp,HWValue,ValidDate)) ⇒
        state(employed),
manager_deciding_change_hours_week:
        state(employed),
        decision(change_emp_hours_week(Emp, HWValue, ValidTime)) ⇒
        msg(→ new_hours_week (Emp, HWValue, ValidTime)), state(employed);
        (value(function) = "manager" and value(rId) ≠ Emp and
        exists Rid(value(Rid, rId) = Emp)),
        /* This rule is only executed by the manager; the manager cannot change his
           own values; there must exist the refered employee. */
change_hours_week:
        state(employed),
        msg(← change_hours_week(Emp, NewHWValue, Valid_Time)) ⇒
        state(employed);
        (value(rId) = Emp),
new_employee_choice:
        state(employed),
        decision(new_employee(Name,CPF,Addr,Gdr,Birth,Date,Sal,Func,HW)) ⇒
        msg(→ add_employee(Name,CPF,Addr,Gdr,Birth,Date,Sal, Func,HW));
        ( value(function) = "manager" and Func ≠ "manager" ),
        /* Only the manager can request a new employee, with function different from
           manager. */
new_employment_for_existing_employee:
        state(employed),
        decision(new_employment(Emp, Date, Salary, Function, HW)) ⇒
        msg(→ add_employee(Oid, Date, Salary, Function, HW)), state(employed);
        ( value(function) = "manager" and
        exists Oid ( has_class_instance (person, Oid) and
        exists Rid( has_role_instance (Oid, employee, Rid) and
        value(Rid, rId) = Emp )),
        /* Only the manager can decide a new employment for an already existent
           employee. */
new_employment_values_register:
        state(employed), msg(← emp_values (Rid, Date, Salary, Function, HW)) ⇒
        state(employed),
        /* This message is received directly by the Role Identifier. */
vacation_request:
        state(employed), decision( vacation_request(Begin, End)) ⇒
        msg(→ ask_vacations(Emp, Begin, End)), state(employed),
vacation_defered:
        state(employed), msg(← record_vacations (Emp, Beginning, Ending)) ⇒
        state(employed);
        (value(rId) = Emp),
vacation_not_defered:
        state(employed), msg(← vacations_indefered (Emp)) ⇒ state(employed);
        (value(rId) = Emp),
```

```
vacation_beginning:
      state(employed), msg(← begin_vacations(Emp)) ⇒
      state(in_vacations);
      (value(rId) = Emp),
vacation_end:
      state(in_vacations), msg(← end_vacations(Emp)) ⇒ state(employed);
      (value(rId) = Emp),
vacation_decision_by_manager:
      state(employed), msg(← confirm_vacations(Emp, Begin, End)) ⇒
      state(employed);
      ( value(function) = "manager" and
      exists Rid (value (Rid, rId) = Emp and value(Rid, function) ≠ "manager" )
      and before(Begin, End) and Begin > now),
      /* Only the manager receives this message; the manager cannot analyze his own
         vacations; a test is done to verify if the dates are in the wright order and if
         they are after the actual date. */
vacations_decided_by_manager_ok:
      state(employed), decision (vacation_decision("ok", Emp, Begin, End)) ⇒
      msg(→ ok_vacations(Emp, Begin, End)), state(employed)
      ( value(function) = "manager" ),
      /* The decided vacation period can be different from the one requested by the
         employee. */
vacations_decided_by_manager_nok:
      state(employed), decision (vacation_decision("nok",Emp,Begin,End)) ⇒
      msg(→ nok_vacations(Emp)), state(employed);
      value(function) = "manager" ),
license_request:
      state(employed), decision (license_request(Type, Begin, End)) ⇒
      msg(→ ask_license(Emp, Type, Begin, End)),
      state(employed),
license_defered:
      state(employed), msg(← record_license (Emp, Beginning, Ending)) ⇒
      state(employed);
      (value(rId) = Emp),
      /* During the license the instance is suspended. */
license_not_defered:
      state(employed), msg(← license_indefered (Emp)) ⇒
      state(employed);
      (value(rId) = Emp),
license_decision_by_manager:
      state(employed), msg(← confirm_license(Emp, Type, Begin, End)) ⇒
      state(employed);
      ( value(function) = "manager" and
      exists Rid (value(Rid, rId) = Emp and value(Rid, function) ≠ "manager )
      and before(Begin, End) and Begin > now),
      /* Only the manager receives this message; he cannot analyze his own license; a
         test is done to verify if the dates are in the wright order and if they are after
         the actual date. */
license_decided_by_manager_ok:
      state(employed), decision (license_decision("ok", Emp, Type, Begin, End)) ⇒
      msg(→ ok_license (Emp, Type, Begin, End)), state(employed);
      ( value(function) = "manager" ),
```

```
    /* The decided license period can be different from the one requested by the
       employee. */
license_decided_by_manager_nok:
    state(employed), decision (license_decision("nok", Emp, Type, Begin, End)) ⇒
    msg(→ nok_license(Emp, Type)), state(employed);
    ( value(function) = "manager" ),
asking_end_employment:
    state(employed), decision (end_emp_request(Date)) ⇒
    msg(→ ask_end_employment(Emp, EndTime)), state(employed),
asking_manager_about_end_employment:
    state(employed), msg(← analyze_emp_end(Emp, EndTime)) ⇒
    state(employed);
    (value(function) = "manager" and
    exists Rid (value(Rid, rId) = Emp and value(Rid, function) ≠ "manager" )),
    /* Only the manager can decide an other employee's end of contract; the
       manager's end of contract is analyzed by the owner. */
manager_end_employment_ok:
    state(employed), decision (end_emp_decision("ok", Emp, Date)) ⇒
    msg(→ ok_end_employment (Emp, FinalDate)), state(employed);
    ( value(function) = "manager" and value(rId) ≠ Emp ),
    /* Only the manager can execute this transition; he can give the answer about a
       requested end of employment or can decide to fire an employee; the decided
       FinalDate can be different from the one requested by the employee. */
manager_answering_end_employment_nok:
    state(employed), decision (end_emp_decision("nok", Emp, Date)) ⇒
    msg(→ nok_end_employment (Emp)), state(employed);
    ( value(function) = "manager"),
receiving_communication_end_employment:
    state(employed),
    msg(← record_end_employment (Emp, EndTime)) ⇒
    state(waiting_end_employment);
    (value(rId) = Emp),
    /* When the date of the end of the employment is reached, the role instance is
       terminated by the control process. */
/* Manager decisions: */
/* ============== */
manager_analyze_new_client_request:
    state(employed), msg(← mgr_analyze_new_client (CPF, Name, Addr,
    Phone, Gdr, Dbirth) ⇒ state(employed);
    (value(function) = "manager"),
new_client_ok:
    state(employed),
    decision(new_client_accepted (CPF, Name, Addr, Phone, Gdr, Dbirth)) ⇒
    msg(→ add_client (CPF, Name, Addr, Phone, Gdr, Dbirth)),
    state(employed);
    ( value(function) = "manager" ),
new_client_nok:
    state(employed),
    decision (new_client_nok (CPF, Name, Addr)) ⇒
    msg(→ new_client_not_accepted (CPF, Name, Addr), state(employed);
    ( value(function) = "manager" ),
```

```
manager_analyze_former_client_request:
      state(employed), msg(← mgr_analyze_former_client (Client) ⇒
      state(employed);
      (value(function) = "manager"),
new_client_ok:
      state(employed), decision(former_client_accepted (Client)) ⇒
      msg(→ add_former_client (Client)),
      state(employed);
      (value(function) = "manager"),
former_client_nok:
      state(employed), decision (former_client_nok (Client)) ⇒
      msg(→ former_client_not_accepted (Client), state(employed);
      ( value(function) = "manager" ),
bad_client_informed_by_manager:
      state(employed), decision(bad_client_record(Client, Reason)) ⇒
      msg(→ record_bad_client(Client, Reason)), state(employed);
      ( value(function) = "manager" ),
bad_client_removed_by_manager:
      state(employed), decision(remove_bad_client (Client)) ⇒
      msg(→ remove_bad_client_record(Client)), state(employed);
      ( value(function) = "manager" ),
new_film_type:
      state(employed), decision (new_film_type(Type, Descr)) ⇒
      msg(→ create_object(type, Oid)),
      msg(→ add_role(Oid, film_type, Rid),
      msg(→ include_film_type (Type, Description)),
      state(employed),
      (value(function) = "manager"),
remove_film_type:
      state(employed), decision (wrong_film_type (Type)) ⇒
      msg(→ remove_film_type(Type)), state(employed),
      (value(function) = "manager"),
new_game_type:
      state(employed), decision (new_game_type(Type, Descr)) ⇒
      msg(→ create_object(type, Oid)),
      msg(→ add_role(Oid, game_type, Rid),
      msg(→ include_game_type (Type, Description)),
      state(employed),
      (value(function) = "manager"),
remove_game_type:
      state(employed), decision (wrong_game_type (Type)) ⇒
      msg(→ remove_game_type(Type)), state(employed),
      (value(function) = "manager"),
buy_new_video:
      state(employed),
      decision (buy_new_video(Date,Furn,Film,Type,NrTapes,Dir,Act,Leg)) ⇒
      msg(→ acquire_video (Date,Furn,Film,Type,NrTapes,Dir,Act,Leg, Price)),
      state(employed);
      (value(function) = "manager"),
buy_new_game:
      state(employed),
      decision (buy_new_game(Date,Furn,Name,Type,Equip)) ⇒
```

msg(→ acquire_game (Date,Furn,Name,Type,Equip, Price)),
state(employed);
(value(function) = "manager"),

buy_material:
state(employed), decision(buy_material(Material, Furn, Price)) ⇒
msg(→ acquire_material(Material, Furn, Price)), state(employed);
(value(function) = "manager"),

actualize_losse_tape_fine_value:
state(employed), decision(change_loose_fine_value(Value)) ⇒
msg(→ tape_loose_fine(Value)), state(employed);
(value(function) = "manager"),

sell_a_tape:
state(employed), decision (sell_a_tape (Tape, Price) ⇒
msg(→ sell_tape(Tape, Price)), state(employed);
(has_role_instance(Tape, tape_status, Rid) and state(Rid) = "avalilable" and
value(function) = "manager"),
/* Only tapes that are not rented are put to sell. */

destruct_a_tape:
state(employed), decision(destruct_a_tape(Tape)) ⇒
msg(→ destruct_tape(Tape)), state(employed);
(has_role_instance(Tape, tape_status, Rid) and state(Rid) = "avalilable" and
value(function) = "manager"),
/* Only tapes that are not rented can be destructed. */

update_rental_value:
state(employed), decision (new_rental_price(Price)) ⇒
msg(→ update_rental_price (Value)), state(employed);
(value(function) = "manager"),

update_minimum_credit_quantity:
state(employed), decision (new_credit_quantity(Quantity)) ⇒
msg(→ update_min_credits(Quantity), state(employed);
(Quantity > 0 and value(function) = "manager"),

update_credit_price:
state(employed), decision (new_credit_price(Price)) ⇒
msg(→ update_credit_price(Price)), state(employed);
(value(function) = "manager"),

salary_payment_authorizatin:
state(employed), decision(payment_employee(Employee, Value)) ⇒
salary_payment_authorization (Employee, Value)), state(employed);
(value(function) = "manager"),

furnisher_payment_authorization:
state(employed), decision (payment_authorization(Creditor, Value)) ⇒
msg(→ pay(Credor, Value)), state(employed);
(value(function) = "manager"),

constraint_on_maximum_of_hours_week:
constraint(sometime past exists Id (active_role(rId) and
has_role_instance(oId, employee, Id) and active_role(Id) and Id ≠ rId ⇒
value(hours_week) + value(Id, hours_week) ≤ 40)),
/* This constraint states that if there are two employee instances for the same person, the total amount of hours worked in a week does not exceed 40 hours. */

constraint_on_existence_of_only_one_manager:
constraint(value(rId, function) = "manager" ⇒

```
        not exists Oid (has_class_instance (person,. Oid) and
        exists Rid (has_role_instance(Oid, employee, Rid) and Rid ≠ value(rId) and
        value(Rid, function) = "manager" )) ),
        /* If one employee instance with the function of manager exists, no other
           employee instance can present this function. */
>,

< Owner,
  dynamic properties = {
     (main_owner, {"Y", "N"}) },
  messages = {
     def_owner_values (Rid: intger, Name: string, Addr: string, CPF: integer,
          Gender: {"F", "M"}, Date-Birth: date)
          from OWNER_CONTROL.General_Owner_Control,
     change_owner_name (Owner: person.owner, Name: string)
          from OWNER_CONTROL.General_Owner_Control,
     change_owner_address (Owner: person.owner, Address: string)
          from OWNER_CONTROL.General_Owner_Control,
     change_owner_phone (Owner: person.owner, Old: integer, New: integer)
          from OWNER_CONTROL.General_Owner_Control,
     add_owner_phone (Owner: person.owner, Number: integer)
          from OWNER_CONTROL.General_Owner_Control,
     change_main_owner_indication(Owner: person.ownerNewIndication: {"Y", "N"})
          from OWNER_CONTROL.General_Owner_Control,
     add_employee (Name: string, CPF: integer, Addr: string, Gdr: {"F", "M"},
          Dbirth: date, Date: date, Salary: real,
          Function: {"manager", "attendance", "accountance", "support"},
          HoursWeek: hours) to EMPLOYEE_CONTROL. General_Employee_Control,
     new_salary (Mgr: person.employee, Value: real, Valid_Time: date)
          to EMPLOYEE_CONTROL. General_Employee_Control,
     new_function (Mgr: person.employee,
          Function: {"manager", "attendance", "accountance", "support"},
          ValidTime: date) to EMPLOYEE_CONTROL.General_Employee_Control,
     confirm_manager_vacations(Mgr:person.employee, Begin:date, End:date)
          from EMPLOYEE_CONTROL. General_Employee_Control,
     confirm_manager_license(Mgr:person.employee,Type: string,Begin:date,End: date)
           from EMPLOYEE_CONTROL. General_Employee_Control,
     ok_vacations (Mgr:person.employee, Begin: date, End: date)
          to EMPLOYEE_CONTROL. General_Employee_Control,
     nok_vacations (Mgr:person.employee)
          to EMPLOYEE_CONTROL. General_Employee_Control,
     ok_license (Mgr:person.employee, Type: string, Begin: date, End: date)
          to EMPLOYEE_CONTROL. General_Employee_Control,
     nok_license (Mgr:person.employee, Type: string)
          to EMPLOYEE_CONTROL. General_Employee_Control,
     analyze_emp_end(Mgr:person.employee, EndTime: date)
          from EMPLOYEE_CONTROL. General_Employee_Control,
     ok_end_employment (Mgr:person.employee, valid_time: date)
          to EMPLOYEE_CONTROL. General_Employee_Control,
     nok_end_employment (Mgr:person.employee)
          to EMPLOYEE_CONTROL. General_Employee_Control},
  decisions = {
     new_manager(Name: string, CPF: integer, Addr: string, Gdr: {"F", "M"},
          Dbirth: date, Date: date, Salary: real,
          Func: {"manager", "attendance", "accountance", "support"}),
```

```
    change_to_manager (Emp:person.employee, Valid_Time: date),
    dec_new_salary (Mgr:person.employee, Value: real, Valid_Time: date),
    vacation_decision(Decision:string, Mgr:person.employee, Beginning:date,
        Ending:date),
    license_decision (Decision: string, Mgr:person.employee, Type: string,
        Beginning:date, Ending:date) },
states = { initializing, active }
rules = {
    begin:   msg(← add_role) ⇒ state(initializing),
    initializing:
        state(initializing),
        msg(← def_owner_values (Rid, Name, Addr, CPF, Gender, Birth, Main)) ⇒
        state(active);
        (value(rId) = Rid),
    changing_name:
        state(active), msg(← change_owner_name(Owner, Name)) ⇒ state(active);
        (value(rId) = Owner),
    changing_addr:
        state(active), msg(← change_owner_address(Owner, Address)) ⇒
        state(active);
        (value(rId) = Owner),
    changing_telephone:
        state(active), msg(← change_owner_phone(Owner, Old, New)) ⇒
        state(active);
        (value(rId) = Owner),
    adding_new_telephone:
        state(active), msg(← add_owner_phone(Owner, Number)) ⇒ state(active);
        (value(rId) = Owner),
    changing_main_owner_indication:
        state(active), msg(← change_main_owner_indication(Owner,NewInd)) ⇒
        state(active),
    new_manager_choice:
        state(active),
        decision (new_manager(Name,CPF,Addr,Gdr,Birth,Date,Sal,Func)) ⇒
        msg(→ add_employee(Name,CPF,Addr,Gdr,Birth, Date,Sal,Func,HW)),
        state(active);
        (value(main_owner) = "Y" and Func = "manager"),
        /* Only the main owner can employe a new manager. */
    owner_deciding_manager_new_salary:
        state(active), decision (dec_new_salary (Emp, Value, Valid_Time)) ⇒
        msg(→ new_salary (Emp, Value, Valid_Time)), state(active);
        (value(main_owner) = "Y" and
        exists Rid (value(Rid, rId) = Emp) and value(Emp, function) = "manager" ),
        /* Only the main owner can decide a new salary for the manager. */
    owner_deciding_change_function_to_manager:
        state(active), decision (change_to_manager(Emp, ValidTime)) ⇒
        msg(→ new_function (Emp, "manager", ValidTime)), state(active);
        (value(main_owner) = "Y" and
        exists Rid (value(Rid, rId) = Emp) and value(Emp, function) ≠ "manager" ),
        /* Only the main owner can change the function of an existent employee to
           become manager; this employee has another function that manager. */
    owner_deciding_change_function_of_manager:
        state(active), (← change_emp_function(Emp, Function, ValidTime)) ⇒
```

```
    msg(→ new_function (Emp, Function, ValidTime)), state(active);
    (value(main_owner) = "Y" and exists Rid (value(Rid, rId) = Emp) and
    value(Emp, function) = "manager" and Function ≠ "manager" ),
    /* Only the main owner can change the function of the manager. */
vacation_decision_by_owner:
    state(active),
    msg(← confirm_manager_vacations(Emp, Begin, End)) ⇒
    state(active);
    ( value(main_owner) = "Y" and value(Emp, function) = "manager" and
    before(Begin, End) and Begin > now),
    /* Decisions are made only by the main owner; the owner decides only the
       vacations of the manager; the condition verifies if the dates are in the right
       order and if they are after the actual date. */
vacation_decided_ok:
    state(active), decision (vacation_decision("ok", Emp, Begin, End)) ⇒
    msg(→ ok_vacations (Emp, Begin, End)), state(active);
    ( value(main_owner) = "Y" ),
vacation_decided_nok:
    state(active), decision (vacation_decision("nok", Emp, Begin, End)) ⇒
    msg(→ nok_vacations_manager (Emp)), state(active);
    ( value(main_owner) = "Y" ),
license_decision_by_owner:
    state(active), msg(← confirm_manager_license(Emp, Type, Begin, End)) ⇒
    state(active);
    ( value(main_owner) = "Y" and value(Emp, function) = "manager" and
     before(Begin, End) and Begin > now),
    /* Decisions are made only by the main owner; the owner decides only the
       licenses of the manager; the condition veirifies if the dates are in the correct
       order and if they are after the actual date. */
license_decided_ok:
    state(deciding_license),
    decision (license_decision ("ok", Emp, Type, Begin, End)) ⇒
    msg(→ ok_license (Emp, Type, Begin, End);
    ( value(main_owner) = "Y" ),
license_decided_nok:
    state(deciding_license),
    decision (license_decision ("nok", Emp, Type, Begin, End)) ⇒
    msg(→ nok_license (Emp, Type, Begin, End);
    ( value(main_owner) = "Y" ),
asking_owner_about_end_employment:
    state(active), msg(← analyze_emp_end(Emp, EndTime)) ⇒
    state(active);
    (value(main_owner) = "Y" and value(Emp, function) = "manager" ),
    /* Only the main owner can decide the manager's end of contract. */
owner_answering_end_employment_ok:
    state(active), decision (decision_end_employment("ok", Emp, FinalDate)) ⇒
    msg(→ ok_end_employment (Emp, FinalDate)), state(active):
    ( value(main_owner) = "Y" ),
    /* The decided FinalDate can be different from the one requested by the
       manager. */
owner_answering_end_employment_nok:
    state(active), decision (decision_end_employment("nok", Emp, FinalDate) ⇒
```

```
            msg(→ nok_end_employment (emp)), state(active):
            ( value(main_owner) = "Y" ),
        owner_deciding_dismiss_manager:
            state(active), decision (decision_end_employment("ok", Emp, FinalDate)) ⇒
            msg(→ ok_end_employment (Emp, FinalDate)), state(active);`
            ( value(main_owner) = "Y" and
            exists Rid(value(Rid, rId) = Emp) and value(Emp, function) = "manager" ),
            /* This rule is executed when the owner decides to fire the manager; the owner
               can fire only the manager. */
        main_owner_constraint:
            constraint (value(main_owner) = "Y" ⇒
            not exists Oid( has_class_instance(person, Oid) and
            exists Rid (has_role_instance (Oid, owner, Rid) and Rid ≠ value(rId)
            and value(Rid, main_owner) = "Y" )) )
            /* Constraint that only one main owner may exist. */ }
>,

< Client,
  static properties = {
      (code, integer ),
  dynamic properties = {
      (inscription_date, date),
      (out_date: date),
      (allowed_persons, set_of string),
      (nr_renting_tapes, set_of tape),
      (rented_tapes, set_of integer),
      (suspention_reason, string),
      (credits, integer),
      (debit, real) },
  messages = {
      initial_values(Rid: integer, Code: integer, CPF: integer, Name: string, Addr: string,
          Telephone: set_of integer, Gender: {F, M}, D-birth: date,
          Inscription_date: date)
          from CLIENT_CONTROL.New_Client_Registration,
      person_allowed(Client: client, Name: string)
          to CLIENT_CONTROL.General_Client_Control,
      register_allowed_person(Client: client, Name: string)
          from CLIENT_CONTROL.General_Client_Control,
      bad_client (Client: client, Reason: string) from CLIENT_CONTROL.General_Client_Control,
      remove_bad_client(Client: client) from CLIENT_CONTROL.General_Client_Control,
      change_name (Client: client, Name: string)
          from CLIENT_CONTROL.General_Client_Control,
      change_address (Client: client, Address: string)
          from CLIENT_CONTROL.General_Client_Control,
      change_phone (Client: client, Old: integer, New: integer)
          from CLIENT_CONTROL.General_Client_Control,
      add_phone (Client: client, Number: integer)
          from CLIENT_CONTROL.General_Client_Control,
      add_credits (Client: client, Quantity: integer)
          from ACCOUNTANCE.Payment_Control,
      deduct_credits (Client: client, Quantity: integer) from ACCOUNTANCE.Rentals,
      decrease_rented_tape (Client: client, Tape: tape)
          from TAPE_CONTROL.Rental_Control,
      add_rented_tape (Client: client, Tape: tape)
```

```
        from TAPE_CONTROL.Rental_Control,
    add_debit(Client: client, Value: real)
        from { ACCOUNTANCE.Rentals, ACCOUNTANCE.Tape_Loose},
    deduct_debit (Client: client, Value: real) from ACCOUNTANCE.Rentals },
decision = {
    allow_person(Name: string),
    end_inscription_request },
states = { inscribing, active, bad_client_suspention, suspended },
rules = {
    begin:
        msg(← add_role) ⇒ state(inscribing),
    initialization:
        state(inscribing),
        msg(← initial_values(Rid,Code,CPF,Name,Addr,Phone,Gndr,Birth,Inscr) ⇒
        state(active),
    allow_person_to_rent:
        state(active), decision(allow_person(Name)) ⇒
        msg(→ person_allowed(Rid, Name)), state(active);
        ( not contains (allowed_persons, Name) ),
        /* The allowed person is not yet an allowed person of this client. */
    allowed_person_register:
        state(active), msg(← register_allowed_person(Rid, Name)) ⇒
        state(active);
        ( value(rId) = Rid ),
    bad_client_register:
        state(active), msg(← bad_client (Rid, Reason)) ⇒
        state(bad_client_suspension);
        ( value(rId) = Rid),
        /* This rule is executed only by the client identified by the role identifier; the
           message procedure sets the suspention reason in the property. */
    remove_bad_client:
        state(bad_client_suspension), msg(← remove_bad_client(Rid)) ⇒
        state(active),
        /* This rule is executed only by the client identified by the role identifier. */
    changing_name:
        state(active), msg(← change_name(Rid, Name)) ⇒
        state(active);
        (value(rId) = Rid),
    changing_addr:
        state(active), msg(← change_address(Rid, Address)) ⇒
        state(active);
        (value(rId) = Rid),
    changing_telephone:
        state(active), msg(← change_phone(Rid, Old, New)) ⇒
        state(active);
        (value(rId) = Rid),
    adding_new_telephone:
        state(active), msg(← add_phone(Rid, Number)) ⇒
        state(active);
        (value(rId) = Rid),
    adding_credits:
        state(active), msg(← add_credits (Rid, Quantity)) ⇒
        state(active);
```

```
        ( value(rId) = Rid),
    deducting_credits_after_rent:
        state(active), msg(← deduct_credits (Rid, Quantity)) ⇒
        state(active);
        ( value(rId) = Rid),
    renting_a_new_tape:
        state(active), msg(← add_rented_tape (Rid, Tape)) ⇒
        state(active);
        ( value(rId) = Rid ),
        /* This message includes the rented tape in the set of all rented tapes and
           increases the counter of actually total rented tapes. */
    a_rented_tape_is_returned:
        state(active), msg(← decrease_rented_tape (Rid, Tape)) ⇒
        state(active);
        ( value(rId) = Rid),
        /* This message informs that a tape that was being rented returned; the total
           amount of actually rented tapes is decreased and this tape is removed from
           the set of renting tapes. */
    adding_new_debit:
        state(active), msg(← add_debit(Rid, Value)) ⇒
        state(active);
        ( value(rId) = Rid),
    deducting_from_debit:
        state(active), msg(← deduct_debit (Rid, Value)) ⇒
        state(active);
        ( value(rId) = Rid),
    asking_end_inscription:
        state(active), decision(end_inscription_request) ⇒
        msg(→ ending_inscription(rId)),
        msg(→ suspend_role(itself)),
        state(suspended),
        /* The client himself decides to suspend his inscription; the role instance keeps
           suspended; if later the client wants to resume his inscription, the same role
           instance his resumed. */
    contraint_on_number_of_allowed_persons:
        constraint ( exists O (has_class_instance(person,O) and
        exists R (has_role_instance(O, client, R) ⇒
        not exists A,B,C,D (contains(R, allowed_persons, A) and contains(R, allowed_persons, B)
and
        and contains(R, allowed_persons, C)and contains(R, allowed_persons, D)) ) )
  > )


agent class (
  WOMAN is_a PERSON,
  inherits (Owner, Client),
  extends Employee,

  < Base_role,
    rules = {
      r1: msg(← create_object) ⇒ state(active),
      r2: state(active) ⇒ msg(→ allow_role(employee)),
```

```
      r3: state(active) ⇒ msg(→ allow_role(owner)),
      r4: state(active) ⇒ msg(→ allow_role(client)) }
>,

< Employee, /* Extension of this inherited role. */

  dynamic properties = {
      (pregnancy_license, interval(date, closed)) },
  messages = {
      ask_license(Emp: employee, Type: string, Begin: date, End: date)
          to EMPLOYEE_CONTROL. General_Employee_Control,
      record_pregnancy_license (Emp: employee, Beginning: date, Ending: date)
          from EMPLOYEE_CONTROL. General_Employee_Control,
      pregnancy_license_indefered (Emp: employee)
          from EMPLOYEE_CONTROL. General_Employee_Control },

 decision = {
      pregnancy_license_request(Begin: date, End: date) },
   states = { },
   rules = {
      pregnancy_license_request:
          state(employed), decision (pregnancy_license_request(Begin, End)) ⇒
          msg(→ ask_license(Emp, "pregnancy", Begin, End)),
          state(employed);
          ( to-days(interval(End, Begin)) ≤ 120 ),
          /* The maximum pregnancy duration is of 120 days. */
      license_defered:
          state(employed),
          msg(← record_pregnancy_license (Emp, Beginning, Ending)) ⇒
          state(employed);
          (value(rId) = Emp),
          /* During the license the instance is suspended. */
      license_not_defered:
          state(license_asked), msg(← pregnancy_license_indefered (Emp)) ⇒
          state(employed);
          (value(rId) = Emp),
      end_license:
          state(waiting_license),
          msg(← end_pregnancy_license(Emp)) ⇒ state(employed);
          (value(rId) = Emp) }
> )


agent class (
  FURNISHER,

  < Base_role,
    static properties = {
      (CGC: integer) },
    dynamic properties = {
      (name: string),
      (address, string) },
    rules = {
      r1: msg(← create_object) ⇒ state(active),
```

```
      r2: state(active) ⇒ msg(→ add_role(purchase_control)) }
  >

  < Purchase_Control,
    dynamic properties = {
        (products, set_of product) },
    messages = {
        record_purchase (Furnisher: furnisher, Product: string, Price: real)
              from {ACCOUNTANCE.Tape_Acquisition, ACCOUNTANCE.General_Acquisition} },
    states = { active },
    rules = {
        begin:
              msg(← add_role) ⇒ state(active),
        purchase_record:
              state(active), msg(← record_purchase(Furn, Product, Price) ⇒ state(active) }
  > )


agent class (
  PURCHASER,

  < Base_role,
    static properties = {
        (CGC: integer) },
    dynamic properties = {
        (name: string),
        (address, string) },
    rules = {
        r1: msg(← create_object) ⇒ state(active) }
  > )



                                  /* ------------------------- */
                                 /* RESOURCE CLASSES */
                                  /* ------------------------- */

resource class (
  TAPE,

  < Base_role,
    static properties = {
        (code, integer),
        (acquisition_date, date),
        (furnisher, FURNISHER),
        (rented_by, person.client) },
    rules = {
        r1: msg(← create_object) ⇒ state(active),
        r2: state(active) ⇒ msg(→ add_role(tape_status)) }
  >

  < Tape_Status,
    dynamic properties = {
        (rented_by, integer),
        (start_rent: date),
```

```
        (selling_price, real) },

    messages = {
            tape_rented (Client: person.client, Tape: tape)
                from TAPE_CONTROL. Rental_Control,
            returned_tape(Tape: tape) from TAPE_CONTROL.Rental_Control,
            lost_tape (Tape: tape) from TAPE_CONTROL.General_Tape_Control,
            put_to_sell (Tape: tape, Price: real) from ACCOUNTANCE.Tape_Sale,
            sold_tape (Tape: tape) from ACCOUNTANCE.Tape_Sale,
            destructed_tape(Tape: tape) from TAPE_CONTROL.General_Tape_Control },

    states = {available, rented, lost, waiting_sell, sold, destructed },

    rules = {
        begin:
            msg(← add_role) ⇒ state(available),
        rental:
            state(avalilable), msg(← tape_rented(Client, Tape)) ⇒
            state(rented);
            ( value(old) = Tape ),
            /* This transaction is executed only by the tape object identified by Tape; the
               conditions on the client are verified by the rental control process, the process
               correpondent to the message sets the Client to the rented_by property and
               the actual date to the start_rent property. */
        rental_devolution:
            state(rented), msg(← returned_tape(Tape)) ⇒
            state(avalilable);
            ( value(old) = Tape ),
        tape_lost:
            msg(← lost_tape (Tape)) ⇒ state(lost),
        tape_put_to_sell:
            state(avalilable), msg(← put_to_sell (Tape, Price)) ⇒
            state(waiting_sell),
        tape_sold:
            state(waiting_sell), msg(← sold_tape (Tape)) ⇒ state(sold),
        destructed_tape:
            state(available),  msg(← destructed_tape(Tape)) ⇒ state(destructed) }
>

resource class (
 VIDEO is_a TAPE,
 inherits Tape_Status,

 < Base_Role,
  rules = {
      r1: msg(← create_object) ⇒ state(active),
      r2: state(active) ⇒ msg(→ add_role(tape_status)),
      r3: state(active) ⇒ msg(→ allow_role(video_tape)) },
 >,

 < Video_Tape,
  static properties = {
      (nr_equal_code, integer),
```

```
        (tape_film, string),
        (film_type, set_of TYPE.Film_Type),
        (director, string),
        (main_actors, set_of string),
        (legend, {"Y", "N"}) },
    messages = {
        initial_values(Rid: integer,Code: integer, Date: date, Furn: furnisher,
              Film: string, Type: set_of type. film_type, NrTapes: integer, Dir: string,
              Actors: set_of string, Legend: {"Y", "N"})
              from TAPE_CONTROL.New_Tape_Registration},
    states = {inscribing, inscripted},
    rules = {
        begin:
              msg(← add_role) ⇒ state(inscribing),
        initialization:
              state(inscribing),
              msg(← initial_values(Rid,Code,Date,Furn,Film,Type,Dir,Actors,Legend)) ⇒
              msg(→ add_role(old, tape_status)),
              state(inscripted) }
    >

resource class (
 GAME is_a TAPE,
  inherits Tape_Status,

  < Base_role,
    rules = {
        r1: msg(← create_object) ⇒ state(active),
        r2: state(active) ⇒ msg(→ add_role(tape_status))
        r3: state(active) ⇒ msg(→ allow_role(game_tape)) },
  >,

  < Game_Tape,
    dynamic properties = {
        (game_name, string),
        (game_type, set_of TYPE.Game_Type),
        (equipment, string) },
    messages = {
        initial_values(Rid: integer, Code: integer, Date: date, Furn: furnisher,
              Game: string, Type: set_of type.game_type, Equip: string )
              from TAPE_CONTROL.New_Tape_Registration },
    states = {inscribing, inscripted},
    rules = {
        begin:
              msg(← add_role) ⇒ state(inscribing),
        initialization:
              state(inscribing),
              msg(← initial_values(Rid,Code, Date, Furn, Game, Type, Equip)) ⇒
              msg(→ add_role(old, tape_status)), state(inscripted) }
  > )


resource class (
  TYPE,
```

```
< Base_role,
  static properties = { },
  dynamic properties = {
      (Name: string),
      (Description: string) },
  rules = {
      r1: msg(← create_object) ⇒ state(active),
      r2: state(active) ⇒ msg(→ allow_role(film_type)),
      r3: state(active) ⇒ msg(→ allow_role(game_type)) }
>

< Film_Type,
/* The usual film types are drama, comedy, war, adventure, musical, cartoon. Only the
   manager can add instances to this role. */
  messages = {
      include_film_type (Type: string, Description: string) from PERSON. Employee,
      remove_film_type (Type: string ) from PERSON.Employee },
  states = { active },
  rules = {
      begin:
          msg(← add_role) ⇒ state(active),
      adding_type:
          state(active), msg(← include_film_type (Type, Description)) ⇒
          state(active);
          ( not exists Oid (has_class_instance(typ, Oid) and
          exists Rid (has_role_instance(Oid, film_type, Rid) and value(name) = Type ))),
          /* The types must all have different names. */
      removing_type:
          state(active, msg(← remove_film_type (Type)) ⇒ state(active);
          ( value(name) = Type)
>

< Game_Type,
/* The usual game types are interactif, quest, adventure, etc. Only the manager can add
   instances to this role. */
  messages = {
      include_game_type (Type: string, Description: string) from PERSON. Employee,
      remove_game_type (Type: string ) from PERSON.Employee },
  states = { active },
  rules = {
      begin:
          msg(← add_role) ⇒ state(active),
      adding_type:
          state(active), msg(← include_game_type (Type, Description)) ⇒
          state(active);
          ( not exists Oid (has_class_instance(type, Oid) and
          exists Rid (has_role_instance(Oid, game_type, Rid) and
          value(name) = Type ))),
          /* The types must all have different names. */
      removing_type:
          state(active, msg(← remove_game_type (Type)) ⇒ state(active);
          ( value(name) = Type)
```

```
>

resource class (
  CASH,

  < Base_role,
    rules = {
        r1: msg(← create_object) ⇒ state(active),
        r2: state(active) ⇒ msg(→ add_role(cash_status)),
        r3: state(active) ⇒ msg(→ allow_role(debits)),
        r4: state(active) ⇒ msg(→ allow_role(credits)) }
  >

  < Cash_Status,
    dynamic properties = {
        (total_debit, real),
        (total_credit, real),
        (in_cash, real) },
    messages = {
        received(Value: real) from ACCOUNTANCE.Rentals,
        payed(Value:real)
              from { ACCOUNTANCE.Salary, ACCOUNTANCE.Payment_Control,
              ACCOUNTANCE.Tape_Acquisition, ACCOUNTANCE.General_Acquisition },
        record_credit(Value: real)
              from { ACCOUNTANCE.Rentals, ACCOUNTANCE.Tape_Sale },
        record_debit(Value: real)
              from { ACCOUNTANCE.Tape_Acquisition, ACCOUNTANCE, General_Acquisition } },
    states = { active },
    rules = {
        begin:
              msg(← add_role) ⇒ state(active),
        payment:
              state(active), msg(← payed(V)) ⇒ state(active),
        receivement:
              state(active), msg(← received(V)) ⇒ state(active),
        record_credit:
              state(active), msg(← record_credit(Value)) ⇒ state(active),
        record_debit:
              state(active), msg(← record_debit(Value)) ⇒ state(active) }
              /* Both rules - record_credit and record_debit - just actualize the values of
                 total_credit and total_debit, and compute the resultant in_cash value. */
  >,

  < Credit,
  /* Cash credit - debits from clients or tape.purchasers. */
    dynamic properties = {
        (debtor, integer),              /* May be client or purchaser's rId. */
        (total_debit, real),
        (debit_dates, set_of date),
        (last_payment, date) },
    messages = {
        debtor_debit_values(Rid: integer, Debtor: integer, Debit: real)
              from ACCOUNTANCE.Rentals,
```

```
    deduct_debtor_debit(Debtor: integer, Payed_Value: real)
        from ACCOUNTANCE.Rentals,
    add_debtor_debit(Debtor: integer, Debit: real)
        from ACCOUNTANCE.Rentals },
  states = { active, ending },
  rules = {
    begin:
        msg(← add_role) ⇒ state(active),
    initial_values:
        state(active), msg(← debtor_debit_values(Rid, Debtor, Debit)) ⇒
        state(active),
    add_debit:
        state(active), msg(← add_debtor_debit(Client, Debtor)) ⇒ state(active);
        ( value(debtor) = Debtor ),
    deduct_debit:
        state(active), msg(← deduct_debtor_debit(Debtor, Payed_Value)) ⇒
        state(active);
        ( value(debtor) = Debtor and value(total_debit) ≠ Payed_Value ),
    quit_debit:
        state(active), msg(← deduct_debtor_debit(Debtor, Payed_Value)) ⇒
        state(ending);
        ( value(debtor) = Debtor and value(total_debit) = Payed_Value ),
    end:
        state(ending) ⇒ msg(← terminate_role(itself)) }
>,

< Debit,
/* Cash debit - payments to furnishers. */
  dynamic properties = {
    (creditor, integer),          /* Furnisher's rId. */
    (debit_value, real),
    (due_date, set_of date),
    (last_payment, date) },
  messages = {
    debit_values(Rid: integer, Creditor: integer, Debit: real)
        from {ACCOUNTANCE.Tape_Acquisition, ACCOUNTANCE.General_Acquisition},
    deduct_total_debit(Debit: integer, Payed_Value: real)
        from  ACCOUNTANCE.Payment_Control },
  states = { active, ending },
  rules = {
    begin:
        msg(← add_role) ⇒ state(active),
    initial_values:
        state(active), msg(← debit_values(Rid, Creditor, Debit)) ⇒
        state(active);
        ( value(rId) = Rid),
    deduct_debit:
        state(active), msg(← deduct_total_debit(Debit, Payed_Value)) ⇒
        state(active);
        ( value(rId) = Debit and value(total_debit) ≠ Payed_Value ),
    quit_debit:
        state(active), msg(← deduct_total_debit(Debit, Payed_Value)) ⇒
        state(ending);
        ( value(rId) = Debit and value(debit_value) = Payed_Value ),
```

```
      end:
          state(ending) ⇒ msg(← terminate_role(itself)) }
  > )


resource class (
  PRODUCTS,

  < Base_role,
    dynamic properties = {
        (name, string),
        (code, integer),
        (price, real) },
    rules = {
        r1: msg(← create_object) ⇒ state(active) }
  > )



                                  /* ------------------------ */
                                  /* PROCESS CLASSES */
                                  /* ------------------------ */


process class (
  OWNER_CONTROL,

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

  < General_Owner_Control,
    messages = {
        new_owner_values (Name: string, Addr: string, CPF: integer,
            Gender: {F, M}, Date-Birth: date, MainOwner: {"Y", "N"})
            from EXTERNAL_WORLD,
        def_owner_values (Rid: integer, Name: string, Addr: string, CPF: integer,
            Gender: {F, M}, Date-Birth: date, MainOwner: {"Y", "N"})
            to PERSON.Owner,
        owner_new_name (Owner: person.owner, Name: string)
            from EXTERNAL_WORLD,
        change_owner_name (Owner: person.owner, Name: string) to PERSON.Owner,
        owner_new_address (Owner: person.owner, Address: string)
            from EXTERNAL_WORLD,
        change_owner_address (Owner: person.owner, Address: string)
            to PERSON.Owner,
        owner_change_phone (Owner: person.owner, Old: integer, New: integer)
            from EXTERNAL_WORLD,
        change_owner_phone (Owner: person.owner, Old: integer, New: integer)
            to PERSON.Owner,
        owner_new_phone (Owner: person.owner, Old: integer, New: integer)
            from EXTERNAL_WORLD,
        add_owner_phone (Owner: person.owner, Number: integer) to PERSON.Owner,
```

```
        new_main_owner_indication (Owner: person.owner, NewIndication: {"Y", "N"})
            from EXTERNAL_WORLD,
        change_main_owner_indication (Owner: person.owner, NewIndication: {"Y", "N"})
            to PERSON.Owner,
        delete_owner(Owner: person.owner, Final_Date: date)
            from EXTERNAL_WORLD },
    states = { active },
    rules = {
        begin:   msg(← add_role) ⇒ state(active),
        new_owner_definition:
            state(active),
            msg(← new_owner_values (Name,Addr,CPF,Gender,Birth,MainInd)) ⇒
            msg(→ add_role(person.owner, Rid)),
            msg(→ def_owner_values (Rid, Name, Addr, CPF, Gender, Birth, MainInd)),
            state(active);
            /* The existence of only one main owner is controlled by a contraint rule in
               owner. */
        changing_an_owner's_name:
            state(active), msg(← owner_new_name (Owner, Name)) ⇒
            msg(→ change_owner_name (Owner, Name)), state(active),
        changing_an_owner's_address:
            state(active), msg(← owner_new_address (Owner, Address)) ⇒
            msg(→ change_owner_address (Owner, Address)), state(active),
        changing_an_owner's_telephone:
            state(active), msg(← owner_change_phone (Owner, Old, New)) ⇒
            msg(→ change_owner_phone (Owner, Old, New)), state(active),
        adding_a_new_telephone:
            state(active), msg(← owner_new_phone (Owner, Old, New)) ⇒
            msg(→ add_owner_phone (Owner, Number)), satate(active),
        changing_main_owner_indication:
            state(active), msg(← new_main_owner_indication (Owner, NewIndication)) ⇒
            msg(→ change_main_owner_indication (Owner, NewIndication)), state(active),
        deleting_owner:
            state(active), msg(← delete_owner(Owner, Final_Date)) ⇒
            terminate_role(Owner), state(active) }
>,

process class (
 EMPLOYEE_CONTROL,

 < Base_role,
   rules = {
       r1: msg(← create_object) ⇒ state(active),
       r2: state(active) ⇒ msg(→ add_role(general_employee_control)),
       r3: state(active) ⇒ msg(→ allow_role(new_employee_registration)),
       r4: state(active) ⇒ msg(→ allow_role(employee_vacation_control)),
       r5: state(active) ⇒ msg(→ allow_role(employee_license_control)),
       r6: state(active) ⇒ msg(→ allow_role(employee_end_employment_control)) }
 >,

 < General_Employee_Control,
   messages = {
```

add_employee (Name: string, CPF: integer, Addr: string, Gdr: {"F", "M"},
Dbirth: date, Date: date, Salary:real,
Function:{"manager", "attendance", "accountance", "support"}, HW: hours)
from { PERSON.Employee, PERSON. Owner },
add_employment (Oid: integer, Date: date, Salary: real,
Function: {"manager", "attendance", "accountance", "support"}, HW: hours)
from PERSON.Employee,
emp_values (Emp: person.employee, Date: date, Salary: real,
Function: {"manager", "attendance", "accountance", "support"}, HW: hours)
to PERSON.Employee,
emp_properties (Emp: person.employee, CPF:integer, Addr:string, Gdr:{"F", "M"},
Dbirth: date, Date: date, Salary: real,
Function:{"manager", "attendance", "accountance", "support"}, HW: hours)
to CONTROL.New_Employee_Registration,
request_change_name(Emp: person.employee, Name: string)
from EXTERNAL_WORLD,
change_name (Emp: person.employee, Name: string) to PERSON.Employee,
request_change_addr(Emp: person.employee, Address: string)
from EXTERNAL_WORLD,
change_address (Emp: person.employee, Address: string) to PERSON.Employee,
request_change_phone(Emp: person.employee, Old: integer, New: integer)
from EXTERNAL_WORLD,
change_phone (Emp: person.employee, Old: integer, New: integer)
to PERSON.Employee,
request_add_phone (Emp: person.employee, Number: integer)
from EXTERNAL_WORLD,
add_phone (Emp: person.employee, Number: integer) to PERSON.Employee,
new_salary (Emp: person.employee, Value: real, Valid_Time: date)
from { PERSON.Employee, PERSON.Owner},
modify_salary (Emp: person.employee, NewValue: real, Valid_Time: date)
to PERSON.Employee,
request_add_skill (Emp: person.employee, Skill: string)
from EXTERNAL_WORLD,
add_skill (Emp: person.employee, NewSkill: string) to PERSON.Employee,
request_remove_skill (Emp: person.employee, Skill: string)
from EXTERNAL_WORLD,
remove_skill (Emp: person.employee, SkillName: string) to PERSON.Employee,
new_function (Emp: person.employee,
Function: {"manager", "attendance", "accountance", "support"},
Valid_Time: date)
from {PERSON.Employee, PERSON.Owner},
change_function (Emp: person.employee,
NewFunc: {"manager", "attendance", "accountance", "support"},
Valid_Time: date)
to PERSON.Employee,
request_change_HW (Emp: person.employee, NewHWValue: hours, V_Date:date)
from EXTERNAL_WORLD,
mgr_change_HW(Emp: person.employee, HWValue: hours,ValidDate:date)
to PERSON. Employee,
new_hours_week (Emp: person.employee, HWValue: hours, ValidTime: date)
from PERSON.Employee,
change_hours_week(Emp: person.employee, NewHWValue: hours, V_Time: date)
to PERSON.Employee,
ask_vacations (Emp: person.employee, Begin: date, End: date)
from PERSON.Employee,
confirm_vacations(Emp: person.employee, Begin: date, End: date))

```
        to PERSON.Employee,
    confirm_manager_vacations (Emp: person.employee, Begin: date, End: date))
        to PERSON.Owner,
    ok_vacations(Emp: person.employee, Begin: date, End: date)
        from { PERSON.Employee, PERSON.Owner },
    nok_vacations(Emp: person.employee, Begin: date, End: date)
        from { PERSON.Employee, PERSON.Owner },
    set_vacation_properties (Rid: integer,Emp:person.employee, Start: date, End: date)
        to CONTROL.Employee_Vacation_Control,
    record_vacations (Emp: person.employee, Begin: date, End: date)
        to PERSON.Employee,
    vacations_indefered (Emp: person.employee, Beginning: date, Ending: date)
        to PERSON.Employee,
    ask_license (Emp: person.employee, Type: string, Begin: date, End: date)
        from {PERSON.Employee, WOMAN.Employee},
    confirm_license (Emp: person.employee, Type: string, Begin: date, End: date)
        to PERSON.Employee,
    confirm_manager_license (Emp: person.employee,Type:string,Begin:date,End:date)
        to PERSON.Owner,
    ok_license(Emp: person.employee, Type: string, Begin: date, End: date)
        from { PERSON.Employee, PERSON.Owner },
    nok_license(Emp: person.employee, Type: string, Begin: date, End: date)
        from { PERSON.Employee, PERSON.Owner },
    set_license_properties (Rid: integer, Emp: person.employee, Type:string,
        Start:date, End:date) to CONTROL.Employee_License_Control,
    record_license (Emp: person.employee, Begin: date, End: date) to
        PERSON.Employee,
    license_indefered(Emp: person.employee) to PERSON.Employee,
    pregnancy_license_indefered(Emp: person.employee) to WOMAN.Employee,
    record_pregnancy_license(Emp: person.employee, Beginning: date, Ending: date)
        to WOMAN.Employee,
    pregnancy_license_indefered(Emp: person.employee) to WOMAN.Employee,
    ask_end_employment(Emp: person.employee, valid_time: date)
        from PERSON.Employee,
    analyze_emp_end(Emp: person.employee, EndTime: date)
        to { PERSON.Employee, PERSON.Owner },
    ok_end_employment (Emp: person.employee, valid_time: date)
        from { PERSON.Employee, PERSON.Employee },
    nok_end_employment (Emp: person.employee, Endime: date)
        from { PERSON.Employee, PERSON.Employee },
    set_end_employment_properties (Emp: person.employee, valid_time: date)
        to CONTROL.Employee_End_Employment_Control,
    record_end_employment (Emp: person.employee, T: date) to PERSON.Employee,
    end_employment_indefered (Emp: person.employee) to PERSON.Employee },
states = {active},
 rules = {
    begin:   msg(← add_role) ⇒ state(active),
    new_employee:
        state(active),
        msg(←add_employee(Name,CPF,Addr,Gdr,Birth,Date,Sal,Func,HW)) ⇒
        msg(→ add_role(new_employee_registration, Rid)),
        msg(→emp_properties(Rid,Name,CPF,Addr,Gdr,Birth,Date,Sal, Func,HW)),
        state(active),
        /* The incomming message is sent only by the manager or by the owner. */
    new_employment_for_existing_employee:
```

```
      state(active),
      msg(←add_employement (Oid, Date, Salary, Func, HW)) ⇒
      msg(→ add_role (Oid, employee, Rid),
      msg(→ emp_values (Rid, Date, Salary, Func, HW)), state(active),
name_change:
      state(active), msg(← request_change_name(Emp, Name)) ⇒
      msg(→ change_name (Emp, Name)), state(active);
      ( exists Oid ( has_class_instance(person, Oid) and
      exists Rid (has_role_instance (Oid, employee, Rid) and
      value(Rid, rId) = Emp )) ),
      /* This employee must exist. */
address_change:
      state(active), msg(← request_change_addr(Emp, Address)) ⇒
      msg(→ change_address (Emp, Address)), state(active);
      ( exists Oid ( has_class_instance(person, Oid) and
      exists Rid(has_role_instance (Oid, employee, Rid) and
      value(Rid, rId) = Emp )) ),
phone_change:
      state(active), msg(← request_change_phone(Emp, Old, New)) ⇒
      msg(→ change_phone (Emp, Old, New)), state(active);
      ( exists Oid ( has_class_instance(person, Oid) and
      exists Rid (has_role_instance (Oid, employee, Rid) and
      value(Rid, rId=Emp)and contains(value(Rid, telephone), Old) )) ),
      /* The transition is executed when the telephone to be substituted is one of the
         defined telephones of the employee. */
adding_new_phone:
      state(active), msg(← request_add_phone (Emp, Number)) ⇒
      msg(→ add_phone (Emp, Number)), state(active);
      ( exists Oid ( has_class_instance(person, Oid) and
      exists Rid(has_role_instance (Oid, employee, Rid) and
      value(Rid, rId) = Emp )) ),
adding_skill:
      state(active), msg(← request_add_skill (Emp, Skill)) ⇒
      msg(→ add_skill (Emp, Skill)), state(active);
      ( exists Oid ( has_class_instance(person, Oid) and
      exists Rid (has_role_instance (Oid, employee, Rid) and
      value(Rid, rId) = Emp )) ),
removing_skill:
      state(active), msg(← request_remove_skill (Emp, Skill)) ⇒
      msg(→ remove_skill (Emp, Skill)), state(active);
      ( exists Oid ( has_class_instance(person, Oid) and
      exists Rid (has_role_instance (Oid, employee, Rid) and
      value(Rid, rId) = Emp ) and
      contains (value(Rid, skills), skill ) )) ),
      /* The skill to be removed must be defined as a skill of this employee. */
salary_modification:
      state(active), msg(← new_salary (Emp, Value, Valid_Time)) ⇒
      msg(→ modify_salary (Emp, Value, Valid_Time)), state(active);
      ( exists Oid ( has_class_instance(person, Oid) and
      exists Rid (has_role_instance (Oid, employee, Rid) and
      value(Rid, rId) = Emp ) ),
function_modification:
```

```
    state(active), msg(← new_function (Emp, Function, Valid_Time)) ⇒
    msg(→ change_function (Emp, Function, Valid_Time)), state(active);
    ( exists Oid ( has_class_instance(person, Oid) and
    exists Rid (has_role_instance (Oid,.employee, Rid) and
    value(Rid, rId) = Emp ) and
    value(Rid, function) ≠ Function )) ),
    /* The new function to be defined must be different from the one defined for this
       employee. */
request_of_hours_week_modification:
    state(active), msg(← request_change_HW(Emp,HWValue,ValidDate)) ⇒
    msg(→ mgr_change_HW(Emp,HWValue,ValidDate)), state(active);
    ( exists Oid ( has_class_instance(person, Oid) and
    exists Rid (has_role_instance (Oid, employee, Rid) and
    value(Rid, rId) = Emp )) ),
    /* Receiving a request of a modification of the value of hours worked per week,
       the control process transfers this request to the manager, who is the only one
       who can decide if the request is possible; the condition verifies if this Emp
       corresponds to an existent employee. */
hours_week_modifications:
    state(active), msg(← new_hours_week (Emp, HWValue, ValidTime) ⇒
    msg(→ change_hours_week(Emp, NewHWValue, Valid_Time),
    state(active);
    ( exists Oid ( has_class_instance(person, Oid) and
    exists Rid (has_role_instance (Oid, employee, Rid) and
    value(Rid, rId) = Emp )) ),
vacation_request:
    state(active), msg(← ask_vacations (Emp, Begin, End)) ⇒
    msg(→ confirm_vacations(Emp, Begin, End)),
    state(active);
    ( exists Oid ( has_class_instance(person, Oid) and
    exists Rid (has_role_instance (Oid, employee, Rid) and
    value(Rid, rId) = Emp ) and
    value(Rid, function) ≠ "manager" )) ),
    /* The message is only received by the instance identified as the manager; the
       code must correpond to an employee that is not the manager. */
manager_vacation_request:
    state(active), msg(← ask_vacations (Emp, Begin, End)) ⇒
    msg(→ confirm_manager_vacations (Emp, Begin, End)),
    state(active);
    ( exists Oid ( has_class_instance(person, Oid) and
    exists Rid (has_role_instance (Oid, employee, Rid) and
    value(Rid, rId) = Emp ) and
    value(Rid, function) = "manager" )) ),
    /* The employee must have the function of manager. */
vacation_analyzed_ok:
    state(active), msg(← ok_vacations(Emp, Begin, End)) ⇒
    msg(→ add_role(employee_vacation_control, Rid)),
    msg(→ set_vacation_properties (Rid, Emp, Beginning, Ending)),
    msg(→ record_vacations (Emp, Beginning, Ending)), state(active),
vacation_analyzed_nok:
    state(active), msg(← nok_vacations (Emp)) ⇒
```

```
        msg(→ vacations_indefered (Emp)), state(active),
license_request:
        state(active), msg(← ask_license (Emp, Type, Begin, End)) ⇒
        msg(→ confirm_license(Emp, Type, Begin, End)),
        state(active);
        ( exists Oid ( has_class_instance(person, Oid) and
        exists Rid (has_role_instance (Oid, employee, Rid) and
        value(Rid, rId) = Emp ) and
        value(Rid, function) ≠ "manager" )) ),
        /* The message is only received by the instance identified as the manager; the
           employee who asks the license cannot be the manager. */
manager_license_request:
        state(active), msg(← ask_license (Emp, Type, Begin, End)) ⇒
        msg(→ confirm_manager_license (Emp, Type, Begin, End)),
        state(active);
        ( exists Oid ( has_class_instance(person, Oid) and
        exists Rid (has_role_instance (Oid, employee, Rid) and
        value(Rid, rId) = Emp ) and
        value(Rid, function) = "manager" )) ),
        /* The employee has the function of manager. */
license_analyzed_ok:
        state(active), msg(← ok_license(Emp, Type, Begin, End)) ⇒
        msg(→ add_role(employee_license_control, Rid)),
        msg(→ set_license_properties (Rid, Emp, Type, Beginning, Ending)),
        msg(→ record_license (Emp, Type, Beginning, Ending)), state(active);
        (Type ≠ "pregnancy"),
pregnancy_license_analyzed_ok:
        state(active), msg(← ok_license(Emp, "pregnancy", Begin, End)) ⇒
        msg(→ add_role(employee_license_control, Rid)),
        msg(→ set_license_properties (Rid, Emp, "pregnancy", Beginning, Ending)),
        msg(→ record_pregnancy_license (Emp, Type, Beginning, Ending)),
        state(active),
license_analyzed_nok:
        state(active), msg(← nok_license(Emp, Type)) ⇒
        msg(→ license_indefered(Emp)), state(active),
pregnancy_license_analyzed_nok:
        state(active), msg(← nok_license(Emp, Type)) ⇒
        msg(→ pregnancy_license_indefered(Emp)), state(active);
        ( Type = "pregnancy" ),
end_employment_request:
        state(active), msg(← ask_end_employment(Emp, EndTime)) ⇒
        msg(→ analyze_emp_end(Emp, EndTime)), state(active),
        /* This rule will be received only by the manager in the case the function is not
           manager; when the function is correpondent to a manager, the rule is
           received by the main owner. */
ok_end_employment_from_manager_or_owner :
        state(active), msg(← ok_end_employment(Emp, End)) ⇒
        msg(→ add_role(employee_end_employment_control, Rid)),
        msg(→ set_end_employment_properties (Rid, Emp, End)),
        msg(→ record_end_employment (Emp, T)),
        state(active),
```

```
    nok_end_employment_from_manager_or_owner :
        state(active), msg(← nok_end_employment (Emp)) ⇒
        msg(→ end_employment_indefered (Code)), state(active) }
>,

< New_Employee_Registration,
  static properties = {
    (name, string),
    (cpf, integer),
    (address, string),
    (gender, {F.M}),
    (date_birth, date),
    (admission_date:date),
    (salary, real),
    (function: {"manager", "attendance", "accountance", "support"})
    (hours_week, hours) },
  messages = {
    emp_properties (Rid: integer, Name: string, CPF: integer, Addr: string,
        Gdr: {"F", "M"}, Dbirth: date, Date: date, Salary: real,
        Function:{"manager", "attendance", "accountance", "support"}, HW: hours )
        from CONTROL.General_Employee_Control,
    get_next_employee_code(Rid: integer) to CODE_GENERATION.Employee_Code,
    employee_code (Rid: integer, Code: integer)
        from CODE_GENERATION.Employee_Code,
    initial_values(Rid: integer, Code: integer, CPF: integer, Name: string, Addr: string,
        Gender: {F, M}, D-birth: date, AdmissionDate: date, Salary: real,
        Function: {"manager", "attendance", "accountance", "support"}, HW: hours )
        to PERSON.Employee,
  states = { active, wating_code, ending },
  rules = {
    begin:   msg(← add_role) ⇒ state(active),
    values:
        state(active),
        msg(← emp_properties(rId,Name,CPF,Addr,Gdr,Birth,Date,Sal,Func,HW)) ⇒
        msg(→ get_next_employee_code(rId)), state(waiting_code),
        /* The rId reffers to the identifier of this role instance and is used here to
           idebtify which of the instances is waiting for the code. */
    initializing_employee_woman:
        state(waiting_code), msg(← employee_code (rId, Code)) ⇒
        msg(→ create_object(woman, Oid)),
        msg(→ add_role(Oid, employee, Rid)),
        msg(→ initial_values(Rid,Code, cpf, name, address, gender, date_birth,
            admission_date, salary, function)), state(ending);
        ( gender = "F" ),
    initializing_employee_man:
        state(waiting_code), msg(← employee_code (rId, Code)) ⇒
        msg(→ create_object(person, Oid)),
        msg(→ add_role(Oid, employee, Rid)),
        msg(→ initial_values(Rid,Code, cpf, name, address, gender, date_birth,
            admission_date, salary, function)), state(ending);
        ( gender = "M" ),

    ending:
```

```
            state(ending) ⇒ msg(→ terminate_role(itself)) }
>,

< Employee_Vacation_Control,
  static properties = {
      (emp: person.employee),
      (start: date),
      (end: date) },
  messages = {
      set_vacation_properties (Rid: integer, Emp: person.employee, Start:date,End: date)
            from CONTROL.General_Employee_Control,
      begin_vacations(Emp: person.employee) to PERSON.Employee,
      end_vacations(Emp: person.employee) to PERSON.Employee,
      control_vacation_time (Target_date: date))
            to CLOCK.Clock_Control
      vacation_time_reached from CLOCK.Time },
  states = {active, waiting_beginning, waiting_ending, ending },
  rules = {
      begin:    msg(← add_role) ⇒ state(active),
      start_begin:
            state(active), msg(← set_vacation_properties (Emp, Start, End)) ⇒
            msg(→ control_vacation_time (start)), state(waiting_beginning),
      start_end:
            state(waiting_beginning), msg(← vacation_time_reached) ⇒
            msg(→ begin_vacations(emp)),
            msg(→ control_vacation_time (end)), state(waiting_end),
      ending:
            state(waiting_end), msg(← vacation_time_reached) ⇒
            msg(→ end_vacations(emp)), state(ending),
      end:
            state(ending) ⇒ msg(terminate_role(itself)) }
>,

< Employee_License_Control,
  static properties = {
      (emp: person.employee),
      (type: string),
      (start: date),
      (end: date) },
  messages = {
      set_license_properties(Rid: integer,Emp: person.employee, Type: string,
            Start: date,End: date) from CONTROL.General_Employee_Control,
      control_license_time (Target_date: date))
            to CLOCK.Clock_Control
      license_time_reached from CLOCK.Time }
  states = {active, waiting_beginning, waiting_end, ending },
  rules = {
      begin:    msg(← add_role) ⇒ state(active),
      start_begin:
            state(active), msg(← set_license_properties (Rid, Emp, Type, Start, End)) ⇒
            msg(→ control_license_time (Start), state(waiting_beginning),
      start_end:
            state(waiting_beginning), msg(← license_time_reached) ⇒
            msg(→ suspend_role(emp)),
```

```
            msg(→ control_license_time (value(end))), state(waiting_end),
        ending:
            state(waiting_end), msg(← license_time_reached) ⇒
            msg(→ resume_role(emp)),
            state(ending),
        end:
            state(ending) ⇒ msg(terminate_role(itself)) }
    >,

    < Employee_End_Employment_Control,
      static properties = {
        (emp: person.employee),
        (end: date) },
      messages = {
        set_end_employment_properties (Rid: integer, Emp: person.employee, valid_time: date)
            from CONTROL.General_Employee_Control,
        end_employment (Emp: person.employee) to PERSON.Employee,
        control_end_emp_time (Target_date: date))
            to CLOCK.Clock_Control
        end_emp_time_reached from CLOCK.Time }
      states = {active, waiting_ending, ending },
      rules = {
        begin:   msg(← add_role) ⇒ state(active),
        start_clock:
            state(active),
            msg(← set_end_employment_properties (Rid, Emp, End)) ⇒
            msg(→ control_end_emp_time (End), state(waiting_ending),
        ending:
            state(waiting_ending), msg(← end_emp_time_reached) ⇒
            msg(→ terminate_role(emp)),
            state(ending),
        end:
            state(ending) ⇒ msg(terminate_role(itself)) }
    > )


process class (
    CLIENT_CONTROL,

    < Base_role,
      rules = {
        r1: msg(← create_object) ⇒ state(active),
        r2: state(active) ⇒ msg(add_role(general_client_control)),
        r3: state(active) ⇒ msg(→ allow_role(new_client_registration)) },
    >,

    < General_Client_Control,
      messages = {
        inscription_request(CPF: integer, Name: string, Address: string,
            Telephone: set_of integer, Gender: {"F", "M"}, D-birth: date)
            from EXTERNAL_WORLD,
        mgr_analyze_new_client (Client:person.client, Name: string, Addr: string,
            Phone: set_of integer, Gdr: {"F","M"}, Dbirth: date)
            to PERSON. Employee,
```

```
    add_client (CPF: integer, Name: string, Addr: string, Phone: set_of integer,
        Gdr: {"F","M"}, Dbirth: date)
        from PERSON. Employee,
    new_client_nok (CPF: integer, Name: string, Addr: string)
        from PERSON. Employee,
    inscription_denied(CPF: integer, Name: string, Addr: string)
        to EXTERNAL WORLD,
    client_properties(Rid: integer, CPF: integer, Name: string, Addr: string,
        Phone: set_of integer, Gdr: {"F","M"}, Dbirth: date)
        to New_Client_Registration,
    mgr_analyze_former_client (Client:person.client) to PERSON. Employee,
    add_former_client (Client:person.client) from PERSON. Employee,
    former_client_not_accepted (Client:person.client) from PERSON. Employee,
    former_inscription_denied(Client:person.client) to EXTERNAL_WORLD,
    record_bad_client (Client:person.client, Reason: string) from  PERSON.Employee,
    remove_bad_client_record (Client: person.client) from  PERSON.Employee,
    bad_client (Client:person.client, Reason: string) to  PERSON.Client,
    remove_bad_client(Client: client) to  PERSON.Client,
    person_allowed(Client:person.client, Name: string) from PERSON.Client,
    register_allowed_person(Client:person.client, Name: string) to PERSON.Client,
    request_change_name(Client:person.client, Name: string)
        from EXTERNAL_WORLD,
    change_name (Client:person.client, Name: string) to PERSON.Client,
    request_change_addr(Client:person.client, Address: string)
        from EXTERNAL_WORLD,
    change_address (Client:person.client, Address: string) to PERSON.Client,
    request_change_phone(Client:person.client, Old: integer, New: integer)
        from EXTERNAL_WORLD,
    change_phone (Client:person.client, Old: integer, New: integer) to PERSON.Client,
    request_add_phone (Client:person.client, Number: integer)
        from EXTERNAL_WORLD,
    add_phone (Client:person.client, Number: integer) to PERSON.Client },
states = { active },
rules = {
    begin:
        msg(← add_role) ⇒ state(active),
    new_client_request:
        state(active),
        msg(← inscription_request(CPF, Name, Address, Phone, Gender, Dbirth)) ⇒
        msg(→ mgr_analyze_new_client (CPF, Name, Addr, Phone, Gdr, Dbirth)),
        state(active);
        ( not exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and
        value(Rid, cpf) = CPF )) ),
        /* There is no other client with the same CPF. */
    new_client_ok:
        state(active), msg(← add_client (CPF, Name, Addr, Phone, Gdr, Dbirth)) ⇒
        msg(→ add_role(new_client_registration, Rid),
        msg(→ client_properties(Rid, CPF, Name, Addr, Phone, Gdr, Dbirth)),
        state(active),
    new_client_nok:
        state(active), msg(← new_client_nok (CPF, Name, Addr)) ⇒
        msg(→ inscription_denied(CPF, Name, Addr)), state(active),
    former_client_request:
```

```
        state(active),
        msg(← inscription_request(Client, Name, Address, Phone, Gender, Dbirth)) ⇒
        msg(→ mgr_analyze_former_client (Client)),
        state(active);
        ( exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and
        value(Rid, rId) = Client )) ),
former_client_ok:
        state(active), msg(← add_former_client (Client)) ⇒
        msg(→ resume_role(Client)),
        state(active),
        /* When the manager agrees with the re-inscription of a former client, the
           former client role instance is resumed. */
former_client_nok:
        state(active), msg(← former_client_not_accepted (Client)) ⇒
        msg(→ former_inscription_denied(Client)), state(active),
bad_client_record:
        state(active), msg(← record_bad_client (Client, Reason)) ⇒
        msg(→ bad_client (Client, Reason)), state(active),
bad_client_record_remove:
        state(active), msg(← remove_bad_client_record (Client)) ⇒
        msg(→ remove_bad_client (Client)), state(active),
allowed_person_request:
        state(active), msg(← person_allowed(Client, Name)) ⇒
        msg(→ register_allowed_person(Client, Name)), state(active),
name_change:
        state(active), msg(← request_change_name(Client, Name)) ⇒
        msg(→ change_name (Client, Name)), state(active);
        ( exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and
        value(Rid, rId) = Client )) ),
        /* The client must exist. */
address_change:
        state(active), msg(← request_change_addr(Client, Address)) ⇒
        msg(→ change_address (Client, Address)), state(active);
        ( exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and
        value(Rid, rId) = Client )) ),
        /* The client must exist. */
phone_change:
        state(active), msg(← request_change_phone(Client, Old, New)) ⇒
        msg(→ change_phone (Client, Old, New)), state(active);
        ( exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and
        value(Rid, rId) = Client and contains(value(Rid, telephone), Old) )) ),
        /* The transition is executed when the telephone to be substituted is one of the
           defined telephones of this client. */
adding_new_phone:
        state(active), msg(← request_add_phone (Client, Number)) ⇒
        msg(→ add_phone (Client, Number)), state(active);
        ( exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and
```

```
            value(Rid, rId) = Client )) ) }
            /* The client must exist. */
>

< New_Client_Registration,
  static properties = {
      (cpf, integer),
      (name, string),
      (address, string),
      (phone, set_of integer),
      (gender, {"F", "M"}),
      (date_birth, date) },
  messages = {
      client_properties (Rid: integer, CPF: integer, Name: string, Addr: string,
            Phone: set_of integer,Gdr: {"F","M"}, Dbirth: date)
            from CLIENT_CONTROL.General_Client_Control,
      get_next_client_code (Rid: integer) to CODE_GENERATION.Client_Code,
      client_code(Rid: integer, Code: integer) from CODE_GENERATION.Client_Code,
      initial_values(Rid, integer, Code: integer, CPF: integer, Name: string, Addr: string,
            Phone: set_of integer, Gdr: {"F","M"}, Dbirth: date) to PERSON.Client },
  states = {active, waiting_for_code, ending }.,
  rules = {
      begin:   msg(← add_role) ⇒ state(active),
      values:
            state(active),
            msg(← client_properties(rId, CPF, Name, Addr, Phone, Birth, Gdr)) ⇒
            msg(→ get_next_client_code(rId)), state(waiting_code),
            /* The rId reffers to the identifier of this role instance and is used here to
               identify which of the instances is waiting for the code. */
      initializing_client:
            state(waiting_code), msg(← client_code (rId, Code)) ⇒
            msg(→ create_object(person, Oid)),
            msg(→ add_role(Oid, client, Rid)),
            msg(→ initial_values(Rid,Code,cpf,name,address,phone,date_birth, gender,
               now)),
            state(ending),
            /* The inscription date is the actual date, furnished by the special function
               "now". */
      ending:
            state(ending) ⇒ msg(→ terminate_role(itself)) }
>,


process class (
  TAPE_CONTROL,

  < Base_role,
    static properties = { },
    dynamic properties = { },
    rules = {
      r1: msg(← create_object) ⇒ state(active),
      r2: state(active) ⇒ msg(→ add_role(general_tape_control)),
      r3: state(active) ⇒ msg(→ allow_role(new_tape_registration)),
```

```
    r4: state(active) ⇒ msg(→ add_role(Rental_Control)) },
>,

< General_Tape_Control,
  messages = {
    inscribe_video (Date: date, Furn: furnisher, Film: string,
        Type:set_of type.film_type, NrTapes, Director: string, Actors: set_of string,
        Legend: {"Y", "N"}) from ACCOUNTANCE.Tape_Acquisition,
    inscribe_game (Date: date, Furn: furnisher, Name: string,
        Type: set_of type.game_type, Equipment: string)
        from ACCOUNTANCE.Tape_Acquisition
    new_video_values (Rid: integer, Date: date, Furn: furnisher, Film: string,
        Type:set_of type.film_type, NrTapes, Director: string, Actors: set_of string,
        Legend: {"Y", "N"}) to New_Tape_Registration,
    new_game_values (Rid: integer, Date: date, Furn: furnisher,
        Type:set_of type.film_type, Equip: string) to New_Tape_Registration,
    inf_client_lost_tape(Tape: tape, client: person.client) from EXTERNAL_WORLD,
    inf_lost_tape(Tape: tape) from EXTERNAL_WORLD,
    credit_lost_tape(Client: person.client) to ACCOUNTANCE.Tape_Loose,
    lost_tape (Tape: tape) to TAPE.Tape_Status,
    destruct_tape (Tape: tape) from PERSON.Employee,
    destructed_tape(Tape: tape) to TAPE.Tape_Status },
  states = { active },
  rules = {
    begin:
        msg(← add_role) ⇒ state(active),
    new_video:
        state(active),
        msg(← inscribe_video (Date,Furn,Film,Type,NrT,Director,Acts,Leg)) ⇒
        msg(→ add_role(new_tape_registration, Rid)),
        msg(→ new_video_values (Rid, Date,Furn,Film,Type,NrT,Director,Acts,Leg)),
        state(active);
    new_game:
        state(active),
        msg(← inscribe_game (Date,Furn,Type,Equip)) ⇒
        msg(→ add_role(new_tape_registration, Rid)),
        msg(→ new_game_values (Rid, Date,Furn,Type,Equip)),
        state(active);
    tape_loss_within_shop:
        state(active), msg(← inf_lost_tape(Tape)) ⇒
        msg(→ lost_tape (Tape)), state(active),
    tape_loss_by_client:
        state(active), msg(← inf_client_lost_tape(Tape, Client)) ⇒
        msg(→ credit_lost_tape(Client)),
        msg(→ lost_tape (Tape)), state(active),
    manager_decides_destruct_tape:
        state(active), msg(← destruct_tape(Tape)) ⇒
        msg(→ destructed_tape(Tape)), state(active) }
>

< New_Tape_Registration,
  static properties = {
    ( Date, date ),
    ( Furn, furnisher ),
```

```
    ( Name: string ),
    ( FilmType: set_of type.film_type ),
    ( GameType: set_of type.film_type ),
    ( NrTapes, integer ),
    ( Dir, string ),
    ( Acts, set_of string ),
    ( Leg, {"Y", "N"} ),
    (Equip, string) },
  messages = {
    new_video_values (Date: date, Furn: furnisher, Film: string,
        Type: set_of type.film_type, NrTapes, Director: string, Actors: set_of string,
        Legend: {"Y", "N"}) from General_Tape_Control,
    new_game_values (Date: date, Furn: furnisher, Name: string,
        Type:set_of type.game_type, Equip: string) from General_Tape_Control,

    get_next_tape_code (rId: integer) to CODE_GENERATION.Tape_Code,
    tape_code (rId: integer, Code: integer) from CODE_GENERATION.Tape_Code,
    initial_values(Rid: integer,Code: integer, Date: date, Furn: furnisher,
        Film: string, Type:set_of type.film_Type, NrTapes: integer, Dir: string,
        Actors: set_of string, Legend: {"Y", "N"}) to VIDEO.Video_Tape },
  states = {active, waiting_video_code, waiting_game_code, ending }.,
  rules = {
    begin:   msg(← add_role) ⇒ state(active),
    video_values:
        state(active),
        msg(← new_video_values(Date,Furn,Name,Type,NrTap,Dir,Acts,Legend)) ⇒
        msg(→ get_next_tape_code(rId)), state(waiting_video_code),
        /* The rId reffers to the identifier of this role instance and is used here to
           identify which of the instances is waiting for the code. */
    game_values:
        state(active),
        msg(← new_game_values (Date,Furn,Name,Type,Equip)) ⇒
        msg(→ get_next_tape_code(rId)), state(waiting_game_code),
        /* The rId reffers to the identifier of this role instance and is used here to
           identify which of the instances is waiting for the code. */
    initializing_video_tape:
        state(waiting_video_code), msg(← tape_code (rId, Code)) ⇒
        msg(→ create_object(video, Oid)),
        msg(→ add_role(Oid, video_tape, Rid)),
        msg(→ initial_values(Rid,Code,Date,Furn,Name,FilmType,NrTapes,Dir,Acts,
        Leg)),
        state(ending),
    initializing_game_tape:
        state(waiting_game_code), msg(← tape_code (rId, Code)) ⇒
        msg(→ create_object(game, Oid)),
        msg(→ add_role(Oid, game_tape, Rid)),
        msg(→ initial_values(Rid,Code,Date,Furn,Name,GameType,Equip)),
        state(ending),
    ending:
        state(ending) ⇒ msg(→ terminate_role(itself)) }
>,

< Rental_Control,
```

```
static properties = { },
dynamic properities = { },
messages = {
    rental_request (Name: string, Client: person.client, Tape: tape)
        from EXTERNAL_WORLD,
    tape_rented (Client: person.client, Tape: tape) to TAPE.Tape_Status,
    add_rented_tape (Client: person.client, Tape: tape) to PERSON. Client,
    rental_denied(Name: string) to EXTERNAL_WORLD },
    inf_tape_devolution(Tape: tape, Client: person.client, Days: days)
        from EXTERNAL_WORLD,
    decrease_rented_tape(Client: person.client, Tape: tape) to PERSON.Client,
    returned_tape(Tape: tape) to TAPE.Tape_Status,
    rental_value(Client: person.client, Days: days) to ACCOUNTANCE.Rentals },
states = { active },
rules = {
    begin:
        msg(→ add_role) ⇒ state(active),
    rental_of_a_tape:
        state(active), msg(← rental_request (Name, Client, Tape)) ⇒
        msg(→ tape_rented (Client, Tape)),
        msg(→ add_rented_tape (Client, Tape)), state(active);
        (exists O (has_class_instance(tape, O) and value(O, oId) = Tape ) and
        exists R (has_role_instance(Tape, tape_status, R) and
        state(R) = "available" ) and
        exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and value(Rid, rId) = Client and
        state(Rid) = "active" and
        contains (value(Client, allowed_persons), Name) and
        value(Client, nr_renting_tapes) < 5 and
        not exists OI (has_class_instance(tape, OI) and
        exists RI (has_role_instance(OI, tape_status, RI) and
        value(RI) = "rented" and value(RI, rented_by) = Client and
        value(RI, start_rent) < now - 60 ))))),
        /* The tape can be rented if (i) the requested tape exists and is avalilable; (ii) the
           requesting client correponds to good client and the person who is requesting
           the rental is allowed by this client to rent; (iii) this client is not renting more
           than 4 tapes; and (iv) this client is not renting another tape for more that 60
           days. */
    not_allowing_renting:
        state(active), msg(← rental_request (Name, Client, Tape)) ⇒
        msg(→ rental_denied(Name)), state(active);
        ( not (exists O (has_class_instance(tape, O) and value(O, oId) = Tape ) and
        exists R (has_role_instance(Tape, tape_status, R) and
        state(R) = "available" ) and
        exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and value(Rid, rId) = Client and
        state(Rid) = "active" and
        contains (value(Client, allowed_persons), Name) and
        value(Client, nr_renting_tapes) < 5 and
        not exists OI (has_class_instance(tape, OI) and
        exists RI (has_role_instance(OI, tape_status, RI) and
        value(RI) = "rented" and value(RI, rented_by) = Client and
        value(RI, start_rent) < now - 60 )))))),
```

```
        /* If any one of the conditions stated on the preceeding rule is not satisfied, the
           rental will not be allowed. */
      tape_devolution:
          state(active), msg((← inf_tape_devolution(Tape, Client, Days)) ⇒
          msg(→ decrease_rented_tape(Client, Tape)),
          msg(→ returned_tape(Tape)),
          msg(→ rental_value(Client, Days)), state(active);
          ( exists O (has_class_instance(person, O) and
          exists R (has_role_instance(O, client, R) and value(R, rId) = ClientCode and
          contains(value(R, rented_tapes), TapeCode) )) and Days > 0 ) }
          /* The condition tests if the informations are correct: if the client exists and if
             the tape is being rented by this client; the total number of renting days must
             be greater than zero. */
>

process class (
 CODE_GENERATION,

 < Base_role,
   static porperties = { },
   dynamic properties = { },
   rules = {
      r1: msg(← create_object) ⇒ state(active),
      r2: state(active) ⇒ msg(→ add_role(employee_code), state(active),
      r3: state(active) ⇒ msg(→ add_role(tape_code), state(active),
      r4: state(active) ⇒ msg(→ add_role(client_code), state(active),

 < Employee_Code,
   dynamic properties = {
      (code, integer) },
   messages = {
      get_next_employee_code (Rid: integer)
           from CONTROL.New_Employee_Registration,
      employee_code (Rid: integer, Code: integer)
           to CONTROL.New_Employee_Registration },
   states = {waiting },
   rules = {
      r1: msg(← add_role) ⇒ state(waiting),
      r2: state(waiting), msg(← get_next_employee_code(Rid)) ⇒
           msg(→ employee_code (Rid, Code)), state(waiting);
           ( not exists Oid ( has_class_instance(person, Oid) and
           exists Rid (has_role_instance(Oid, employee, Rid) and
           value(Rid, code) = Code )) ),
          /* The new code is differente from all the other existing employee codes; the
             code calculus is left to the implementation. */
 >,

< Tape_Code,
   dynamic properties = {
      (code, integer) },
   messages = {
      get_next_tape_code (Rid: integer) from CONTROL.New_Tape_Registration,
      tape_code (Rid: integer, Code: integer) to CONTROL.New_Tape_Registration },
```

```
  states = {waiting },
  rules = {
      r1: msg(← add_role) ⇒ state(waiting),
      r2: state(waiting), msg(← get_next_tape_code) ⇒
            msg(→ tape_code (Code)), state(waiting);
            ( not exists Oid ( has_class_instance(video, Oid) and
            exists Rid (has_role_instance(Oid, video_tape, Rid)
            and value(Rid, code) = Code )) and
            not exists O ( has_class_instance(game, O) and
            exists R (has_role_instance(O, game_tape, R)
            and value(R, code) = Code ))),
           /* The new code is differente from all the other existing tape and game codes;
              the code calculus is left to the implementation. */
>,

< Client_Code,
  dynamic properties = {
      (code, integer) },
  messages = {
      get_next_client_code (Rid: integer) from CONTROL.New_Client_Registration,
      client_code (Rid: integer, Code: integer) to CONTROL.New_Client_Registration },
  states = {waiting },
  rules = {
      r1: msg(← add_role) ⇒ state(waiting),
      r2: state(waiting), msg(← get_next_client_code) ⇒
            msg(→ client_code (Code)), state(waiting);
            ( not exists Oid ( has_class_instance(person, Oid) and
            exists Rid (has_role_instance(Oid, client, Rid) and value(Rid, code) = Code ))),
           /* The new code is differente from all the other existing client codes; the code
              claculus is left to the implementation. */
>,

process class (
 CLOCK,

 < Base_role,
  rules = {
      r1: msg(← create_object) ⇒ state(active),
      r2: state(active) ⇒ msg(→ add_role(clock_control)),
      r3: state(active) ⇒ msg(→ allow_role(event)) }
 >,

 < Clock_Control,
  /* the clock control initializes the clock for each requested date control, passing to the clock the role
      identifier of the role that shall receive the time notice */
  dynamic properties = {
      (target_role_identifier, integer),
      (target_date, date) },
  state = {waiting, initializing_vacation_clock, initializing_license_clock}
  messages = {
      control_vacation_time (Target_date: date))
            from CONTROL.Employee_Vacation_Control,
      control_license_time (Target_date: date))
            from CONTROL.Employee_License_Control,
```

```
    control_end_emp_time (Target_date: date))
        from CONTROL.Employee_End_Employment_Control,
    set_values(Rid: integer, Process:string, Target_date:date)) to Event },
  rules = {
    begin:
        msg(← add_role) ⇒ state(waiting),
    start_vacation:
        state(waiting), msg(← control_vacation_time (Code, Date) ⇒
        msg(→ add_role(old, event, Rid)),
        msg(→ set_values(Rid, "vacation", Target_date)),
        state(waiting),
    start_license:
        state(waiting), msg(← control_license_time (Code, Date) ⇒
        msg(→ add_role(old, event, Rid)),
        msg(→ set_values(Rid, "license", Target_date)),
        state(waiting),
    start_end_employment:
        state(waiting), msg(← control_end_emp_time (Code, Date) ⇒
        msg(→ add_role(old, event, Rid)),
        msg(→ set_values(Rid, "end_emp", Target_date)),
        state(waiting) }
>,

< Event,
  /* each instance of the role clock controls a specific date, sending a message to
     general_employee_control the moment this date is reached */
  static properties = {
     (process, { "vacation", "license", "end_emp" }),
     (date_to_control: date) },
  mensagens = {
     set_values(Process: string, Target_date: date))
        from clock_control },
     vacation_time_reached to CONTROL.employee_vacation_control,
     license_time_reached to CONTROL.employee_license_control },
     end_emp_time_reached to CONTROL.employee_end_employment_control,
  states = { active, controling, ending },
  rules = {
     início:
        msg(← add_role) ⇒ state(active),
     values:
        state(active), msg(← set_values(Process, Target_date)) ⇒
        state(controling),
     contr1:
        state(controling) ⇒
        msg(→ vacation_time_reached), state(ending);
        ( value(date_to_control) = now and value(process) = "vacation" ),
        /* This rule will be executed only when the date to control is reached; the
           message will be send to the vacation control process. /*
     contr2:
        state(controling) ⇒
        msg(→ license_time_reached), state(ending);
        ( value(date_to_control) = now and value(process) = "license" ),
```

```
            /* This rule will be executed only when the date to control is reached; the
               message will be send to the license control process. /*
        contr3:
            state(controling) ⇒
            msg(→ end_emp_time_reached), state(ending);
            ( value(date_to_control) = now and value(process) = "end_emp" ),
            /* This rule will be executed only when the date to control is reached; the
               message will be send to the end employment control process. /*
        end:
            state(ending) ⇒ msg(← terminate_role(itself)) }
>


process class (
  ACCOUNTANCE,

  < Base_role,
    rules = {
        r1: msg(← create_object) ⇒ state(active),
        r2: state(active) ⇒ msg(→ add_role(rentals)),
        r3: state(active) ⇒ msg(→ add_role(tape_acquisition)),
        r4: state(active) ⇒ msg(→ add_role(general_acquisition)),
        r5: state(active) ⇒ msg(→ add_role(tape_loose)),
        r6: state(active) ⇒ msg(→ add_role(tape_sale)),
        r7: state(active) ⇒ msg(→ add_role(tape_loose)),
        r8: state(active) ⇒ msg(→ add_role(salary)),
        r9: state(active) ⇒ msg(→ add_role(Payment_Control)) }
  >,

  < Rentals,
    dynamic properties = {
        (day_rental, real) },
    messages = {
        update_rental_price (Price) from PERSON.Employee,
        rental_value(Client: person.client, Days: days)
            from TAPE_CONTROL_Rental_Control,
        deduct_credits (Client: person.client, Quantity: integer) to PERSON. Client,
        record_credit(Value: real) to CASH.Cash_Status,
        add_debit(Client: person.client, Value: real) to PERSON.Client,
        debtor_debit_values(Rid: integer, Client: person.client, Value: real)
            to CASH.Credit,
        add_debtor_debit(Client: person.client, Debit: real) to CASH.Credit,
        deduct_debtor_debit(Client: person.client, Debit: real) to CASH.Credit,
        deduct_debit (Client: person.client, Value: real) to PERSON.Client,
        payment(Client, Value) from EXTERNAL_WORLD,
        received (Value: real) to CASH.Cash_Status },
    states = { active },
    rules = {
        begin:
            msg(← add_role) ⇒ state(active),
        rental_value_update:
            state(active), msg(← update_rental_price (Price)) ⇒ state(active),
        rental_payed_by_credits:
```

```
        state(active), msg(← rental_value(Client, Days)) ⇒
        msg(→ deduct_credits (Client, Days)), state(active);
        ( exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and
        value(Rid, credits) ≥ Days )) ),
        This rule is used when the client has enough credits to pay for the rent. */
    rental_record1:
        state(active), msg(← rental_value(Client, Days)) ⇒
        msg(→ record_credit(Value)), msg(→ add_debit(Client, Value)),
        msg(→ add_role(cash.credit(Rid)),
        msg(→ debtor_debit_values(Rid, Client, Value)), state(active);
        ( exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and
        value(Rid, credits) < Days )) and
        not exists Oid (has_class_instance(cash, Oid) and
        exists R (has_role_instance(Oid, credit, R) and value(R, debtor) = Client )) ),
        /* This rule applies when this client does not have any debit in the shop. */
    rental_record2:
        state(active), msg(← rental_value(Client, Days)) ⇒
        msg(→ record_credit(Value)), msg(→ add_debit(Client, Value)),
        msg(→ add_debtor_debit(Client, Debit)),
        state(active);
        ( exists Oid (has_class_instance(person, Oid) and
        exists Rid (has_role_instance(Oid, client, Rid) and
        value(Rid, credits) < Days )) and
        exists Oid (has_class_instance(cash, Oid) and
        exists Rid (has_role_instance(Oid, cash, Rid) and
        value(Rid, debtor = Client )) ) }
        /* This rule applies when this client present other debits in the shop; this debit
           will be added to the total debit of this client. */
>

< Tape_Acquisition,
  messages = {
    acquire_video (Date: date, Furn: furnisher, Film: string,
        Type:set_of type.film_type, NrTapes: integer, Director: string,
        Actors: set_of string, Legend: {"Y", "N"}, Price: real)
        from PERSON.Employee,
    acquire_game (Date: date, Furn: furnisher, Name: string,
        Type:set_of type.film_type, Equipment: string, Price: real)
        from PERSON.Employee,
    record_purchase(Furnisher:furnisher, Product: string, Price: real) to FURNISHER,
    payed(Value: real) to CASH.Cash_Status,
    record_debit (Price: real) to CASH.Cash_Status,
    debit_values(Rid: integer, Furn: furnisher,Price: real) to CASH.Debit,
    inscribe_video (Date: date, Furn: furnisher, Film: string,
        Type: set_of type.film_type, NrTapes: integer, Director: string,
        Actors: set_of string, Legend: {"Y", "N"})
        to TAPE_CONTROL.General_Tape_Control,
    inscribe_game (Date: date, Furn: furnisher, Name: string,
        Type: set_of type.game_type, Equipment: string)
        to TAPE_CONTROL.General_Tape_Control },
  states = { active },
  rules = {
```

```
begin:
      msg(← add_role) ⇒ state(active),
video_acquisition1:
      state(active),
      msg(← acquire_video (Date,Furn,Film,Type,NrTapes,Dir,Acts,Leg,Price) ⇒
      msg(← record_purchase(Furn, video, Price)),
      msg(→ payed (Price)),
      msg(→ inscribe_video (Date, Furn, Film, Type, NrTapes, Dir, Acts,Leg)),
      state(active);
      ( exists Oid (has_class_instance(cash, Oid) and
      exists Rid (has_role_instance(Oid, cash_status, Rid) and
      value(Rid, in_cash) ≥ Price )) ),
     /* The rule is only executed if there is enough money in cash. */
game_acquisition1:
      state(active),
      msg(← acquire_game (Date, Furn, Name, Type, Equipment, Price) ⇒
      msg(← record_purchase(Furn, game, Price)),
      msg(→ payed (Price)),
      msg(→ inscribe_game (Date, Furn, Name, Type, Equipment)),
      state(active);
      ( exists Oid (has_class_instance(cash, Oid) and
      exists Rid (has_role_instance(Oid, cash_status, Rid) and
      value(Rid, in_cash) ≥ Price )) ) }
     /* The rule is only executed if there is enough money in cash. */
video_acquisition2:
      state(active),
      msg(← acquire_video (Date,Furn,Film,Type,NrTapes,Dir,Acts,Leg,Price) ⇒
      msg(← record_purchase(Furn, video, Price)),
      msg(→ record_debit (Price)),
      msg(→ add_role(cash.debit, Rid)), msg(→ debit_values(Rid, Furn,Price)),
      msg(→ inscribe_video (Date, Furn, Film, Type, NrTapes, Dir, Acts,Leg)),
      state(active);
      ( exists Oid (has_class_instance(cash, Oid) and
      exists Rid (has_role_instance(Oid, cash_status, Rid) and
      value(Rid, in_cash) < Price )) ),
     /* The rule is only executed if there is not enough money in cash to pay for the
        tape. */
game_acquisition2:
      state(active),
      msg(← acquire_game (Date, Furn, Name, Type, Equipment, Price) ⇒
      msg(← record_purchase(Furn, game, Price)),
      msg(→ record_debit (Price)),
      msg(→ add_role(cash.debit, Rid)), msg(→ debit_values(Rid, Furn,Price)),
      msg(→ inscribe_game (Date, Furn, Name, Type, Equipment)),
      state(active);
      ( exists Oid (has_class_instance(cash, Oid) and
      exists Rid (has_role_instance(Oid, cash_status, Rid) and
      value(Rid, in_cash) < Price )) ) }
     /* The rule is only executed if there is not enough money in cash to pay for the
        tape. */
>
```

```
< General_Acquisition,
  messages = {
     acquire_material( Material: string, Furnisher: furnisher, Price: real)
          from PERSON.Employee,
     record_purchase(Furnisher:furnisher, Product: string, Price: real) to FURNISHER,
     payed(Value: real) to CASH.Cash_Status,
     record_debit (Price: real) to CASH.Cash_Status,
     debit_values(Rid: integer, Furn: furnisher,Price: real) to CASH.Debit },
  states = { active },
  rules = {
     begin:
          msg(← add_role) ⇒ state(active),
     acquisition_cash:
          state(active), msg(← acquire_material( Material, Furnisher, Price)) ⇒
          msg(← record_purchase(Furnisher, Material, Price)),
          msg(→ payed (Price)), state(active);
          ( exists Oid (has_class_instance(cash, Oid) and
          exists Rid (has_role_instance(Oid, cash_status, Rid) and
          value(Rid, in_cash) ≥ Price )) ),
          /* The rule is only executed if there is enough money in cash. */
     acquisition_debit:
          state(active), msg(← acquire_material( Material, Furnisher, Price)) ⇒
          msg(← record_purchase(Furnisher, Material, Price)),
          msg(→ record_debit (Price)),
          msg(→ add_role(cash.debit, Rid)), msg(→ debit_values(Rid, Furn,Price)),
          state(active);
          ( exists Oid (has_class_instance(cash, Oid) and
          exists Rid (has_role_instance(Oid, cash_status, Rid) and
          value(Rid, in_cash) < Price )) ) }
          /* The rule is only executed if there is not enough money in cash to pay for the
             purchase. */
>

< Tape_Loose,
  dynamic properties = {
     (fine, real) },
  messages = {
     tape_loose_fine(Value: real) from PERSON.Employee,
     credit_lost_tape(Client: client) from TAPE_CONTROL.General_Tape_Control,
     add_debit(Client: client, Value: real) to PERSON.Client },
  states = { active },
  rules = {
     begin:
          msg(← add_role) ⇒ state(active),
     fine_value:
          state(active), msg(← tape_loose_fine(Value)) ⇒ state(active),
     lost_tape_by_client1:
          state(active), msg(← credit_lost_tape(Client)) ⇒
          msg(→ add_debit(Client, fine)),
          msg(→ add_role(CASH.Credit, Rid)),
          msg(→ debtor_debit_value(Rid, Client, fine)), state(active);
          ( not exists O (has_class_instance(cash, O) and
          exists R (has_role_instance(O, credit, R) and value(R, debtor) = Client)) ),
```

```
        /* This rule executes when this client does not have any debit in the shop. */
    lost_tape_by_client2:
        state(active), msg(← credit_lost_tape(Client)) ⇒
        msg(→ add_debit(Client, fine)),
        msg(→ add_debtor_debit(Client, Fine)), state(active);
        ( exists O (has_class_instance(cash, O) and
        exists R (has_role_instance(O, credit, R) and value(R, debtor) = Client)) ) }
        /* This rule executes when this client does not have any debit in the shop. */
>

< Tape_Sale,
  messages = {
    sell_tape (Tape: tape, Price, real) from PERSON.Employee,
    put_to_sell (Tape: tape, Price: real) to TAPE.Tape_Status,
    buying_tape(Purchaser: purchaser, Tape: tape, Price: real)
        from EXTERNAL_WORLD,
    record_credit(Price: real) to CASH.Cash_Status,
    debtor_debit_value(Rid: integer, Purchaser: purchaser, Price: real)
        to CASH.Credit,
    add_debtor_debit(Purchaser: purchaser, Price: real) to CASH.Credit,
    sold_tape (Tape: tape) to TAPE.Tape_Status },
  states = { active },
  rules = {
    begin:
        msg(← add_role) ⇒ state(active),
    start_to_sell:
        state(active), msg(← sell_tape(Tape, Price)) ⇒
        msg(→ put_to_sell(Tape, Price), state(active),
    selling1:
        state(active), msg(← buying_tape(Purchaser, Tape, Price)),
        msg(→ record_credit(Price)), msg(→ add_role(cash.credit, Rid)),
        msg(→ debtor_debit_value(Rid, Purchaser, Price)),
        msg(→ sold_tape (Tape)), state(active);
        ( exists O1 (has_class_instance(purchaser, O1)) and
        exists O2 (has_class_instance(tape, O2)) and
        not exists Oid (has_class_instance(cash, Oid) and
        exists Rid (has_role_instance(Oid, credit, Rid) and
        value(Rid, debtor) = Purchaser )) ),
        /* The purchaser and the tape exist and this purchaser has no other debit in the
           shop. */
    selling2:
        state(active), msg(← buying_tape(Purchaser, Tape, Price)),
        msg(→ record_credit(Price)), msg(→ add_debtor_debit(Purchaser, Price)),
        msg(→ sold_tape (Tape)), state(active);
        ( exists O1 (has_class_instance(purchaser, O1)) and
        exists O2 (has_class_instance(tape, O2)) and
        exists Oid (has_class_instance(cash, Oid) and
        exists Rid (has_role_instance(Oid, credit, Rid) and
        value(Rid, debtor) = Purchaser )) ) }
        /* The purchaser and the tape exist and this purchaser has already other debits in
           the shop. */
>
```

```
< Salary,
  messages = {
      salary_payment_authorization (Employee:person.employee, Value: real)
          from PERSON.Employee,
      salary_payment(Employee: person.employee, Value: real)
          to EXTERNAL_WORLD,
      payed(Value: real) to CASH.Cash_Status },
  states = { active },
  rules = {
      begin:
          msg(← add_role) ⇒ state(active),
      salary_payment:
          state(active), msg(← salary_payment_authorization(Employee, Value)) ⇒
          msg(→ salary_payment(Employee, Value)),
          msg(→ payed(Value)), state(active);
          ( exists Oid( has_class_instance(person, Oid) and
          exists Rid (has_role_instance(Oid, employee, Rid) )) ) }
>

< Payment_Control,
/* Income in cash and payments done by the store. */
  dynamic properties = {
      (minimum_credit_value, integer),
      (credit_price, real) },
  messages = {
      update_min_credits(Quantity: integer) from PERSON.Employee,
      update_credit_price(Price: real) from PERSON.Employee,
      payment(Client: client, Value: real) from EXTERNAL_WORLD,
      received(Value: real) to CASH.Cash_Status,
      deduct_debtor_debit(Client: client, Value: real) to CASH.Credit,
      deduct_debit (Client: client, Value: real) to PERSON.Client,
      buy_credits (Client:person.client, Quantity: integer ) from EXTERNAL_WORLD,
      add_credits (Client:person.client, Quantity: integer) to PERSON. Client,
      pay(Credor: furnisher, Value: real) from PERSON.Employee,
      payed(Value: real) to CASH.Cash_Status,
      deduct_total_debit(Debit: cash.debit, Value: real) to CASH.Debit },
  states = { active },
  rules = {
      begin:
          msg(← add_role) ⇒ state(active),
      update_minimum_credits_quantity:
          state(active), msg(← update_min_credits(Quantity)) ⇒ state(active),
      update_credit_price:
          state(active), msg(← update_credit_price(Price)) ⇒ state(active),
      client_payment:
          state(active), msg(← payment(Client, Value)) ⇒
          msg(→ received(Value)),
          msg(→ deduct_debtor_debit(Client, Value)),
          msg(→ deduct_debit (Client, Value)), state(active);
          ( exists Oid( has_class_instance(cash, Oid) and
          exists Rid (has_role_instance (Oid, credit, Rid) and
          value(Rid, debtor) = Client ),
      purchaser_payment:
          state(active), msg(← payment(Purchaser, Value)) ⇒
```

msg(→ received(Value)),
msg(→ deduct_debtor_debit(Purchaser, Value)), state(active);
(exists Oid(has_class_instance(cash, Oid) and
exists Rid (has_role_instance(Oid, credit, Rid) and
value(Rid, debtor) = Purchaser))),

somebody_buys_credits_for_client:
state(active), msg(← buy_credits (Client, Quantity)) ⇒
msg(→ received(Price)),
msg(→ add_credits (Client, Quantity)), state(active);
(exists Oid (has_class_instance(person, Oid) and
exists Rid (has_role_instance(Oid, client, Rid) and
value(Rid, rId) = Client)) and Quantity ≥ value(minimum_credit_value)),
/* Credits can be bought by anybody, not only by the client himself; the corresponding client must exist. */

payment_to_furnisher:
state(active), msg(← pay(Credor, Value)) ⇒
msg(→ payed(Value)),
msg(→ deduct_total_debit(Debit, Value)), state(active);
(exists Oid (has_class_instance(cash, Oid), and
exists Rid (has_role_instance (Oid, debit, Rid) and
value(Rid, rId) = Debit)) and
exists O (has_class_instance(cash,O) and
exists R (has_role_instance (O,cahs_status, R) and
value(R, in_cash) ≥ Value))) }
/* The authorized payment can only be done if this credor exists and when there is enough money in cash. */

>

BIBLIOGRAFIA

[ADI 85] ADIBA, M.; QUANG, N.B.; OLIVEIRA, J.P.M. de. Time concept in generalized data bases. In: ACM ANNUAL CONFERENCE, Oct. 14-16, 1985, Denver. **Proceedings...** New York: ACM, 1985. p.214-23.

[ADI 86a] ADIBA, M.; QUANG, N.B.; OLIVEIRA, J.P.M. de Notion de temps dans les bases de données generalisées. In: ADIBA, M. et al. **Nouvelles Perspectives des Bases de Données**. Paris: Eyrolles, 1986. p.72-86.

[ADI 86b] ADIBA, M. Problématique des bases de données multi-media. In: ADIBA, M. et al. **Nouvelles Perspectives des Bases de Données**. Paris: Ed. Eyrolles, 1986. p.11-17.

[ADI 87] ADIBA, M.; BUI QUANG, N.; COLLET, C. Aspect temporels, historiques et dynamiques des bases de données. **TSI - Technique et Science Informatiques**, AFCET-Bordas, v.6, n.11, p.832-43, Nov. 1983.

[ALL 83] ALLEN, J.F. Maintaining knowledge about temporal intervals. **Communications of the ACM**, New York, v.26, n.11,p.832-43, Nov. 1983.

[AHL 87] AHLSEN, M. et al. An Architecture for object management in OIS. **ACM Transactions on Office Information Systems**, New York, v.2, n.3, p.173-96, July 1984.

[ALB 93] ALBANO, A.; BERGAMINI, R.; GHELLI, G.; ORSINI, R. An Object data model with roles. In: VERY LARGE DATABASES CONFERENCE, 19., 1993, Dublin, Ireland. **Proceedings...** Dublin: [s.n.], 1993. p.39-51.

[ALV 88] ALVARES, L.O.C. **Contribuition à l'Étude du Pilotage de la Modelisation das Systèmes d'Information**. Grenoble: Université Joseph Fourier, 1988. 225 p. (Tèse de Doctorat).

[ARA 90] ARAPIS, C. Specifying object life-cycles. In: TSICHRITZIS,D. (Ed.) **Object Management**. Genève: Universitè de Genève, 1990. p.197-225.

[ARA 91] ARAPIS, C. Specifying object interactions. In: TSICHRITZIS,D. (Ed.) **Objects Composition**. Genebra: Université de Genève, 1991. p.303-22.

[ARI 86] ARIAV, G. A Temporally oriented data model. **ACM Transactions on Database Systems**, New York, v.4, n.4, p.499-527, Dec. 1986.

[BAN 88] BANCILLON, F. et al. The Design and implementation of O_2, and object-oriented database system. In: **Advances in Object-oriented Database Systems**. Berlin: Springer-Verlag, 1988. p.1-22.

[BAN 87] BANERJEE, J. et al. Data model issues for object-oriented applications. **ACM Transactions on Office Information Systems**, New York, v.5, n.1, p.3-26, Jan.1987.

[BAN 89] BANCILLON, F. Query languages for object-oriented databases: analysis and a proposal. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 4., 5-7 Apr. 1989, Campinas, São Paulo. **Proceedings...** Campinas: R.Vieira Gráfica e Editora Ltda, 1989. p.22-38.

[BAR 85] BARBIC, F.;CERI, S.;BRACCHI, G. Modeling and integrating procedures in office information systems design. **Information Systems**, New York, v.10, n.2, p.149-68, 1985.

[BEE 90] BEERI, C. A Formal approach to object-oriented databases. **Data & Knowledge Engineering**, Netherlands, v.5, n.4, p.353-382, Oct. 1990.

[BEL 91] BELLINZONA,R.; FUGINI,M.G. **RECAST PROTOTYPE DESCRIPTION**. Milano: Politecnico di Milano, 1991. (Tech. Repport).

[BJO 89] BJORNER, D. **The VDM Specification & Implementation Methodology**. Petrópolis: IFIP, 1989. 100p.

[BOE 88] BOEHM, B.W. A Spiral model of software development and enhancement. **Computer**, Los Alamitos, v.21, n.5, p.61-72, May 1988.

[BOL 83] BOLOUR, A.; DEKEYSER, L.J. Abstractions in temporal information. **Information Systems**, Great Britain, v.8, n.1, p.41-9, 1983.

[BRA 84] BRACCHI, G.; PERNICI, B. The Requirements of office systems. **ACM Transactions on Office Information Systems**, New York, v.2, n.2, p.151-70, Apr. 1984.

[BRU 93] BRUNET, J.; CAUVET, C.; MEDDAHI, D.; SEMMAK, F. Object-oriented analysis in practice. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING CAiSE'93, 5., June 8-11, 1993, Paris. **Proceedings...** Berlin: Springer-Verlag, 1993. p.293-308. (Lecture Notes in Computer Science 685).

[CAR 86] CAREY, M.J. et al. The Architecture of the EXODUS extensible DBMS. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED DATABASE SYSTEMS, 23-26 Sept. 1986, Pacific Grove, California. **Proceedings...** Pacific Grove: K.Dittrich & U.Dayal(Eds.), 1986. p.52-65.

[CAR 88a] CAREY, M.J.; DEWITT, D.J.; VANDENBERG, S.L. A Data model and query language for EXODUS. **ACM-SIGMOD Record**, v.17, n.3, p.413-23, Sept. 1988.

[CAR 88b] CARMO, J.; SERNADAS, A. A Temporal logic framework for a layered approach to systems specification and verification. In: ROLLAND,C.; BODART,F.; LEONARD,M. (Eds.) **Temporal Aspects in Information Systems**. Amsterdam: North-Holland, 1988. p.31-46.

[CAS 88] CASAIS, E. An Object oriented system implementation KNOs. In: CONFERENCE OF OFFICE INFORMATION SYSTEMS, 23-25 mar. 1988, Palo Alto, California. **Proceedings...** Palo Alto: Robert B. Allen (Ed.), 1988. p.284-90.

[CAS 91] CASAIS, E. Managing class evolution through reorganization. In: TSICHRITZIS, D. (Ed.) **Objects Composition**. Genève: Université de Genève, 1991. p.287-301.

[CAS 82] CASTILHO, J.M.V.; CASANOVA, M.A.; FURTADO, A.L. A Temporal framework for database specifications. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 8., Sept. 1982, Mexico City. **Proceedings...** Mexico City: [s.n.], 1982. p.280-91.

[CEL 92] CELENTANO, A.; FUGINI, M.G.; POZZI, S. Conceptual document browsing and retrieval in *KABIRIA*. SIGMOD Record, v.21, n.2, June 1992. p.3. Also published in: ACM SIGMOD INTERNATIONAL

CONFERENCE ON MANAGEMENT OF DATA, June 2-5, 1992, San Diego, California.

[CHE 76] CHEN, P.P. The Entity-relationship model - toward a unified view of data. **ACM Transactions on Database Systems**, New York, v.1, n.1, p.9-36, Mar. 1976.

[CHE 93] CHENG, T.S.; GADIA, S.K. An Object-oriented model for temporal databases. In: INTERNATIONAL WORKSHOP ON AN INFRASTURCTURE FOR TEMPORAL DATABASES, June 14-16, Arlington, Texas. **Proceedings...** Arlington: [s.n.], 1993. p.N-1-N-19.

[CLI 87] CLIFFORD, J.; CROKER, A. The Historical Relational Data Model (HRDM) and algebra based on lifespans. In: INTERNATIONAL CONFERENCE ON DATA ENGINERING, 3., Feb. 1987, Los Angeles. **Proceedings...** Los Angeles: [s.n.], 1987.

[CLI 88a] CLIFFORD, J.; CROKER, A. Objects in time. **Data Engineering**, Washington, v.11, n.4,p.11-18, Dec. 1988.

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

[CLI 93] CLIFFORD, J.; CROKER, A. The Historical Relational Data Model revisited. In: TANSEL, A.U. et al. (Eds.) **Temporal Databases**. Redwood City, California: Benjamin/Cummings, 1993. p.6-27.

[COD 70] CODD, E.F. A Relational model of data for large shared data banks. **Communications of the ACM,** New York, v.13, n.6, p.377-387, June 1970.

[COH 86] COHEN, B.; HARWOOD, W.T.; JACKSON, M.I. **The Specification of Complex Systems**. Great Britain: Addison-Wesley, 1986. 143 p.

[COR 91] CORSETTI, E. et al. Dealing with different time scales in formal specifications. In: INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN, 6., Oct. 25-6, 1991, Como, Italy. **Proceedings...** IEEE Computer Society Press, [s.n.], 1991. p.92-101.

[DAT 87] DATE, C. **Introduction ao standard SQL**. Paris: InterEditions, 1987. 239p.

[DAY 93] DAYAL, U.; WUU, G.T.J. Extending existing DBMSs to manage temporal data: an object-oriented approach. In: INTERNATIONAL WORKSHOP ON AN INFRASTURCTURE FOR TEMPORAL DATABASES, June 14-16, Arlington, Texas. **Proceedings...** Arlington: [s.n.], 1993. p.J-1-J-8.

[DEA 91a] DeANTONELLIS, V.; PERNICI, B. **ITHACA Object-Oriented Methodology Manual - Application Development Manual**. Milano: Politecnico di Milano, 1991. (ITHACA ESPRIT Project).

[DEA 91b] DeANTONELLIS, V.; PERNICI, B.; SAMARATI, P. Object-orientation in the analysis of work organization and agent cooperation. In: INTERNATIONAL CONFERENCE ON DYNAMIC ASPECTS IN INFORMATION SYSTEMS, 2., July 1991, Washington, DC. **Proceedings...** Washington: [s.n.], 1991.

[DEA 91c] De ANTONELLIS, V.; PERNICI, B.; SAMARATI, P. F-ORM Method: a F-ORM Methodology for reusing specifications. In: Assche F.V.; Moulin,

B.; Rolland, C. (Eds.) **Object Oriented Approach in Information Systems**. Amsterdam: North-Holland, 1991. p.117-35.

[DEM 79] DeMARCO, T. **Structured Analysis and System Specification**. New York: Yourdon, 1979.

[DER 86] DERRET, N.P. et al. An Object-oriented approach to data management. **IEEE**, p.330-35, 1986.

[DEU 91] DEUX, 0. et al. The O2 System. **Communications of the ACM**, New York, v.34, n.10, p.34-48, Oct.1991

[EDE 93a] EDELWEISS, N.; OLIVEIRA, J.P.M.de; PERNICI, B. An Object-Oriented Temporal Model. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING - CAiSE'93, 5., June 8-11, 1993, Paris. **Proceedings...** Paris: [s.n.], 1993. p. 397-415. (Lecture Notes in Computer Science, 685).

[EDE 93b] EDELWEISS, N.; CASTILHO, J.M.V.de; OLIVEIRA, J.P.M.de. A Temporal logic language for temporal conditions definitions. In: INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY, 13., Oct. 14-16, 1993, La Serena, Chile. **Proceedings...** Santiago: University of Chile, 1993.

[EDE 93c] EDELWEISS, N.; OLIVEIRA, J.P.M.de. Representação interna de um banco de dados para suportar o modelo Temporal F-ORM. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 8., 12-14 maio, 1993, Campina Grande, Paraíba. **Anais...** Campina Grande: SBC, 1993. p.297-311.

[EDE 94] EDELWEISS, N; OLIVEIRA, J.P.M.de; PERNICI, B. An Object-oriented approach to a temporal query language. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS, 5., Sept. 7-9, 1994, Athens, Greece. **Proceedings...** (to be published).

[ELM 90] ELMASRI, R.; WUU, G.T.J. A Temporal model and query language for ER databases. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 6., Feb.5-9, 1990, Los Angeles. **Proceedings...** Los Angeles: IEEE, 1990. p.76-83.

[ELM 93] ELMASRI, R.; WUU, G.T.J.; KOURAMAJIAN, V. A Temporal model and query language for EER databases. In: **Temporal Databases: Theory, Design and Implementation**. Redwood City: Benjamim/Cummings, 1993. p.212-247.

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

[FIN 91] FINGER, M.; McBRIEN, P.; OWENS, R. Databases and executable temporal logic. In: ESPRIT '91 ANNUAL ESPRIT CONFERENCE, Nov. 25-29, 1991, Brussels. **Proceedings...** Brussels: ECSC, 1991. p.289-302.

[FIS 87] FISHMAN, D.H. et al. IRIS: An Object-oriented database management system. **ACM Transactions on Office Information Systems**, New York, v.5, n.1, p.48-69, Jan. 1987.

[GAD 88] GADIA, S.K. A Homogeneous relational model and query language for temporal databases. **ACM Transactions on Database Systems**, New York, v.14, n.4, p.418-448, Dec. 1988.

[GAB 91] GABBAY, D.; McBRIAN, P. Temporal logic & historical databases. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES, 17., Sept. 3-6, 1991, Barcelona. **Proceedings...** Barcelona: Industria Grafica, 1991. p.423-30.

[GAN 91] GANDRIEAU, M.-A.; DURIN, B. Identification and classification of reusable elements in space domain. In: **REBOOT Workshop on REUSE** , Sept. 26-27, 1991, Grenoble. Grenoble: [s.n.], 1991. (ESPRIT Project 5327).

[GAR 88] GARZA, J.F.; KIM, W. Transaction management in an object-oriented database system. **ACM-SIGMOD Record**, New York, v.17, n.3, p.37-45, Sept. 1988.

[GIR 90] GIRARDI, M.R. **Uma Ferramenta de Apoio à Reutilização de Software no Desenvolvimento Orientado a Objetos**. Porto Alegre: CPGCC - UFRGS, 1990. 228p. Tese de Mestrado.

[GIR 92] GIRARDI, R. Application engineering: putting reuso to work. In: **Object Frameworks**. Genebra: University of Genebra, 1992. p.137-150.

[GOG 78a] GOGHEN, J.A. Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs. In: INTERNATIONAL CONFERENCE ON MATHEMATICAL STUDIES OF INFORMATION PROCESSING, Aug. 23-26, 1978, Kioto. **Proceedings...** Berlin: Springer-Verlag, 1978. p.429-475. Lecture Notes in Computer Science 75.

[GOG 78b] GOGHEN, J.A.; THATCHER, J.W.; WAGNER, E.G. An Initial algebra approach to specification correctness and implementation of abstract data types. **Current Trends in Programming Methodology**. Englewood Cliffts: Prentice-Hall, 1978. v.4 - Data Structuring. p.80-149.

[GOG 82] GOGHEN, J.A.; MESEGUER, J. Rapid prototyping in OBJ executable language. **ACM Sigsoft Software Engineering Notes**, v.7, n.5, p.75-84, Dec. 1982.

[GRE 86] GREENSPAN, S.J.; BORGIDA, A.; MYLOPOULOS, J. A Requirements modeling language and its logic. In: BRODIE,M.L. & MYLOPOULOS,J. (eds.) **On Knowledge Base Systems**. Springer-Verlag: New York, 1986. p.471-502.

[HEI 88] HEIJMINK, F. et al. Development of tools for designing OIS. In: **Information Technology for Organizational Systems**. Brussels: Elsevier/North-Holland, 1988. p.66-73.

[HEN 90] HENDERSON-SELLERS, B.; EDWARDS, J.M. The Object-oriented systems life cicle. **Communications of the ACM**, New York, v.33, n.9, p.142-59, Sept. 1990.

[HEN 92] HENDERSON-SELLERS, H.; FREEMAN, C. Cataloguing and classification for object libraries. **Software Engineering Notes**, New York, v.17, n.1, p.62-64, Jan. 1992.

[HEU 88] HEUSER, C.A. **Modelagem Conceitual de Sistemas**. Buenos Aires: Kapelusz, 1988. 94p.

[HOR 87] HORNICK, R.K.; ZDONIK, B.Z. A Shared, segmented memory system for an object-oriented database. **ACM Transactions on Office Information Systems**, New York, v.5, n.1, p.79-95, Jan. 1987.

[JEN 93a] JENSEN, C.S. (Ed.). Proposed temporal database concepts - May 1993. In: INTERNATIONAL WORKSHOP ON AN INFRASTRUCTURE FOR TEMPORAL DATABASES, June 14-16, 1993. **Proceedings...** Arlington: Richard Snodgrass (Ed.), 1993.

[JEN 93b] JENSEN, C.S. et al. (Eds.) The TSQL Benchmark. In: INTERNATIONAL WORKSHOP ON AN INFRASTRUCTURE FOR TEMPORAL DATABASES, June 14-16, Arlington, TX. **Proceedings...** Arlington: Richard Snodgrass (Ed.), 1993. p.QQ-1-QQ-18.

[JON 86] JONES, C.B. **Systematic Software Development using VDM**. UK: Prentice Hall, 1986.

[KÄF 92] KÄFER, W.; SCHÖNING, H. Realizing a temporal complex-object data model. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, June 2-5, 1992, San Diego. **Proceedings...** San Diego: [s.n.], 1992. p.266-275.

[KIM 87] KIM, W. et al. Composit objects support in an object-oriented database system. **SIGPLAN NOTICES**, New York, v.22, n.12, p.118-25, Oct. 1987.

[KIN 85] KING, R.; McLEOD, D. A Database design methodology and tool for information systems. **ACM Transactions on Office Information Sysytems**, v.3, n.1, p.2-21, Jan. 1985.

[KLA 83] KLAEREN, H.A. **Algebraische Spezifikation - eine Einführung**. Berlin: Springer-Verlag, 1983. 235p.

[KON 82] KONSYNSKI, B.R.;BRACKER, L.C.;BRACKER JR, W.E. A Model for specification of office communications. **IEEE Transactions on Communications**, New York, v.COM-30, n.1, p.27-36, Jan. 1982.

[KOR 90] KORSON, T.; McGREGOR, J.D. Understanding object-orientation: a unifying paradigm. **Communications of the ACM**, New York, v.33, n.9, p.40-60, Sept. 1990.

[KOW 86] KOWALSKI, R.; SERGOT, M. A Logic based calculus of events. **New Generation Computing**, 4, 1986. p.67-95.

[LAD 88] LADDEN, R.M. Survey on issues to be considered in the development of an object-oriented development methodology for ADA. **ACM Software Engineering Notes**, New York, v.13, n.3, p.24-31, July 1988.

[LEC 88] LÉCLUSE, C.; RICHARD, P.; VELEZ, F. O_2, an object-oriented data model. **ACM SIGMOD Record**, New York, v.17, n.3, p.424-433, Sept. 1988.

[LEN 87] LENZ, M.; SCHMIDT, H.A.; WOLF, P.F. Software reuse through building blocks. **IEEE Software**, Los Alamitos, v.4, n.4, p.34-42, July 1987.

[LIP 87] LIPECK, U.W.; SAAKE, G. Monitoring dynamic integrity constraints based on temporal logic. **Information Systems**, Great Britain, v.12, n.3, p.255-269, 1987.

[LOR 93] LORENTZOS, N.A. The Interval-extended Relational Model and its applications to valid-time databases. In: A.U. TANSEL et al. (eds.) **Temporal Databases**. Redwood City, California: Benjamin/Cummings, 1993. p.67-91.

[LOU 91a] LOUCOPOULOS, P.; THEODOULIDIS, B.; PANTAZIS, D. Business rules modelling: conceptual modelling and object-oriented specifications. In: ASSCHE, F.V.; MOULIN, B.; ROLLAND, C. **Object Oriented Approach in Information Systems**. Amsterdam: North-Holland, 1991. (Proceedings of the IFIP TC8/WG8.1 Working Conference – Quebec City, Canada, Oct. 28-31, 1991). p.323-342.

[LOU 91b] LOUCOPOULOS,P.; McBRIEN,P.; PERSSON,U.; SCHUMACKER,F.; VASEY,P. TEMPORA - Integrating database technology, rule-based systems and temporal reasoning for information systems development. (to be included in the **IEEE Knowledge Engineering Newsletters**, Feb. 1991.

[LYN 84] LYNGBAEK, P.; MCLEOD, D. Object management in distributed information systems. **ACM Transactions on Office Information Systems**, New York, v.2, n.2, p.96-122, Apr. 1984.

[LYN 86a] LYNGBAEK, P. **Atomic vs. Molecular Objects in Iris**. Palo Alto: Hewlett-Packard Laboratories, 1986. 11p. (STL-86-08).

[LYN 86b] LYNGBAEK, P.; KENT, W. A Data modeling methodology for the design and implementation of information systems. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED DATABASE SYSTEMS, 23-26 Sept. 1986, Pacific Grove, California. **Proceedings...** Pacific Grove: K.Dittrich & U.Dayal (Eds.), 1986. p.6-17.

[MAI 91a] MAIOCCHI, R.; PERNICI, B.; BARBIC, F. Automatic deduction of temporal information. University of Udine, Dipartimento de Matematica e Informatica, 1991. 58p. (Research Report).

[MAI 91b] MAIOCCHI, R.; PERNICI, B. Temporal Data Management Systems: a comparative view. **IEEE Transactions on Knowledge and Data Engineering**, v.3, n.3, Dec. 1991.

[MAI 92] MAIOCCHI, R.; PERNICI, B.; BARBIC, F. Automatic deduction of temporal information. **ACM Transactions on Database Systems**, New York, v.17, n.4, p.647-688, Dec. 1992.

[MAN 81] MANNA, Z.; PNUELI, A. Verification of concurrent programs: the temporal framework. **The Correctness Problem of Computer Science**. BOYE-MOORE (Eds.). Academic press, 1981. p.215-273.

[MAN 86] MANOLA, F..; DAYAL, U. PDM: An Object-oriented data model. In: INTERNATIONAL WORKSHOP ON OBJECT ORIENTED DATABASE SYSTEMS, 23-6 Sept. 1986, Pacific Grove, California. **Proceedings...** Pacific Grove: K.Dittrich, & U.Dayal (Eds.), 1986. p.18-25.

[MEY 90] MEYER, B. Lessons for the design of the Eiffel libraries. **Communications of the ACM**, New York, v.33, n.9, p.68-88, Sept. 1990.

[MOT 93] MOTSCHNIG-PITRIK, R. The semantics of parts versus aggregates in data/knowledge modelling. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING - CAiSE'93, 5., June 8-11, Paris. **Proceedings...** Paris: [s.n.], 1993. p.352-373. (Lecture Notes in Computer Science, 685).

[MYL 90] MYLOPOULOS, J. et al. Telos: representing knowledge about information systems. **ACM Transactions on Information Systems**, New York, v.8, n.4, p.325-62, Oct. 1990.

[NAV 88] NAVATHE, S.B.; AHMED, R. TSQL: A Language interface for history databases. In: ROLLAND, C.; BODART, F.; LEONARD, M. (Eds.) **Temporal Aspects in Information Systems**. Amsterdam: North-Holland, 1988. p.109-122.

[NAV 93] NAVATHE, S.B.; AHMED, R. Temporal extensions to the relational Model and SQL. In: TANSEL, A.U. et al. (Eds.) **Temporal Databases**. Redwood City, California: Benjamin/Cummings, 1993. p.92-109.

[NEI 84] NEIGHBORS, J.M. The Draco approach to constructing software from reusable components. **IEEE Transactions on Software Engineering**, New York, v.10, n.5, p.564-73, Sept. 1984.

[NEW 80] NEWMAN, W. Office models and office systems design. In: N.Naffath (ed.), **Integrated Office Systems - Burotics**. Amsterdam: North-Holland, 1980. p.3-10.

[OLI 93] OLIVEIRA, L.C.M.de. **Incorporação da Dimensão Temporal a Bancos de Dados Orientados a Objetos**. Campinas: Departamento de Ciência da Computação/IMECC - UNICAMP, 1993. Dissertação de Mestrado.

[PER 88] PERNICI, B. et al. **C-TODOS : An Automatic Tool for System Conceptual Design**. Milano, Italy: Politecnico di Milano, Oct. 1988. 103p.

[PER 90] PERNICI, B. Objects with Roles. **SIGOIS Bulletin**, v.11, n.2-3, p.205-15, 1990. Also published in: ACM/IEEE CONFERENCE ON OFFICE INFORMATION SYSTEMS, April 25-27, 1990, Cambridge, MA, Proceedings....

[PIN 90] PINTADO, X. Selection and exploration in an object-oriented environment: the affinity browser. In: **Object Management.** Genève: Universitè de Genève, 1990. p.79-105.

[PIS 93] PISSINOU, N.; MAKKI, K.; YESHA, Y. Research perspectives on time in object databases. In: INTERNATIONAL WORKSHOP ON AN INFRASTRUCTURE FOR TEMPORAL DATABASES, June 14-16, Arlington, Texas. **Proceedings...** Arlington: Richard Snodgrass (Ed.), 1993. p.Y-1-Y-9.

[PON 91] PONTÉN, L. Reuse in Software Engineering: an overview. In: **REBOOT Workshop on REUSE**, Sept. 26-27, 1991, Grenoble. Grenoble: [s.n.], 1991. (ESPRIT Project 5327).

[PRI 87] PRIETO-DIAZ, R.; FREEMAN, P. Classifying software for reusability. **IEEE Software**, Los Alamitos, v.4, n.1, p.6-16, Jan. 1987.

[PRO 89] PROFROCK, A.-K.; TSICHRITZIS, D.; MULLER, G.; ADLER, M. ITHACA: an Integrated tollkit for highly advanced computer applications. TSICHRITZIS, D. (Ed) **Object Oriented Development**. Genebra: Université de Genève, 1989. p.321-44.

[PUR 90] PURCHASE, J.A.; WINDER, R. Message pattern specification: a new technique for handling errors in parallel object oriented systems. **SIGPLAN Notices**, v.25, n.10, p.116-125, Oct. 1990.

[REI 85] REISIG, W. **Petri Nets : An Introduction**. Berlin: Springer-Verlag, 1985. 161p.

[RIC 87a] RICHTER, G. et al. **Generic Office Frame of Reference (GOFOR)**. St.Augustin, Germany: GMD, 1987. 142p. (Esprit Project 56: Functional Analysis on Office Requirements).

[RIC 87b] RICHARDSON, J.E.; CAREY, M.J. Programming constructs for database system implementation in EXODUS. In: ANNUAL CONFERENCE ON MANAGEMENT OF DATA, 27-29 May, 1987, San Francisco. **Proceedings...** California: ACM, U.Dayal & I.Traiger (Eds), 1987. p.208-219.

[RIC 91] RICHARDSON, J.; SCHWARZ, P. Aspects: extending objects to support multiple, independent roles. **SIGMOD Record**, New York, v.20, n.2, p.298-307, June 1991. Also published in: ACM SIGMOD INSTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, May 29-31, Denver, Colorado. Proceedings...

[ROL 82] ROLAND, C.; RICHARD, C. The REMORA Methodology for information systems design and management. In: **Information Systems Design Methodologies : A Comparative Review**. Amsterdam: North-Holland Publishing Company, 1982. p.369-426.

[ROS 93] ROSE, E.; SEGEV, A. **TOOSQL - A Temporal object-oriented query language**. Berkeley: The University of California and Information and Computing Science Division, Mar. 1993. (Tech. Rep. LBL-33855).

[SAR 90] SARDA, N.L. Extensions to SQL for historical databases. **IEEE Transaction on Knowledge and Data Engineering**, v.2, n.2, p. 220-230, June 1990.

[SCH 83] SCHIEL, U. An Abstract introduction to the Temporal-Hierarchic Data Model (THM). In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 9., Oct. 31 - Nov. 2, 1983, Florence (Italy). **Proceedings...** Italy: [s.n.], 1983. p.322-30.

[SCH 91] SCHIEL, U. An Open environment for objects with time and versioning. In: EASTEUROOPE'91, Bratislava, CSFR, 1991. **Proceedings...** Bratislava: [s.n.], 1991. p.119-25.

[SCI 89] SCIORE, E. Object specialization. **ACM Transactions on Information Systems**, New York, v.7, n.2, p.103-122, April 1989.

[SEG 88a] SEGEV, A.; SHOSHANI, A. Modeling temporal semantics. In: ROLLAND,C.; BODART,F.; LEONARD,M. (Eds.) **Temporal Aspects in Information Systems**. Amsterdam: North-Holland, 1988. p.47-57.

[SEG 88b] SEGEV, A.; SHOSHANI, A. Functionality of temporal data models and physical design implications. **Data Engineering**, Washington, v.11, n.4, p.38-45, Dec. 1988.

[SER 91] SERNADAS, C. et al. In-the-large object-oriented design of information systems. In: ASSCHE, F.V.; MOULIN, B.; ROLLAND, C. **Object Oriented Approach in Information Systems**. Amsterdam: North-Holland, 1991. (Proceedings of the IFIP TC8/WG8.1 Working Conference – Quebec City, Canada, Oct. 28-31, 1991). p.209-232.

[SNO 85] SNODGRASS, R.; AHN, I. A Taxonomy of time in databases. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF

DATA, Texas, May 28-31, 1985. **Proceedings...** New York: ACM, 1985. p.236-46.

[SNO 86] SNODGRASS, R.; AHN, I. Temporal databases. **Computer**, Los Alamitos, v.19, n.9, p.35-42, Sept. 1986.

[SNO 87] SNODGRASS, R. The Temporal query language TQuel. **ACM Transactions on Database Systems**, New York, v.12, n.2, p.247-298, June 1987.

[STA 84] STANDISH, T.A. An Essay on software reuse. **IEEE Transactions on Software Engineering**, New York, v.10, n.5, p.494-7, Sept. 1984.

[STE 86] STERLING, L.; SHAPIRO, E. **The Art of Prolog - Advanced Programming Techniques**. Cambridge: The MIT Press, 1986.

[STO 76] STONEBRAKER, M. et al. The Design and implementation of INGRES. **Transactions on Database Systems**, New York, v.1, n.3, p.189-222, Sept. 1976.

[STO 86] STONEBRAKER, M.; ROWE, L.A. The Design of POSTGRES. **ACM SIGMOD**, 1986. p.340-55.

[SU 91a] SU, J. Dynamic contraints and object migration. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES, 17., Sept. 3-6, Barcelona. **Proceedings...** Barcelona: Industria Grafica, 1991. p.233-242.

[SU 91b] SU, S.Y.W.; CHEN, H.-H. M. A Temporal knowledge representation model OSAM*/T and its query language OQL/T. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 17., Sept. 3-6, 1991, Barcelona (Spain). **Proceedings...** Barcelona: Industria Grafica, 1991. p.431-442.

[SU 93] SU, S.Y.W.; CHEN, H.-H.M. Modeling and management of temporal data in object-oriented knowledge bases. In: INTERNATIONAL WORKSHOP ON AN INFRASTRUCTURE FOR TEMPORAL DATABASES, June 14-16, Arlington, Texas. **Proceedings...** Arlington: Richard Snodgrass (Ed.), 1993. p.HH-1-HH-18.

[SUT 91] SUTCLIFFE, A.G. Object oriented systems analysis: the abstraction question. In: **Object Oriented Approach in Information Systems**. Amsterdam: North-Holland, 1991. p.23-37. (Proceedings of the IFIP TC8/WG8.1 Working Conference - Quebec City, Canada, Oct. 28-31, 1991).

[TAN 88] TANSEL, A.U. Non first normal form temporal relational model. **Data Engineering**, Washington, v.11, n.4, p.46-52, Dec. 1988.

[TAN 93] TANSEL, A.U. A Generalized relational framework for modeling temporal data. In: **Temporal Databases: Theory, Design and Implementation**. Redwood City: Benjamim/Cummings, 1993. p.183-201.

[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.

[TUE 88] TUENI, M.; LI, J.; FARES, P. AMS: A Knowledge-based approach to task representation, organization and coordination. CONFERENCE ON OFFICE INFORMATION SYSTEMS, Mar. 23-25, 1988, Palo Alto, California. **Proceedings...** New York: ACM, 1988. p.78-87.

[VLA 93] VLACHANTONIS, N.; HERZIG,R.; GOGOLLA, M.; DENKER, G.; CONRAD, S.; EHRICH, H.-D. Towards reliable information systems: the

KorSo approach. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING CAiSE'93, 5., June 8-11, 1993, Paris. **Proceedings...** Berlin: Springer-Verlag, 1993. p.463-482. (Lecture Notes in Computer Science 685).

[WIE 91] WIEDERHOLD, G.; JAJODIA, S.; LITWIN,W. Dealing with granularity of time in temporal databases. In: INTERNATIONAL CONFERENCE CAISE'91, 3., May 13-15, 1991, Trondheim, Norway. **Proceedings...** Berlin: Springer-Verlag, 1991. p.124-40.

[WIR 89] WIRSING, M. **Algebraic Specification: Semantics, Parameterization and Refinement**. Petrópolis: IFIP, 1989. 66p.

[WOO 86] WOO, C.C.; LOCHOVSKY, F.H. Supporting distributed office problem solving in organizations. **ACM Transactions on Office Information Systems**, New York, v.4, n.3, p.185-204, July 1986.

[WUD 92] WUDA, G.; DAYAL, U. A Uniform model for temporal object-oriented databases. In: IEEE DATA ENGINEERING CONFERENCE, 8. **Proceedings...** 1992.

[WUU 93] WUU, G.T.J.; DAYAL, U. A Uniform model for temporal and versioned object-oriented databases. In: **Temporal Databases: Theory, Design and Implementation**. Redwood City: Benjamim/Cummings, 1993. p.230-247.

Informática
UFRGS

Sistemas de Informação de Escritórios: Um Modelo para Especificações Temporais.

Tese apresentada aos Senhores:

Prof. Dr. Alberto Laender (UFMG)

Profa. Dra. Cláudia Bauzer Medeiros (UNICAMP)

Prof. Dr. Clésio Saraiva dos Santos

Prof. Dr. José Mauro Volkmer de Castilho

Prof. Dr. José Palazzo Moreira de Oliveira

Vista e permitida a impressão.
Porto Alegre, 22 / 06 / 94.

Prof. Dr. José Palazzo Moreira de Oliveira,
Orientador.

Prof. Dr. José Palazzo Moreira de Oliveira,
Coordenador do Curso de Pós-Graduação
em Ciência da Computação.