UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM MODELO DE DADOS PARA AMBIENTES DE PROJETO

por

JAVAM DE CASTRO MACHADO

Dissertação submetida como requisito parcial para
a obtenção do grau de Mestre em
Ciência da Computação

Profa. Lia Goldstein Golendziner
Orientadora

Porto Alegre, dezembro de 1990

CATALOGAÇÃO NA FONTE

Machado, Javam de Castro

Um modelo de dados para ambientes de projeto. Porto Alegre, CPGCC da UFRGS, 1990.

1 v.

Diss. (mestr. ci. comp.) UFRGS - CPGCC, Porto Alegre, BR-RS, 1990.

Dissertação: modelos de dados: sistemas de gerência de banco de dados: ambientes de projeto

À ROSA.

Ao LEVI.

Ao meu Pai.

AGRADECIMENTOS

À Profa. Lia Golendziner pela orientação, amizade, respeito e, mais ainda, pelas contribuições valiosas ao texto desta dissertação. Aos professores Castilho, Tom, Palazzo e Clesio, pelas muitas dúvidas tiradas.

Ao Miguel, pelo trabalho na implementação do protótipo.

À CAPES e ao CNPQ pelo apoio financeiro.

Aos de casa, pelo apoio e incentivo, especialmente minha mãe e meus irmãos Elian e Jonathan.

Ao grande Kita, pelo companheirismo e amizade. Este texto ficou melhor após as suas sugestões e correções.

Ao casal Aliomar e Adja. Foram incontáveis as "sopas" e os "fondues".

Ao Manel, pelo companheirismo e, antecipadamente, pelo trabalho pós defesa.

Aos colegas e amigos que encontrei aqui em POA: Eiji, Marco, Eloi, Walcélio e Keila, Alvaro e Vera, Remis e Alba, Aristeu Dayse, Giovanni e Solange, Deoni e Marlete. À Clélia, pelas aulas de francês e, principalmente, pela amizade.

Aos funcionários Luis Otávio, Margarida, Tânia, Eni, Joice, Mara, Baixinha e Silvânia.

À Rosa pelo amor, carinho, dedicação e incentivo constante durante todo o curso. Ao Levi, por ser uma criança maravilhosa. Ambos são, pela experiência que vivemos juntos, co-autores deste trabalho.

A DEUS.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

A aplicação de projeto é comumente classificada como uma aplicação não-convencional. Como tal, apresenta requerimentos no tratamento dos dados que os bancos de dados comercialmente disponíveis não conseguem satisfazer /SID 80/. A estruturação dos dados dos sistemas que apóiam o projeto de sistemas digitais tem muito em comum com a organização dos dados nos modernos ambientes de desenvolvimento de software /MAC 89/. Este trabalho tem por objetivo propor um modelo de dados capaz de suprir as necessidades de modelagem dos dados dos ambientes de projeto de software e de hardware.

O modelo proposto, chamado de BD_PAC, é uma extensão do modelo de dados do Sistema DAMOKLES. Ele apresenta os conceitos de objetos e relacionamentos organizados em tipos. Um tipo define a estrutura dos seus objetos. Os relacionamentos possibilitam ligações genéricas entre objetos, ou seja, ligações sem significado semântico preciso para o modelo de dados.

O BD_PAC suporta uma coleção de conceitos de abstração /BRO 84/ que organiza e associa os objetos. O conceito de agregação corresponde à noção de propriedade no sentido de composição /MAT 88/. Objetos agregados podem compartilhar objetos componentes. O conceito de generalização suporta a formação de hierarquias de supertipos e subtipos. Nestas hierarquias, os subtipos herdam as propriedades dos supertipos. O conceito de associação permite a criação de conjuntos de objetos. O modelo fornece algumas funções para calcular automaticamente valores dos atributos dos conjuntos de acordo com valores dos atributos dos objetos membros. Todos estes conceitos podem-se compor indistintamente, formando hierarquias mistas em conceitos.

O plano de versões do DAMOKLES também foi modificado. Ele agora reflete as características semânticas dos tipos versionados. Os tipos de objetos podem ser versionados ou não-versionados. Os objetos

de um tipo não-versionado são objetos que mantêm a estrutura declarada no tipo. Por outro lado, cada objeto de tipo versionado é interpretado como um objeto abstrato que agrupa o conjunto de suas versões. As versões dos tipos versionados assumem a estrutura declarada no tipo.

Finalmente, o BD_PAC é implementado tendo como base o sistema de gerência de banco de dados DAMOKLES. Isto significa que, além das características comuns de SGBDs (controle de concorrência, reconstrução de falha), são mantidas todas as facilidades de gerência de dados para ambientes de projeto do DAMOKLES. Por exemplo, suporte para transação de projeto e manutenção de áreas de dados públicas e privativas a projetistas.

ABSTRACT

Design application has been classified as a nonconventional application. As such, it presents data processing requirements that are not fulfilled with the database management systems available commercially /SID 80/. The data structure of the systems that support digital systems design has much in common with the data organization in the moderns software development environments /MAC 89/. This work's goal is to propose a data model that is able to fulfil the data modelling necessities of the software and hardware design environments.

The proposed data model, called BD_PAC, is an extension of the DAMOKLES Design Object Data Model. It supports objects and relationships organized in types. An object type defines the structure of its objects. Relationships allow generic connection between objects, that is, connections that have no semantic meaning to the data model.

BD_PAC supports a collection of abstraction concepts /BRO 84/ that organize and associate objects. The aggregation concept corresponds to the notion of property, in the sense of composition /MAT 88/. Aggregate objects are allowed to share component objects. The generalization concept supports supertype and subtypes hierarchies. In those hierarchies, subtypes inherit the supertypes properties. The association concept supports object set creation. the data model offers some functions to be used in the automatic derivation of set attributes. The derivation is done over the values of the member objects' attributes. All these abstraction concepts may be integrated to form concept mixed hierarchies.

The DAMOKLES versions plan has been also modified. It now reflects the semantic characteristics of versioned types. Object types may be either versioned or non-versioned. The objects of a non-versioned type keep the structured declared for the type. Otherwise, each object of a versioned type is treated as an abstract object that groups its version set. Versions of versioned types assume the structured declared

for the type.

Finally, BD_PAC is implemented over the DAMOKLES database management system. It means that, besides those common characteristics found in DBMS (concurrence control, recovery), facilities specially implemented for design application in DAMOKLES are also kept. For example, support for design transaction and public and private databases.

1 INTRODUÇÃO

A necessidade de automação da produção sempre crescente levou ao surgimento, nos últimos anos, de sistemas de apoio ao processo de desenvolvimento e manufatura de software e de hardware. Por um lado, engenheiros de software perceberam que poderiam utilizar o computador como ferramenta de auxílio na construção e manutenção de sistemas para o próprio computador. Por outro, engenheiros de hardware também viram no computador um instrumento capaz de ajudá-los no desenvolvimento de sistemas digitais. No início, eram usadas ferramentas isoladas que realizavam alguma tarefa específica no desenvolvimento do produto. Apareceram editores diagramáticos especializados em metodologias (Diagrama Fluxo de Dados, Diagrama Estruturado, Editores de "layout"), compiladores de silício, verificadores, etc. Estas ferramentas trabalhavam isoladamente e os resultados de uma não eram compartilhados pelas outras. Na última década, o esforço tem se concentrado em juntar um conjunto de ferramentas já existentes e outras novas que porventura venham a aparecer em um ambiente integrado, capaz de suportar todas as fases de desenvolvimento tanto do produto de software como de hardware.

Para a completa integração dos ambientes, é de vital importância que as ferramentas compartilhem um conjunto de dados comum. Sistemas Gerenciadores de Banco de Dados (SGBD) se tornaram então um dos componentes centrais dos modernos ambientes de desenvolvimento. SGBDs podem oferecer independência de dados e uma interface homogênea a todas as ferramentas que compõem o ambiente. As propriedades de SGBDs tornam mais fácil o desenvolvimento e inclusão de novas ferramentas no ambiente, diminuem a redundância de dados, facilitam o armazenamento e a recuperação de informação em memória secundária, possibilitam o controle de acesso e tolerância a falhas, além de mais uma coleção de outras vantagens /DAT 86/.

Os ambientes de projeto, no plano de tratamento dos dados, reúne um conjunto de características comuns. A complexidade dos dados, o modelo avançado de transações de projeto e o controle do

histórico de versões de objetos são algumas das características básicas. Foi constatado (/SID 80/, /HAS 82/) que os sistemas de banco de dados comercialmente disponíveis não são adequados a este tipo de aplicação, porque não satisfazem às necessidades de organização dos dados nos ambientes de projeto.

Os primeiros ambientes de projeto utilizavam o sistema de arquivos da máquina hospedeira para guardar os dados de projeto. Cada ferramenta gerava os dados de acordo com as suas especificações. Havia, então, a necessidade de procedimentos para transformar os dados gerados por uma ferramenta para que outra pudesse fazer acesso aos mesmos dados. Em seguida, sistemas de bancos de dados foram utilizados para armazenar as informações de projeto. Isto diminuiu a redundância e forneceu interface padronizada de acesso aos dados. No entanto, a estrutura de dados dos ambientes de projeto era complexa, ao mesmo tempo em que surgiam outras necessidades de gerência de dados. Foram propostas, então, extensões aos modelos de dados comercialmente disponíveis (hierárquico, rede, relacional) (/BEN 82/, /HAS 82/, /LOR 82/). Novas propostas têm surgido, seja apresentando características dos modelos semânticos de dados (/BRO 84/, /PEC 88/), seja incorporando propriedades dos sistemas de banco de dados orientados a objetos (/DIT 86a/, /BAN 88a/, /ATK 89/).

O objetivo deste trabalho é prover um sistema de banco de dados capaz de suportar tais ambientes. Tal sistema deve oferecer um modelo de dados avançado com tipos simples, construtores de tipos e mecanismos de abstração; deve fornecer facilidades para o controle de versões de objetos do sistema; deve implementar um mecanismo de transações de banco de dados que suporte, de forma conveniente, as transações de projeto; e, por fim, deve oferecer facilidades para o trabalho em grupo, através de áreas de dados acessíveis a todos os projetistas e áreas de dados privativas.

Este sistema implementa um modelo de dados projetado como uma extensão do modelo de dados do sistema de gerência de banco de dados DAMOKLES. O modelo do DAMOKLES será estendido para

suportar novos conceitos de abstração de dados /MAT 88/.

O DAMOKLES foi projetado para suportar sistemas de projeto de software, portanto muitas das necessidades de dados encontradas nos ambientes de projeto foram levadas em consideração na concepção e construção do DAMOKLES. Entre as principais características do DAMOKLES, podem ser citadas o modelo de dados avançado (objetos complexos, relacionamentos, versões, domínios estruturados) e o suporte para o trabalho cooperativo de vários projetistas em um ambiente distribuído (transações de projeto, múltiplos bancos de dados).

Em /MAC 89a/, foi realizada a modelagem dos dados de um ambiente de desenvolvimento de software, em construção na UFRGS, utilizando-se o modelo de dados suportado pelo DAMOKLES. Em /MAC 89b/, foi feito um estudo sobre os modernos sistemas de projeto, tanto de software como de hardware, no que se refere aos conceitos de abstração de dados presentes nestes sistemas. Ambos os trabalhos levaram à conclusão de que o modelo de dados do DAMOKLES, embora avançado, não suporta adequadamente os conceitos de abstração necessários aos sistemas de projeto atuais. Este trabalho, então, estende o modelo de dados do DAMOKLES, a fim de torná-lo semanticamente mais expressivo e eficaz o bastante para facilitar a modelagem de dados dos ambientes de projeto de software ou hardware.

Um quadro geral para utilização e validação do novo sistema de banco de dados é encontrado na Universidade Federal do Rio Grande do Sul (UFRGS). Ambientes de projeto em desenvolvimento nesta Universidade para as áreas de software e hardware podem usufruir das facilidades oferecidas pelo novo sistema.

No capítulo 2 é descrito o contexto onde se insere este trabalho. São mostradas as propriedades necessárias aos SGBDs para o suporte a ambientes de projeto. Em seguida, é feita uma análise das principais propostas de modelos de dados para estes ambientes.

No capítulo 3 é apresentado, em detalhe, o Sistema DAMOKLES. anfase é dada ao modelo de dados e às características importantes para sistemas de projeto.

O capítulo 4 descreve o novo modelo de dados proposto. São mostrados cada um dos conceitos de abstração suportados pelo modelo e o inter-relacionamento entre eles.

No capítulo 5 é realizada uma pequena explanação sobre o plano de versões em algumas proposta de SGBDs e, logo em seguida, é explicado o plano de versões para o modelo proposto.

O capítulo 6 trata da implementação do protótipo. Descreve como foi estendida a linguagem de definição de dados do DAMOKLES e os reflexos sobre a linguagem de manipulação.

No capítulo 7 é feito um estudo de caso. Os dados dos sistemas AMPLO e AMADEUS - ambientes para suporte a projetos de hardware e de software, respectivamente - são modelados através do novo modelo. Os dados dos mesmos ambientes são modelados utilizando-se o modelo do DAMOKLES. Um quadro comparativo é, em seguida, delineado.

O capítulo 8 conclui o trabalho e propõe alguns temas para trabalhos futuros.

Acompanham 4 anexos.

2 BANCO DE DADOS PARA AMBIENTES DE PROJETO

Ambientes de projeto são sistemas interativos que fornecem facilidades para a criação de produtos. Pesquisas nessa área têm identificado a gerência de dados como um dos componentes mais importantes de tais sistemas. Este capítulo apresenta uma visão geral das propostas para o armazenamento de informações em sistemas de CAD/CAM e CASE.

Os primeiros ambientes de projeto utilizavam o sistema de arquivos do sistema operacional hospedeiro como repositório de dados. Os sistemas de CAD e CASE foram, então, evoluindo através da incorporação crescente dos conceitos de sistemas de bancos de dados para o gerenciamento das informações. As propostas atuais de sistemas de projeto incluem tecnologia emergente da área de banco de dados - modelos semânticos e modelos orientados a objetos - como quadro geral para o armazenamento e controle de dados de projeto.

A próxima seção descreve algumas características dos sistemas de apoio a projeto que utilizavam sistemas de arquivos e explica porque é muito melhor utilizar a tecnologia de banco de dados. A seção seguinte dá um passo adiante afirmando que os sistemas de bancos de dados comercialmente disponíveis estão longe de satisfazerem às necessidades dos modernos ambientes de projeto. Em seguida, são descritas algumas propriedades dos modelos de dados avançados e como estes sistemas se aplicam adequadamente à aplicação de projeto. A partir dai, são apresentadas as características gerais de alguns dos modernos sistemas de apoio a projeto encontrados na literatura tanto de software quanto de hardware.

2.1 Sistemas de Arquivos

Sistemas de projeto são formados, em geral, por um conjunto de ferramentas automáticas ou semi-automáticas que

auxiliam projetistas em seu trabalho /KAT 86a/. As ferramentas manipulam os dados gerando-os, analisando-os e os transformando. Nos primeiros ambientes, os dados eram gerados independentemente por cada uma das ferramentas e posteriormente armazenados em arquivos. O conteúdo de um arquivo só era interpretado pela ferramenta que o gerava, de forma que nenhuma das outras ferramentas do ambiente tinha acesso às informações daquele arquivo. O formato dos dados de um arquivo era transformado para outro formato a fim de que outras ferramentas pudessem utilizar os dados e continuar o processo de projeto (figura 2.1). A lista abaixo relaciona algumas desvantagens dessa abordagem

- O sistema de arquivos não tem conhecimento do conteúdo de cada arquivo. A semântica e a estrutura dos dados ficam armazenadas nas ferramentas, portanto os dados se tornam totalmente dependentes das ferramentas;

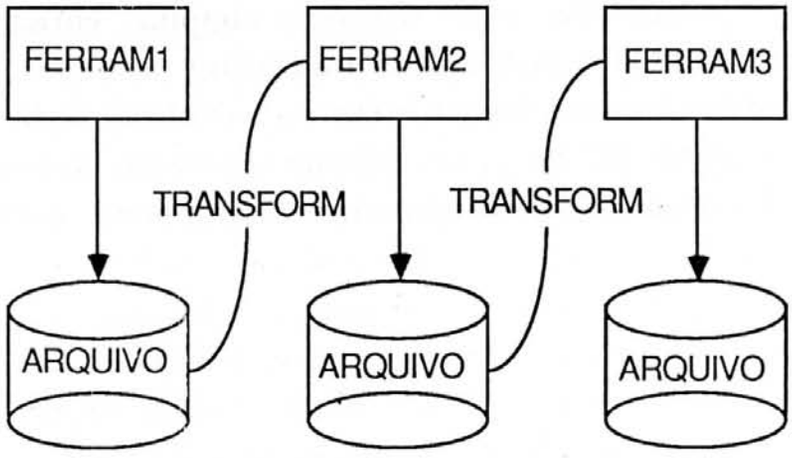

Figura 2.1: Arquitetura dos primeiros ambientes

- O armazenamento dos dados em arquivos específicos para cada ferramenta gera uma grande redundância de dados. Essa redundância traz consigo problemas de consistência da base de dados;

- O compartilhamento dos dados é feito a nível de arquivo, resultando

em bloqueio, muitas vezes desnecessário, de todos os dados de um arquivo;

- Funções inerentes à estrutura de dados precisam ser implementadas em todas as ferramentas para proporcionar acesso eficiente e correto aos dados.

Por outro lado, sistemas de bancos de dados ao mesmo tempo que delimitam uma abordagem comum e integrada para a gerência de dados, ainda proporcionam uma série de facilidades que tornam bem mais simples a manutenção das informações de projeto. Um banco de dados integrado de projeto é necessário para que uma coleção de ferramentas possa ser formada em um sistema de projeto. O sistema de banco de dados organiza informações de objetos e versões evolucionárias. Estes sistemas controlam acesso concorrente aos dados e garantem que os dados podem sobreviver a falhas do sistema. Colocar todas as informações de projeto sob a responsabilidade de um único sistema gerenciador de dados torna a consistência do projeto mais facilmente mantida. Quando dependências entre as partes do projeto são explícitas, ramificações de mudanças podem ser descobertas e propagadas de maneira controlada /KAT 83/. O uso de Sistemas Gerenciadores de Banco de Dados proporciona maior integração dos dados e interface de acesso padronizada a todas as ferramentas do ambiente. além dos benefícios normais que SGBDs geralmente fornecem: independência do dados, controle de consistência, segurança, reconstrução em caso de falhas, redução de redundância, manutenção de integridade, entre outros /DAT 86/. A figura 2.2 mostra a arquitetura de um sistema de apoio a projeto que utiliza tecnologia de banco de dados.

A atividade de projetar é invariavelmente uma atividade complexa e mecanismos de abstração são o meio mais eficiente utilizado pelo homem para tratar problemas complexos /ATW 85/. Certamente, uma das facilidades mais importantes fornecidas por SGBDs aos ambientes de projeto é a capacidade de abstração da estrutura interna dos dados através dos conceitos de modelagem

suportados pelo modelo de dados do SGBD. O projetista, através da utilização desses conceitos, determina a estrutura externa dos dados definindo o *esquema* de banco de dados. O esquema deve poder mais tarde ser modificado (evolução de esquema) sem que haja prejuízo para a aplicação que faz acesso aos dados. A utilização dos conceitos de modelagem permite que projetistas possam expressar, em termos lógicos no banco de dados, parte da semântica da aplicação e ainda proporciona um meio valioso para a documentação do ambiente.

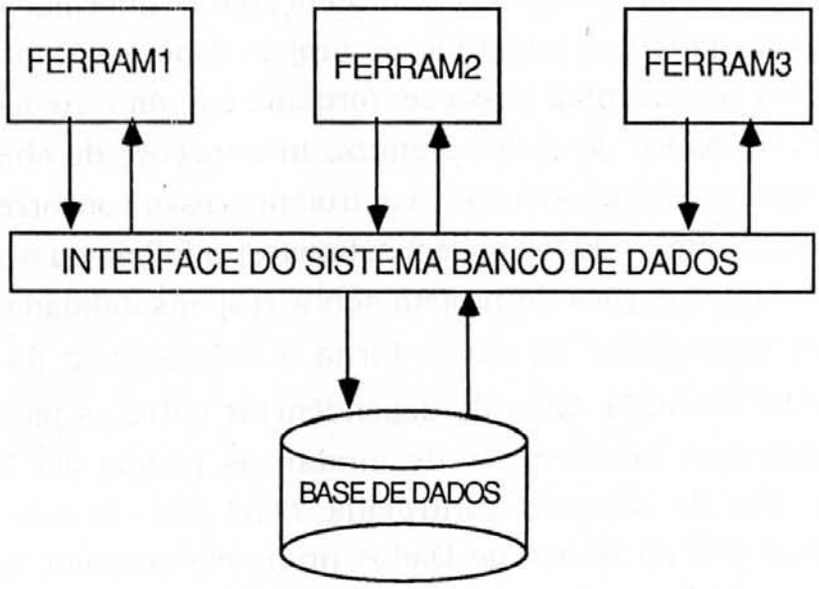

Figura 2.2: Arquitetura de ambientes modernos

2.2 Fragilidade dos Modelos Convencionais

O suporte de SGBDs aos ambientes de projeto trouxe ganhos significativos em relação aos sistemas de arquivos outrora utilizados. Por outro lado, os modelos de dados comercialmente disponíveis no mercado, chamados clássicos (Hierárquico, Rede e Relacional) não se mostram totalmente adequados para satisfazer as necessidades dos modernos ambientes de projeto em vários aspectos. Os sistemas de

gerência de bancos de dados correntes, desenvolvidos com aplicações comerciais em mente, não satisfazem os requerimentos de modelagem de dados emergentes da natureza iterativa e tentativa do processo de projeto e da natureza multiaspecto dos objetos de projeto que são compostos de outros objetos /KET 87/.

A lista a seguir (/SID 80/, /KAT 83/, /HAS 82/, /DIT 86b/) inclui algumas das propriedades dos SGBDs correntes que não satisfazem totalmente as necessidades de sistemas de projeto:

- Suporte a um conjunto fixo e muito simples de tipos de dados. Os dados somente podem ser representados por registros e conjuntos de registros. Não existe a possibilidade de formar novos tipos através de construtores de tipos. Os domínios dos atributos comportam somente valores atômicos como inteiros, caracteres ou booleanos;

- O modelo de transação não é adequado às transações interativas de projeto. Fazer a restauração da base de dados até o início de uma transação após uma falha pode destruir o trabalho de dias ou mesmo meses de um projetista. Deve ser possível a uma transação de projeto manter estado inconsistente da base de dados entre diversas sessões de trabalho de um projetista;

- O mecanismo de verificação de restrições de integridade é muito simplificado. Restrições de domínio, unicidade de chave e integridade referencial não cobrem todas as necessidades de controle de restrições de manipulação de dados de projeto;

- Mecanismo rígido de definição dos dados. Dados de projeto evoluem em sua estrutura com a atividade de projeto. Não existe qualquer facilidade para a evolução na definição de um esquema em sistemas correntes;

- Não menos importante do que os já citados é o aspecto de performance. A manipulação dos dados de projeto nos SGBDs comerciais, em geral, exige um grande volume de operações no banco de dados, degradando a performance do sistema na busca de

um objeto de projeto.

Objetos de projeto (não registros individuais) são as unidades de recuperação e armazenamento, e são a base para verificar integridade semântica de modificações, inserções e exclusões de objetos complexos. SGBDs atuais não suportam o tratamento adequado de objetos complexos /BAT 85/. A estrutura complexa dos dados tratados nos ambientes de projeto demanda novas técnicas de otimização para armazenamento de dados (ex. "clustering").

Constatada a dificuldade de utilização dos modelos clássicos de bancos de dados para suportar completa e eficientemente ambientes de projeto, foram propostas inicialmente extensões aos modelos existentes e posteriormente modelos de dados completamente novos.

2.3 Modelos de Dados para Ambientes de Projeto

Os dados de projeto são grandes em tamanho e complexos em organização. Tal estrutura de dados não é de gerenciamento simples, nem tampouco pode ser deixada manipular livremente pelas ferramentas. O uso de Sistemas Gerenciadores de Banco de Dados é fundamental para a integração do ambiente e manutenção da integridade dos dados. O modelo de dados de um SGBD incorpora um conjunto de conceitos para estruturação da informação, um conjunto de operações genéricas e um conjunto de regras implícitas de restrições de integridade. Para ambientes de projeto, esse modelo deve ser poderoso o bastante para acomodar uniforme e eficientemente toda a informação manipulada no ambiente.

Os modelos de dados para ambientes propostos na literatura incorporam avanços realizados nas áreas de Modelos Semânticos de Dados (/PEC 88/, /HUL 87/) e Banco de Dados Orientados a Objetos (/DIT 86a/, /BAN 88a/, /ATK 89/). Estruturas de dados

complexas podem ser formadas pela aplicação de construtores de tipos; os modelos suportam relacionamentos semânticos como agregação /BAT 85/, generalização /SMI 77/ e associação /HAM 81/; uniformidade no tratamento da informação através do conceito de objeto; armazenamento e controle das operações (métodos) aplicáveis a cada tipo (classe) de objeto. Estes são apenas alguns avanços importados para os ambientes de projeto das pesquisas nas áreas citadas acima.

Aplicações de projeto necessitam estruturar dados hierarquicamente, suportar múltiplas representações de objetos de projeto, manter alternativas e versões de projeto, ajudar projetistas a cooperarem entre si e interagir como um time, suportar projeto remoto em estações de trabalho com armazenamento no servidor de banco de dados, e manter a consistência do projeto. Abaixo são listadas algumas características que necessitam estar presentes em um SGBD para ambientes de projeto /HAS 82/, /KAT 83/, /DIT 86b/, /CHU 86/, /MUL 87/:

Conceitos de Abstrações: As informações de projeto são geralmente organizadas dentro de hierarquias de composição e classes de objetos inter-relacionadas. Por exemplo, programas compõem-se de módulos, que podem ser formados pela junção de módulos mais simples, que por sua vez são compostos por comandos. O modelo de dados deve ser capaz de explicitamente suportar esta estrutura de composição e classificação dos objetos de projeto;

Atributos Estruturados e de Comprimento Variável: Atributos de objetos de projeto podem ser representados por árvores, listas, conjuntos, matrizes e registro. Há ainda a necessidade de se armazenar grande quantidade de informação não-estruturada em um objeto (campo longo). Este são utilizados para guardar dados sem significado semântico para o banco de dados, mas que são manipulados diretamente pelas ferramentas do ambiente;

Projeto Evolucionário: Atividade de projeto é em grande parte exploratória. O SGBD deve suportar diversas *alternativas*, *versões* e *revisões* de um mesmo objeto na base de dados. Posteriormente

deverá ser possível definir uma *configuração* consistente para efeito de prototipação ou simulação do produto de projeto;

Transações: Transações de projeto envolvem grande volume de dados por um longo período de tempo. Mecanismos convencionais de bloqueio não são adequados para esse modelo de transações /IOC 89/. Em caso de falha, o subsistema de reconstrução não deve retornar a base de dados ao estado inicial da transação, mas até pontos determinados implicitamente pelo sistema ou explicitamente pelo projetista;

Controle de consistência: restrições de integridade complexas devem ser aplicadas a tipos de objetos diferentes e em tempo variado. O SGBD deve permitir a definição descritiva e procedural de restrições de consistência;

Biblioteca de Objetos: Um objeto é projetado uma única vez e quando pronto pode ser usado na composição de outros objetos. O SGBD deve manter uma biblioteca de objetos acabados e suportar operações de cópia desses objetos para áreas de trabalho de projetistas. Objetos considerados acabados poderão ser incorporados à biblioteca, portanto colocados a disposição da equipe;

Arquitetura: Deve fazer parte do sistema um repositório central que comporta a biblioteca geral do ambiente e área de dados privativa a cada projetista colocada possivelmente em estações de trabalho. As estações ligam-se a um servidor onde é armazenado o banco de dados geral do projeto;

Sistemas documentados na literatura apresentam algumas dessas características, mas nenhum deles oferece todas elas como atrativo. Muitos ambientes enfatizam o aspecto de controle de versões e configurações (/CAM 86/, /BEL 87/), outros reforçam o conceito de transação e controle de processo de projeto (/KUO 86/), outros ainda enfatizam o poder de modelagem dos dados do ambiente (/RUD 86/, /YAN 88/, /DEB 89/).

2.5 SGBDs para Ambientes de Projeto de Hardware

Os ambientes de projeto de hardware suportam a concepção de sistemas digitais através de um conjunto de ferramentas de síntese, análise e gerência de informação /HAY 88/, /MUL 87/. Ferramentas de síntese assistem a equipe de projeto na concepção de sistemas digitais - editores, compiladores, etc. Ferramentas de análise verificam se o projeto é bem formado e age como esperado, com o desempenho necessário - verificadores de regras, simuladores, etc /BUC 85/, /CHE 88/. As funções de gerência de informação são entregues a Sistemas Gerenciadores de Bancos de Dados. Eles organizam a estrutura dos dados de projeto e são o objeto da discussão que se segue.

Ambientes de Projeto de Hardware dedicam atenção especial aos mecanismos de gerência de objeto de projeto /KEM 87/. Propostas como a de Katz /KAT 83/, /KAT 86a/, Batory /BAT 84/, /BAT 85/ e Ketabchi /KET 87/ implementam um conjunto variado de relacionamentos pré-definidos entre objetos de projeto que determina a estrutura organizacional dos objetos na base de dados. Relacionamentos de equivalência entre representações, definição de interface de objetos complexos, dinamicidade no processo de instanciação de objetos compostos, estas são todas características que norteiam as propostas de banco de dados para ambientes de hardware.

A proposta de Ketabchi /KET 87/ enfatiza o aspecto de evolução de esquemas na formação de objetos compostos. Um objeto composto é formado passo a passo pelo desenvolvimento de seu "template" em relação aos demais "templates" da base.

Os dados gerados pelas ferramentas dos ambientes são ditos *puros*. É o espaço representacional definido em /KAT 86/. As informações relativas à organização dos dados puros são chamadas de dados estruturais. É o espaço estrutural definido também em /KAT 86/. Alguns Sistemas de Gerência de Dados fornecem a capacidade de manipulação de objetos puros de projeto, enquanto outros deixam a

responsabilidade para as ferramentas do ambiente. Os Sistemas KRISYS /DEB 89/, DAMASCUS /DIT 87/, /MUL 87/ e ADAM /AFS 85/, /AFS 86/ suportam o controle e manipulação tanto de objetos puros, como também apresentam a capacidade para a gerência de dados do espaço estrutural de projeto. As propostas de Katz /KAT 86/, Batory /BAT 85/, Wolf /WOL 86/, /WOL 88/ e Ketabchi /KET 87/ tratam apenas de relacionamentos estruturais entre representações de objetos.

A maioria dos ambientes apresentam um mecanismo de controle de versões para os dados de projeto. O mesmo acontece com a equivalência entre diferentes níveis de abstração nas representações dos objetos.

Objetos de projeto são modelados nestes ambientes através de conceitos de abstração. Três conceitos são encontrados com mais frequência: agregação molecular /BAT 85/, generalização /SMI 77/ e associação /HAM 81/. Agregação molecular permite a composição de objetos por outros mais simples. O conceito subentende a idéia de juntar partes para formar um todo. Dada uma coleção de objetos com propriedades comuns, o conceito de generalização possibilita a formação de um outro objeto de nível mais alto, onde as propriedades comuns são enfatizadas. Estas propriedades passam a fazer parte do objeto de nível mais alto, ao mesmo tempo que são, por herança, válidas também para os objetos generalizados. A associação permite a criação de conjuntos de objetos.

Objetos complexos são invariavelmente suportados por todos os ambientes. Agregação molecular reflete bem a estrutura de desenvolvimento de projeto de hardware. Generalização, classificação e associação não são tão comuns quanto a agregação, mas principalmente as duas primeiras aparecem com alguma freqüência nos ambientes. A propostas de Katz /KAT 86/ e Batory /BAT 85/ e o Sistema DAMASCUS /MUL 87/ suportam apenas o conceito de agregação molecular. Batory /BAT 85/ suporta o conceito de generalização, mas restringe sua aplicação apenas às versões de objetos, inclusive com a herança de atributos. No caso do DAMASCUS

/MUL 87/, o sistema apresenta relacionamentos gerais que podem ser utilizados para definir tanto generalizações (sem herança) quanto associação, embora a semântica não possa estar no banco de dados. As demais propostas - Ketabchi /KET 87/, KRISYS /DEB 89/, Wolf /WOL 88/ e o Sistema ADAM /AFS 86/ - suportam tanto a agregação, quanto a generalização. O conceito de associação não aparece explicitamente em qualquer proposta, salvo no Sistema KRISYS /DEB 89/, mas pode ser modelado via relacionamento geral suportado por alguns ambientes.

2.5 SGBDs para Ambientes de Projeto de Software

Como os ambientes de projeto de hardware, os de projeto de software também auxiliam projetistas no desenvolvimento de componentes de projeto. No caso específico de software, estes componentes são especificações textuais e gráficas, programas, massas de testes, documentação de sistemas e outros /PEN 88/, /PEN 89/. Ambientes de software, a exemplo dos de hardware, também são formados por um conjunto de ferramentas em um ambiente integrado /ROS 89/.

A discussão que se segue relaciona algumas propostas para o gerenciamento de dados de ambientes de projeto.

A modelagem dos ambientes de desenvolvimento de software apresenta um conjunto de características comuns e ainda um conjunto de propriedades peculiares a cada ambiente.

Os conceitos de abstração como agregação, generalização e associação são quase que unanimemente suportados pelos sistemas gerenciadores de objetos. A proposta de Rudmik /RUD 86/ e o Sistema CACTIS /HUD 86/ /HUD 87/, /HUD 88/ contêm esses conceitos. O Sistema DAMOKLES /DIT 86b/, /REH 88/, /ABR 88/ tem somente a agregação enquanto que o Sistema PCTE /GAL 87/ tem somente a

generalização. Sistemas como o DAMOKLES /ABR 88/, PCTE /GAL 87/ e PSE /KUO 86/ suportam relacionamentos genéricos entre objetos, possibilitando a definição de qualquer dos relacionamentos semânticos, embora a semântica tenha que ficar na aplicação. A maioria dos ambientes utiliza esses conceitos para manipular objetos de projeto, estruturando as informações de gerência de dados dos ambientes. Os objetos do espaço representacional são, em geral, armazenados em campos longos - atributos que podem guardar grande volume de informação não estruturada. Estes dados não têm significado semântico para o gerenciador de objetos, mas têm para as ferramentas que os manipulam.

anfase especial precisa ser dada ao Sistema CACTIS /HUD 88/ e à proposta de Horwitz /HOR 86/. Esses ambientes incorporam conceitos da área de compiladores a bancos de dados, a fim de suportar melhor o armazenamento e manipulação de programas e documentos com gramática. O modelo do CACTIS /HUD 88/ permite a definição de atributos derivados e relacionamentos direcionados. Estas características facilitam a modelagem de objetos tratados em gramática de atributos, teoria usada com frequência nos modernos ambientes de software. O modelo da Horwitz /HOR 86/ é pobre em termos de modelagem semântica, mas poderoso na representação de objetos que utilizam estruturas gramaticais. Horwitz associa o modelo relacional à teoria de gramática de atributos de forma eficiente e elegante. Os atributos associados às produções da gramática são armazenados em relações do modelo relacional. O armazenamento de informações de ,rvores Sintáticas Abstratas torna persistentes dados que só estariam disponíveis no momento em que a árvore estivesse montada em memória. Este aspecto torna possível a recuperação de informações internas de programas através da linguagem de consulta do relacional.

Muitas propostas deixam de lado uma infinidade de características importantes para ambientes de projeto. O controle de versões é pouco citado pelas propostas, salvo os sistemas PSE /KUO 86/, e DAMOKLES /ABR 88/, este último apresentando um sofisticado mecanismo de versões. Relacionamentos especiais como equivalência

de representações e configurações também são negligenciados pela maioria das propostas.

Existem ainda na literatura várias propostas de sistemas de bancos de dados orientados a objetos que têm aplicação direta em ambientes de projeto (/ATW 85/, /BAN 87/, /LEC 88/, /BAN 88b/, /ZDO 84/). A maioria desses sistemas tem como principal característica o suporte à manipulação de objetos, desde os mais simples até os mais complexos, como uma unidade de processamento, através do paradigma de orientação a objeto /GOL 83/. Os sistemas ORION /BAN 87/ e O_2 /LEC 88/ implementam sofisticados mecanismos para o controle de versões de objetos, enquanto que o OMS /ZDO 84/ e o OM /ATW 85/ apresentam mecanismo de versões mais simples, embora ambos se utilizem do conceito de percolação (ver capítulo 5 - O Plano de Versões) para a propagação de atualizações em versões.

Estes sistemas foram projetados para serem sistemas de propósito geral, portanto muitas necessidades inerentes à aplicação de projeto (ver seção 2.3) não são plenamente satisfeitas. Por outro lado, o poder de modelagem de dados fornecido por esses sistemas tem sido muito maior do que o oferecido pelos modelos de dados de sistemas de bancos de dados dedicados à aplicação de projeto.

3 O SISTEMA DAMOKLES

O sistema DAMOKLES ("DAtabase Management System Of KarLsruhe for Environments for Software Engineering") é um Sistema Gerenciador de Banco de Dados (SGBD) que foi desenvolvido na Alemanha Federal como parte do projeto UNIBASE - pesquisa e desenvolvimento de ambientes integrados de engenharia de software. O projeto do Sistema DAMOKLES foi guiado por três objetivos:

- Funcionalidade a fim de fornecer suporte eficiente aos requisitos de Ambientes de Desenvolvimento de Software (ADS);
- Generalidade a fim de poder ser aplicado em ambientes de projeto em geral e não somente de software;
- Simplicidade com a implementação de poucos, mas suficientes, conceitos, facilitando o aprendizado e a manipulação do sistema.

Este capítulo descreve o sistema DAMOKLES, seu modelo de dados, propriedades especiais necessárias a um SGBD para ambiente de projeto e alguns aspectos de utilização do sistema.

As seções seguintes tratam das principais características do DAMOKLES. Inicialmente, o modelo de dados será mostrado em detalhe. Objetos, versões, relacionamentos, atributos e restrições de integridade implícitas são os tópicos abordados. A seção seguinte descreve propriedades do DAMOKLES que não fazem parte do modelo, mas que são igualmente importantes para ambientes de projeto. Transações longas e hierarquias de bancos de dados são algumas dessas propriedades. Ao final, aspectos de operacionalidade do SGBD mostram como podem ser utilizados os conceitos definidos no modelo.

3.1 O Modelo de Dados

O modelo de dados do DAMOKLES, chamado de DODM

("Design Object Data Model"), é uma extensão do Modelo Entidade/Relacionamento /CHE 76/. Possui, além das facilidades do modelo base, os conceitos de *objetos estruturados e versões*, características imprescindíveis a um modelo de dados para ambientes de projeto /KAT 86/. O DODM se enquadra na classificação de /DIT 86a/ como um modelo estruturalmente orientado a objeto.

3.1.1 Objetos

Objetos são unidades autocontidas do universo de discurso /ABR 88/. Eles são organizados no sistema de acordo com a definição dos tipos de objetos de um esquema DODM. Os objetos são caracterizados pelas propriedades que possuem (figura 3.1). O modelo apresenta dois tipos de propriedades de objetos:

Propriedades Descritivas: Representadas pelos atributos que são mapeados para domínios implícitos ao sistema. Os atributos podem ser *atômicos*, por exemplo, inteiros, booleanos, reais, cadeias de caracteres, ou podem ser *estruturados* como vetores e registros;

Propriedades Estruturais: conjunto de subobjetos que participam da formação de um objeto complexo. Um relacionamento implícito de composição é mantido, determinando a estruturação de objetos em hierarquias (figura 3.1).

De acordo com a estruturação, os objetos podem ser simples ou estruturados.

Objetos Simples: são formados somente pela parte descritiva de objetos;

Objetos Estruturados: também chamados de objetos complexos, permitem a representação de objetos compostos por sub-objetos. Objetos estruturados são formados por propriedades descritivas e por propriedades estruturais (figura 3.1).

Um conjunto (possivelmente unário) de atributos pode ser especificado como *chave do objeto* tendo como base a restrição de integridade de unicidade de chave, descrita em /DAT 86/. Os objetos também possuem *Identificador Único de Objeto* (OID) - um tipo de *surrogate* associado a um objeto no momento de sua criação.

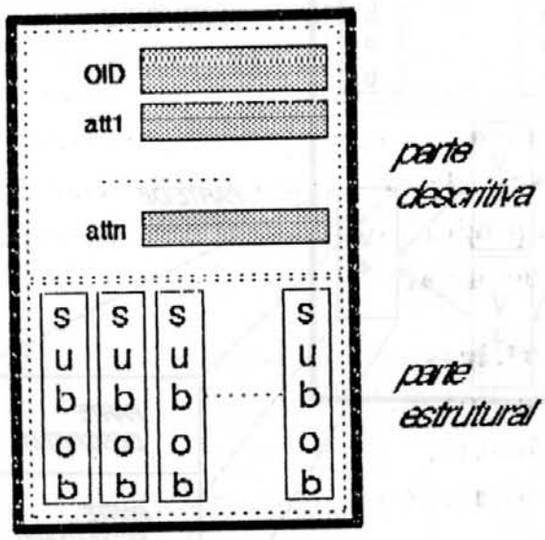

Figura 3.1: Estrutura de um objeto

O modelo suporta a definição de *objetos recursivos* e *hierarquias de objetos sobrepostas*. Direta ou indiretamente, um tipo de objeto pode estar presente na sua própria parte estrutural, facilitando a criação de objetos que se compõem de outros objetos do mesmo tipo. Embora a recursão ocorra na definição de tipos de objetos, ela não pode ocorrer a nível de objetos, um objeto não pode fazer parte de si próprio. A sobreposição de objetos ocorre pela presença de um mesmo objeto simples ou complexo na parte estrutural de mais de um objeto composto. Assim, o sistema não restringe as hierarquias de objetos à forma de árvore, mas também suporta hierarquias na forma de grafos.

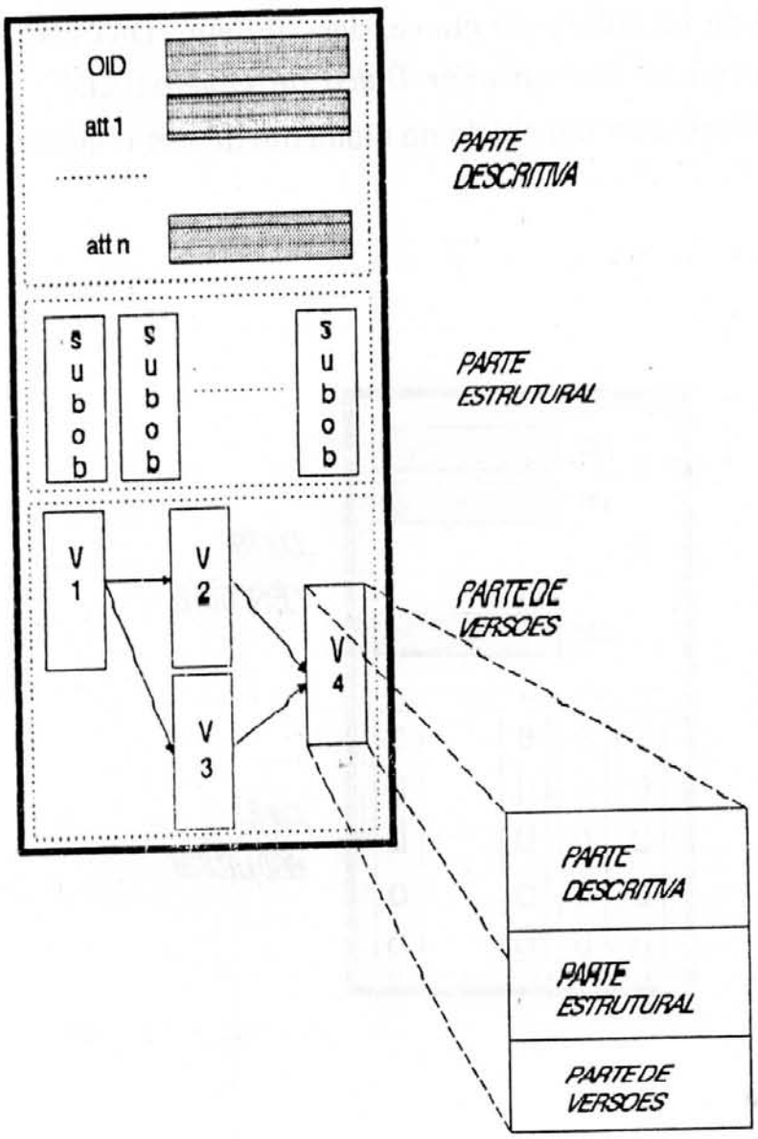

Figura 3.2: Estrutura de um objeto complexo com versão

3.1.2 Versões

Versões são diversas ocorrências de um mesmo objeto que coexistem numa base de dados e representam diferentes estados do mesmo *objeto semântico* /DIT 87/. A figura 3.2 mostra a estrutura de um objeto complexo com versão. O conceito de versão no DODM

as seguintes características:

- Assim como os objetos, versões podem ter propriedades descritivas e estruturais, podem participar de relacionamentos, podem ter versões e também possuem identificadores únicos gerados pelo sistema (figura 3.2);
- As versões pertencem a exatamente um objeto, chamado de "objeto genérico";
- Referências podem ser feitas ao objeto como um todo ou às suas versões individualmente;
- Entre as versões de um objeto genérico, um relacionamento implícito predecessor-sucessor é mantido. Este define uma ordem parcial - chamada de *grafo de versões* - sobre o conjunto de versões do objeto genérico. O relacionamento pode ser opcionalmente definido como:

  Linear: Qualquer versão tem, no máximo, um predecessor e um sucessor;

  Árvore: Qualquer versão tem um predecessor mas pode possuir um número arbitrário de sucessores;

  Acíclico: Qualquer versão, salvo a primeira, tem um número arbitrário de predecessores e sucessores.

Versões são, na realidade, tratadas como objetos. Aquelas diferem destes pelo relacionamento implícito do grafo de versões, porque dependem da existência de um objeto genérico. Cada elemento do conjunto de versões de um objeto tem que ter obrigatoriamente a mesma estrutura, cuja definição aparece na descrição do tipo de objeto versionado. A manipulação das versões pode ser feita individualmente ou em grupo, através do objeto genérico.

3.1.3 Relacionamentos

Os relacionamentos do DODM são um tipo especial de objeto. Assim como objetos, relacionamentos podem possuir atributos, podem fazer parte da composição de objetos complexos e possuem identificador único atribuído pelo sistema, mas não são permitidos "relacionamentos estruturados".

Os relacionamentos associam os objetos numa relação n-ária ($n \geq 1$), ou seja, um objeto pode se relacionar com vários outros. Os objetos participantes são opcionalmente identificados por *papéis* que cada objeto exerce no relacionamento. A *cardinalidade* dos objetos pode ser determinada na definição do esquema para no mínimo um ou no máximo um. Os relacionamentos podem ser estabelecidos entre níveis arbitrários de hierarquias de objetos, inclusive associando objetos genéricos ou versões individuais.

3.1.4 Atributos

Os atributos compõem a parte *descritiva* de objetos, relacionamentos e versões (figura 3.2). O DAMOKLES fornece um conjunto de *domínios* pré-definidos de atributos e ainda uma coleção de construtores de tipos que possibilitam a declaração de atributos estruturados. Entre os pré-definidos estão os mais usuais como inteiros, reais, caracteres, booleanos e campos longos, enquanto que os construídos podem ser cadeias de caracteres, vetores de domínios pré-definidos, registros e uniões (*union* da linguagem C /KER 78/).

Um tipo especial de atributo, chamado de campo longo ("Long Field"), é fornecido pelo modelo. Um campo longo tem como objetivo o armazenamento de informação não estruturada de comprimento arbitrário. Embora a estrutura interna de um campo longo não seja conhecida pelo sistema, este provê operadores especiais que facilitam a manipulação desse tipo de atributo.

3.1.5 Restrições de Integridade Implícitas

Restrições de integridade possibilitam a manutenção de estado sempre consistente do banco de dados, desde que estados internos às transações não sejam considerados. O DODM possui três tipos de restrições implícitas ao modelo: unicidade de chave, integridade referencial e cardinalidade de relacionamentos.

O DAMOKLES garante unicidade de valores para um atributo, definido como chave de um tipo de objeto. Esta chave, comumente chamada de chave externa, não tem qualquer conotação de referência interna para o DAMOKLES e fica sob o controle do usuário. O modelo apenas verifica, no momento da inclusão de um objeto, a existência de alguma outra ocorrência com o mesmo valor de chave e, caso a verificação seja afirmativa, a inclusão é rejeitada. Na versão disponível para a implementação deste trabalho (Versão 2.0), o controle de consistência para unicidade de chave externa não está implementado.

Integridade referencial para relacionamentos determina a propagação de exclusão de objeto até aos relacionamentos nos quais o objeto participa. Assim se um objeto participante de um relacionamento for removido da base de dados, o relacionamento automaticamente também o será.

```
OBJECT TYPE ACOES
  ATTRIBUTES
    simb  : NOME;
    codigo : LONG_FIELD
  AT LEAST ONCE ( P_AC )
END ACOES;
```

Figura 3.3: Exemplo de cardinalidade

A cardinalidade de um tipo de objeto em um relacionamento restringe o número de vezes que cada objeto do tipo pode aparecer no relacionamento. Quando não há restrição explícita para o tipo, cada objeto pode ter N ocorrências no relacionamento, com N variando de zero a infinito, conceitualmente. Por outro lado, o usuário pode limitar a participação de objetos para "AT MOST ONCE" ou "AT LEAST ONCE". No primeiro caso, o sistema garante que no máximo uma ocorrência do objeto pode estar associada ao relaciomanento através do papel restringido. No segundo caso, pelo menos uma ocorrência do objeto tem que estar no relacionamento para que o banco de dados esteja em um estado consistente. A figura 3.3 mostra uma declaração exemplo de um tipo de objeto. No exemplo, cada objeto do tipo ACOES tem que participar do relacionamento P_AC pelo menos uma vez para que o banco de dados esteja consistente.

3.2 Outras Características

3.2.1 Bancos de Dados Múltiplos

O sistema DAMOKLES foi originalmente projetado para atender a sistemas de auxílio no projeto de software. Este tipo de atividade envolve, geralmente, um grupo de projetistas trabalhando cooperativamente para formar um produto. Cada projetista desenvolve sua parte do produto que, uma vez terminada, deve se tornar disponível para os demais projetistas a fim de ser utilizada. Muitas vezes é conveniente também manter uma biblioteca de produtos acabados, acessíveis a todos os projetistas da organização para o possível uso no desenvolvimento de novos produtos. No sentido de suportar atividades de projeto cooperativo, o DAMOKLES permite a manutenção de bancos de dados logicamente separados. Desta maneira, projetistas podem possuir uma área de trabalho pessoal, chamada comumente de banco de dados privativo, onde eles podem armazenar os produtos de software que estão desenvolvendo. Posteriormente, este produto pode ser repassado para uma outra área de trabalho, chamada de banco de dados

de grupo, acessível a todos os membros do mesmo grupo de projetistas, que agora podem utilizar o produto em seu trabalho. Esta característica do DAMOKLES possibilita a criação de bancos de dados para os vários grupos de projetistas que desenvolvem um produto de engenharia. Por sua vez, cada projetista pode ter uma área de trabalho privativa, sob sua responsabilidade. Os diversos bancos de dados DAMOKLES estão sujeitos às seguintes regras:

- Um objeto é membro de exatamente um único banco de dados. Todavia, cópias são permitidas em outros bancos de dados;
- Objetos de bancos de dados diferentes podem participar de um relacionamento, mas a parte descritiva do relacionamento deve pertencer a somente um dos bancos de dados envolvidos;
- Os efeitos das operações da DML estão restritos a um único banco de dados, de forma que eles não podem ser automaticamente propagados para outros bancos de dados;

A fim de manter o controle de acesso aos vários bancos de dados, o DAMOKLES suporta a entrada de identificadores para usuários e grupos de usuários. Estes identificadores são associados aos bancos de dados, caracterizando sua possessão pelos usuários e grupos. Um usuário pode pertencer a vários grupos e ainda grupos podem ser membros de outros grupos. Para os bancos de dados privativos, somente o dono tem acesso irrestrito aos dados. No caso do banco de dados pertencer a um grupo, o direito de acesso segue algumas regras listadas a seguir. O sistema oferece quatro tipos de direito de acesso:

Leitura: o usuário pode somente ler o banco de dados;

Escrita: o usuário pode atualizar o banco de dados;

Transferência: o usuário pode transferir ou copiar objetos no banco de dados;

Projeto: o usuário pode realizar operações de requisição ("checkout") e liberação ("checkin") de objetos no banco de dados.

Esses tipos de direito de acesso relacionam usuários aos bancos de dados da seguinte forma:

- O usuário dono tem todos os direitos de acesso sobre os seus bancos de dados;
- O direito de escrita é exclusivo do proprietário do banco de dados;
- É concedido o direito de leitura a todos os membros do grupo que possui o banco de dados;
- A transferência é possível a todos os membros do grupo proprietário, mas ela só pode ocorrer para banco de dados cujo dono também seja membro do mesmo grupo. Por exemplo, transferir objetos de um banco de dados privativo para um banco de dados do grupo, público a todos os projetistas do mesmo grupo;
- O direito de projetar (requisitar e liberar) é fornecido a todo membro do grupo proprietário.

A figura 3.4 mostra um exemplo dos direitos de acesso explicados acima. O usuário U1 pode ler os bancos de dados D1 e D2 porque é membro direto do grupo G2 e indireto do grupo G1. Ele também pode fazer operações de requisição e liberação no banco de dados D2. Transferências podem ocorrer entre bancos de dados dos membros de um mesmo grupo. Por conseguinte, objetos podem ser transferidos entre D2, D4 e D5.

3.2.1.1 Operações e Transações Longas

O DAMOKLES suporta uma coleção de operadores para o movimento de objetos entre bancos de dados. As operações especiais de *Requisição* e *Liberação* acontecem necessariamente dentro do escopo de uma *Transação Longa* (figura 3.5), enquanto que operações de *Cópia* e *Transferência* podem ocorrer fora de uma transação longa, desde que sejam observados os direitos de acesso. Uma operação de cópia cria um novo objeto à imagem e semelhança do objeto original no banco de

dados destino. Entretanto, o objeto original se mantém inalterado. O mesmo não acontece quando se utiliza a operação de transferência, onde a cópia é feita, mas o objeto original é descartado do banco de dados ao qual pertence. Observa-se que a operação de cópia cria um novo objeto, enquanto que a operação de transferência não o faz.

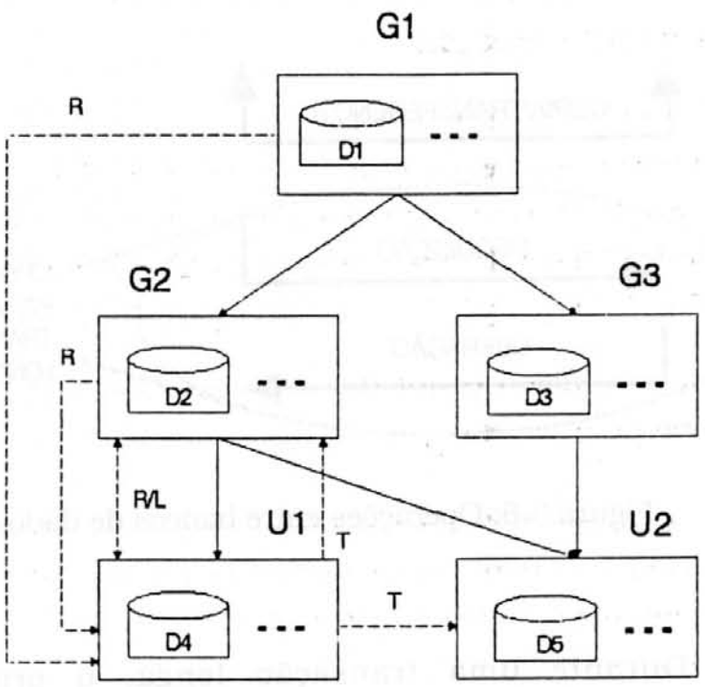

Figura 3.4: Direitos de acesso para grupos e usuários

Uma *Transação Longa* define uma unidade de trabalho em ambientes de projeto. De acordo com propriedades especiais do processo de projeto, uma transação longa não pode ser atômica, tampouco pode ser completamente isolada de outras transações /ABR 88/. O início e o fim de uma transação longa são explicitamente determinados pelo usuário através de operações específicas. O usuário, em geral, solicita objetos de um banco de dados público, modifica-os através da atividade de projeto por um longo período de tempo (em comparação com o tempo de uma transação de SGBDs convencionais) e retorna-os para o banco de

dados público, já modificados (figura 3.5).

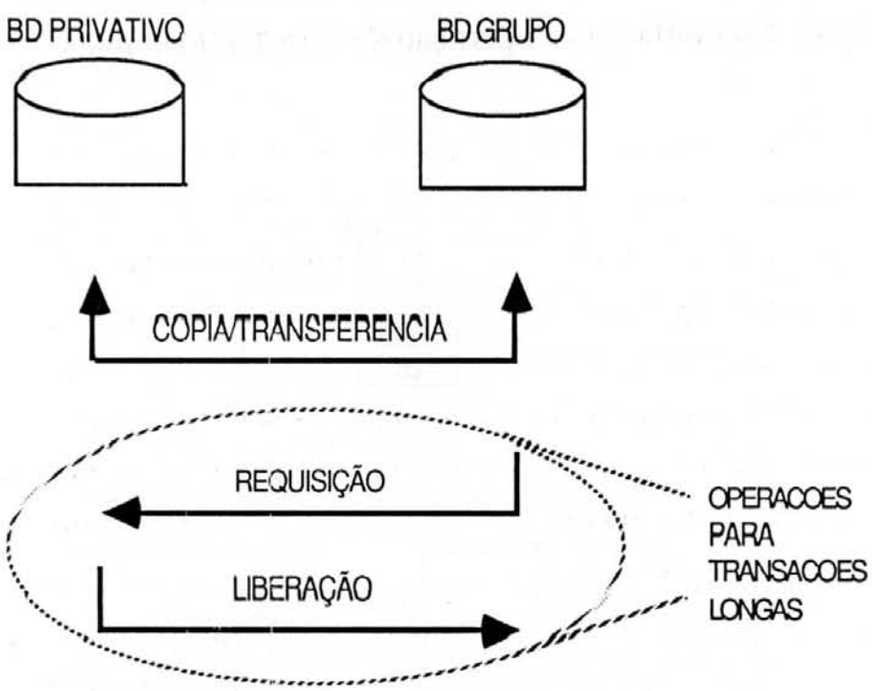

Figura 3.5: Operações entre bancos de dados

Durante uma transação longa, o projetista pode aleatoriamente requisitar e liberar objetos de/para qualquer banco de dados ao qual ele tenha acesso. A movimentação dos objetos de um banco de dados para outro envolve não somente a parte descritiva, mas também toda a parte estrutural do objeto que está sendo copiado. O usuário, no momento da requisição, deve determinar o tipo de bloqueio que será mantido pelo sistema até que o objeto, chamado de *objeto âncora*, seja liberado. Para isso vários tipos de bloqueios são possíveis, combinando permissões de leitura e escrita nos bancos de dados origem e destino da operação. Quando o objeto é liberado, todos os bloqueios são removidos e o objeto âncora pode ser manipulado livremente por qualquer usuário. A liberação não está restrita ao fim de uma transação longa, portanto pode ser realizada em qualquer instante durante a transação. Uma vez o objeto é liberado para o banco de dados original, seu velho estado é substituído pelo novo, denotando uma evolução no

desenvolvimento do objeto. No entanto, se o usuário não estiver satisfeito com as modificações feitas no objeto até então, ele pode descartá-las e liberar o objeto sem qualquer alteração no objeto âncora. Quando o objeto é liberado, o sistema modifica o objeto requisitado. Não existe a criação automática de uma nova versão do objeto.

O usuário proprietário de uma transação longa deve, quando achar conveniente, abortar ou terminar com sucesso sua transação. Em ambos os casos os objetos âncora são liberados, no entanto, no primeiro eles são mantidos na forma original, exatamente como antes da operação de solicitação, salvo aqueles que porventura tenham sido explicitamente liberados durante a transação. Uma transação longa não é uma unidade de reconstrução, porque não realiza o "rollback" dos objetos liberados durante o transcorrer da transação, no caso da mesma ser abortada.

As transações longas controlam, portanto, o compartilhamento dos objetos entre os diversos bancos de dados do ambiente. Ao usuário proprietário de uma transação longa, o DAMOKLES oferece um conjunto de operações que lhe possibilita manter total controle sobre os objetos por ele requisitados. O mecanismo verifica, ainda, no final de uma transação, as restrições de integridade implícitas do modelo, mostradas na seção 3.1.5.

3.2.2 Controle de Acesso Orientado a Objeto

A DML do DAMOKLES implementa uma interface com aplicação através da manipulação objeto por objeto. A unidade de processamento do DAMOKLES é o objeto, que é referenciado de acordo com o identificador interno. Isto quer dizer que a busca dos objetos, mesmo que eles sejam estruturados, é realizada individualmente e torna disponível apenas a parte descritiva do objeto. Para buscar objetos componentes, o sistema fornece operadores que permitem a navegação por toda a estrutura do objeto, mas sempre buscando um sub-objeto após o outro.

A busca de objetos do bancos de dados é invariavelmente realizada tendo como ponto de partida o identificador interno do objeto. Pesquisas associativas não são diretamente possíveis, portanto não existem operadores da DML que busquem objetos baseados em valores de atributos. Embora um conjunto de atributos possa ser declarado como chave externa (seção 3.1.1), um usuário não pode utilizar o valor da chave externa para recuperar diretamente o objeto. A fim de permitir a manipulação de conjuntos de objetos como unidade de processamento e a pesquisa associativa de objetos do banco de dados, o DAMOKLES oferece o conceito de cursor.

Um *cursor* representa um conjunto de objetos e relacionamentos de tipos arbitrários. Seus elementos são identificadores internos que podem ser obtidos individualmente ou como um todo, através do conjunto. O sistema suporta dois tipos de cursor:

Cursor Temporário: sua existência está restrita ao processo que executa o programa de aplicação. Não é persistente, isto é, não se mantém entre sessões distintas da aplicação. Seu conteúdo pode ser salvo em um cursor permanente. No entanto, as operações sobre o cursor se aplicam somente ao temporário;

Cursor Permanente: é um cursor persistente, logo sua existência não depende do processo da aplicação.

O conteúdo de um cursor pode ser determinado por uma expressão de pesquisa complexa que incorpora critérios associativos e estruturais, inclusive em conjunto. Uma pesquisa associativa se refere aos valores de atributos de objetos ou relacionamentos. Por outro lado, numa pesquisa orientada a estrutura, objetos e relacionamentos são localizados de acordo com suas conexões em relacionamentos gerais ou implícitos (versões e subobjetos) do modelo. O resultado da avaliação da expressão de pesquisa, isto é, o conjunto de todos os objetos e relacionamentos que satisfazem as condições específicas é armazenado em um cursor, o qual pode mais tarde ter seus elementos classificados de acordo com um critério definido pelo usuário.

Além da manipulação de elementos, o sistema suporta operações de conjuntos, entre as quais estão união, intersecção e diferença de cursores.

3.3 Aspectos de Operacionalidade

O Sistema DAMOKLES está implementado em linguagem C para o ambiente UNIX. Embora ainda seja somente um protótipo, a maioria das características descritas nas seções anteriores já está funcionando na versão 2.0. O protótipo implementado é um sistema modular, projetado para ter uma arquitetura multinível, onde no nível mais elementar estão os componentes que realizam a transferência de dados entre disco e memória principal, e no nível mais alto estão os componentes que implementam diretamente o modelo de dados do sistema.

3.3.1 Módulo de Administração

O usuário administrador de banco de dados tem no módulo de administração uma de suas principais ferramentas de trabalho. Através deste módulo, o administrador é capaz de instalar o SGBD, pode criar usuários do sistema, associá-los em grupos e ainda identificar os proprietários dos BDs que existem no ambiente.

A configuração do sistema é realizada pela primeira vez no momento da instalação. Neste instante são definidos valores de parâmetros necessários ao funcionamento do SGBD, por exemplo, tamanho do "buffer" do sistema. O usuário administrador tem a opção de entrar com valores próprios ou deixar que o sistema determine-os com valores "default".

Os bancos de dados presentes em um ambiente têm que ter necessariamente um proprietário que também precisa estar cadastrado. Esta associação entre BDs e seus usuários donos é realizada pelo módulo

de administração. Os BDs podem pertencer tanto a usuários individuais como a grupos. Neste caso os membros do grupo se tornam proprietários indiretos. Os direitos de acesso e a semântica de propriedade dos BDs estão descritos na seção 3.2.1. O módulo de administração não tem a função de criar bancos de dados, função esta desempenhada pelo módulo DDL-DAMOKLES.

3.3.2 Módulo DDL-DAMOKLES

A criação de um banco de dados implica a definição das entidades que farão parte daquele. A estrutura e a semântica da associação entre as entidades é determinada através de um *esquema* DAMOKLES. O esquema descreve a estrutura da base de dados. Ele se constitui num conjunto de declarações de constantes, domínios de atributos, definições de sinônimos, tipos de objetos e tipos de relacionamentos (figura 3.6).

```
SCHEMA <nome_esquema>
        CONST
                <decl_constantes>
        VALUE_SET
                <decl_domínios>
        <decl_sinônimos>
        <decl_tipos_objetos>
        <decl_tipos_relacionamentos>
END <nome_esquema>
```

Figura 3.6: Estrutura de um esquema

Constantes são geralmente usadas para definir diferentes estruturas de dados dependentes ou de mesmo tamanho /ABR 88/

(exemplo na figura 3.7). Seus valores são fixos para os bancos de dados do esquema.

```
CONST
        COMP_NOME = 30;
        COMP_TEXTO = 50;
        INICIO_PROJ = @04/06/90@;
        NOME_PROJ = "PROJ_GAMA";
```

Figura 3.7: Exemplo de constantes

Domínios de atributos descrevem o tipo dos atributos dos objetos e relacionamentos (exemplo na figura 3.8). Todos os domínios têm que ser definidos antes de usados. O DODM contém uma coleção de domínios pré-definidos:

CHAR: Caracteres de 8 bits;

INT: Inteiro sem sinal. Varia de -32768 a +32767;

BOOL: Assume valores TRUE e FALSE;

TIME: Denota tempo, inclui data, hora, minuto e segundos;

LONG_FIELD: Cadeia de bytes de comprimento arbitrário. Um atributo deste tipo é manipulado por operadores especiais;

ENUM: Descreve um conjunto através da enumeração de seus elementos;

STRING [n]: Para cadeias de caracteres ASCII de comprimento no máximo n;

BYTES [n]: Seqüência de bytes de comprimento máximo n;

SUBR: Para subconjuntos dos conjuntos de valores CHAR, INT, TIME;

ARRAY: Vetores dos conjuntos de valores CHAR, INT, BOOL, TIME;

STRUCT: Usado para registros. Um atributo deste tipo assume a estrutura declarada no domínio (tipo "DATA" no exemplo da figura 3.8);

UNION: União de conjunto de valores.

```
VALUE-SET
    TIPO_ESP : ENUM { VDM, LOGICA, ALGEBRICA };
    DATA     : STRUCT
                   dia: INT SUBR [1..31];
                   mes: INT SUBR [1..12];
                   ano: INT SUBR [1980..2010];
               END;
    NOME     : STRING [COMP_NOME];
    TEXTO    : STRING [COMP_TEXTO];
    INTEIRO  : INT;
```

Figura 3.8: Exemplo de domínios de atributos

Uma declaração de sinônimo define um novo identificador para um tipo de objeto, versão de objeto ou relacionamento.

A declaração de um tipo de objeto descreve as propriedades dos objetos do tipo. As propriedades incluem uma lista de atributos, declaração de versões, estrutura para tipos complexos e restrições de consistência em relacionamentos (a figura 3.9 mostra um exemplo). Os atributos são definidos através de uma lista, onde são declarados os

domínios de cada atributo. A cláusula de versões declara as versões do tipo. Elas podem conter propriedades descritivas e estruturais, possuir versões e podem participar de relacionamentos. A cláusula STRUCTURE define a estrutura complexa dos objetos do tipo. Esta cláusula é uma lista de tipos de objetos e tipos de relacionamentos declarados em outro lugar do esquema. Restrições de consistência em relacionamentos são declaradas por cláusulas de cardinalidade. Estas delimitam a freqüência de cada objeto do tipo no relacionamento.

```
OBJECT TYPE Documentos
    ATTRIBUTES
        Nome: STRING [COMP_NOME];
        Autor: STRING [COMP_NOME];
        Data: DATE;
    VERSIONS TREELIKE
    (
        ATTRIBUTE
            Autor: STRING [COMP_NOME];
            Data: DATE;
        STRUCTURE
            Formais, Informais
    )
    STRUCTURE
        Formais, Informais
    AT MOST ONCE ( Pertence_a )
END Documentos;
```

Figura 3.9: Exemplo de declaração de tipo de objeto

Um tipo de relacionamento associa vários tipos de objetos. Geralmente dois tipos de objetos participam do relacionamento, mas também é possível declarar um auto-relacionamento, onde somente

objetos de um único tipo são relacionados. Um papel tem que ser definido para cada tipo de objeto no auto-relacionamento. A figura 3.10 mostra um exemplo de declaração de relacionamento.

```
RELSHIP TYPE Pertence_a
        RELATES
                Documentos,
                Sistema
END Pertence_a;

RELSHIP TYPE Refer
        RELATES
                refnte: Documentos,
                refado: Documentos
END Refer;
```

Figura 3.10: Exemplo de declaração de tipo de relacionamento

Uma vez especificado, o esquema é então compilado pelo módulo DDL, gerando informações para o *dicionário de dados* DAMOKLES e criando um arquivo fonte C com as estruturas de dados definidas no esquema. Este arquivo é posteriormente incluído no programa de aplicação que utiliza o BD referente ao esquema.

Criar um BD no módulo DDL significa para o usuário determinar seu proprietário e associá-lo a um esquema previamente compilado. Não há restrições quanto à utilização de um esquema para diversos BDs, nem quanto à posse de vários BDs para um único usuário.

3.3.3 Módulo DML-DAMOKLES

O módulo DML-DAMOKLES implementa a linguagem de manipulação do sistema. Esta é realizada através da chamada de operações da DML pelo programa de aplicação escrito em linguagem C. O DAMOKLES possui somente a interface embutida na linguagem C, portanto não é possível ao usuário fazer consultas "ad hoc" através do uso de uma linguagem tipo SQL. A DML pode ser caracterizada como uma interface "um objeto por vez" /ABR 88/, isto significa que cada operação sempre retorna no máximo uma entidade (objeto ou relacionamento) para o programa de aplicação. Este, por seu vez, pode obter acesso e manipular os dados armazenados pela navegação de objeto para objeto através dos relacionamentos estruturais ou explícitos entre os objetos.

3.3.3.1 Operadores básicos

Estes operadores fazem a navegação, busca e atualização de objetos, relacionamentos e seus atributos. A navegação é sempre realizada na ordem seqüencial das entidades dentro de seus conjuntos. Operadores de navegação também utilizam relacionamentos entre objetos como base para a busca, assim pode-se localizar objetos que estão conectados a outros através de um relacionamento implícito ou explícito. A pesquisa seqüencial pode partir do início do conjunto ("FIRST"), do fim ("LAST") ou de acordo com um objeto do mesmo tipo previamente localizado ("PRIOR" e "NEXT"). Os operadores de navegação retornam sempre um identificador interno para a aplicação, que daí por diante tem a possibilidade de armazená-lo em uma variável de programa e usá-lo para leitura do objeto referenciado ou ainda para continuar a navegação.

Os operadores de busca transferem para o espaço de endereçamento da aplicação uma entidade referenciada pelo identificador interno. É necessário, portanto, que este identificador esteja previamente disponível para o programa de aplicação. Como a

busca é realizada em cima de um identificador interno, não é diretamente possível buscar uma entidade de acordo com seus valores de atributos.

Os operadores de atualização realizam a inserção, modificação e remoção de uma entidade da base de dados. No caso da remoção de um objeto, o alcance da operação, isto é, o conjunto de entidades que são afetadas, é controlado por um parâmetro de modo, formado pela composição das seguintes opções:

Descriptive: a exclusão afeta somente a parte descritiva e as versões caso existam;

Only: a remoção será realizada se nenhum dos objetos a ser removido participa de relacionamentos;

Selective: esta opção restringe os objetos àqueles que não são compartilhados por mais de um objeto estruturado (hierarquia sobreposta).

A inclusão de subobjetos pode acontecer de duas formas: na primeira, o objeto é inserido diretamente na hierarquia a qual pertence; na segunda, o objeto é inserido na base e num instante posterior é associado a sua hierarquia. A segunda forma é particularmente importante para hierarquias sobrepostas, porque o objeto é inserido na base uma única vez e pode posteriormente ser associado a vários objetos estruturados.

3.3.3.2 Operadores sobre versões

Os operadores sobre versões são essencialmente extensões ao conjunto de operadores sobre objetos. Estes operadores manipulam grafos de versões: navegação e busca de objetos versões de acordo com um número que controla o grafo ou identificador interno; inclusão e remoção de um objeto versão do grafo.

3.3.3.3 Operadores de campos longos

Para cada campo longo, o DAMOKLES mantém um apontador para uma posição interna. Esta posição corresponde ao "offset" - distância entre a posição corrente e o primeiro byte. O sistema oferece operadores para mover o apontador interno, recuperar e atualizar partes de um campo longo e fazer cópias destes para outros.

3.3.3.4 Operadores sobre cursor

Um cursor possibilita o tratamento de um conjunto de objetos como uma unidade de processamento (maiores detalhes ver seção 3.2.2). Existem dois tipos de cursor: temporário e permanente. As operações agem sobre cursores temporários, exceto aquelas que salvam e recuperam o conteúdo de um temporário para um permanente. O DAMOKLES oferece as seguintes operações:

Gerência de cursor: operações para criar e descartar cursores temporários e permanentes;

Operações sobre conjuntos: união, intersecção e diferença de dois cursores;

Operações sobre elementos: busca, inserção e remoção de elementos (entidades) de um cursor temporário;

Operações especiais: classificação e formação dos elementos de um cursor a partir de uma pesquisa associativa e/ou estrutural à base de dados.

O DAMOKLES oferece ainda um conjunto de operadores, já explicados na seção 3.2.1.1, para tratamento de transações longas e transferências de objetos entre múltiplos BDs. O conceito de transação curta (equivalente a uma transação nos modelos clássicos) está especificado mas seus operadores ("BEGIN", "ABORT", "END") ainda não foram implementados na versão 2.0.

3.4 Arquitetura do Sistema

A arquitetura multinível do sistema possibilita uma separação clara dos diferentes graus de abstração tratados em cada camada da implementação. Embora este tipo de arquitetura tenha sido utilizado na produção do DAMOKLES, foi necessária a inclusão de novas técnicas que permitissem o tratamento eficiente de objetos complexos. A figura 3.11 mostra a arquitetura em camadas do DAMOKLES.

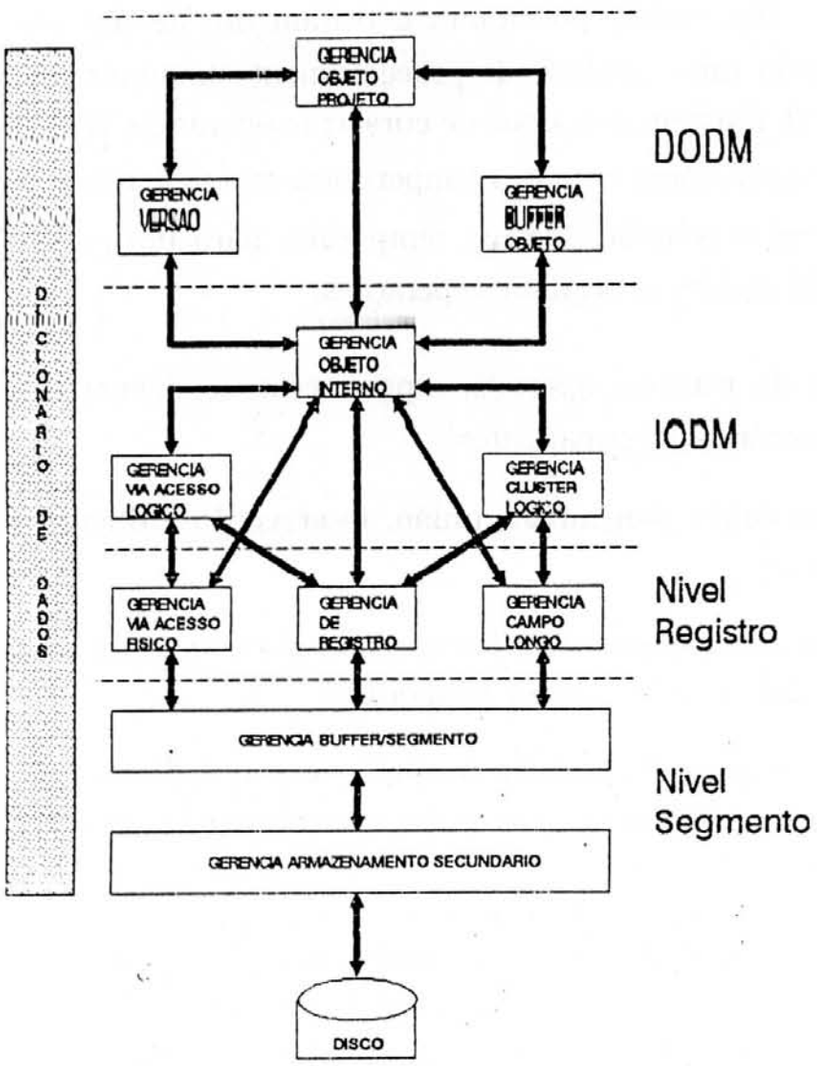

Figura 3.11: Arquitetura do DAMOKLES

O nível mais externo, chamado de DODM, implementa o modelo de dados do sistema. A unidade de processamento aqui é o objeto, de forma que se possa manipular um objeto estruturado. Mecanismos para reconstrução, proteção, controle de consistência e sincronização realizados sobre objetos são implementados neste nível. O gerenciamento de "buffer" de objeto leva em consideração agrupamentos de objetos para a transferência eficiente de objetos entre os níveis. As versões são diferenciadas dos objetos somente nesta camada, onde também é mantido o grafo de versões.

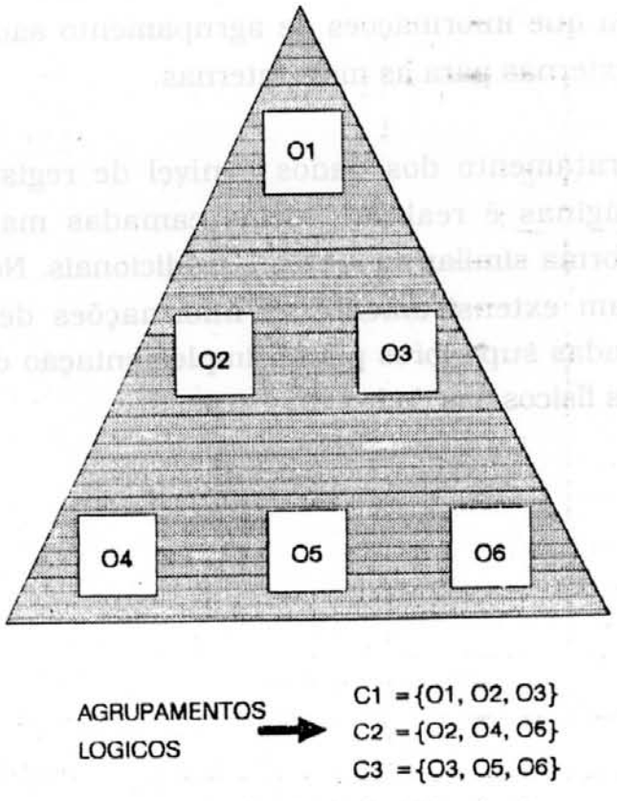

Figura 3.12: Agrupamentos Lógicos

A próxima camada abaixo do DODM, chamada de IODM ("Internal Object Data Model"), é ainda orientada a objeto, mas diminui a complexidade dos objetos estruturados através da utilização de agrupamentos lógicos de entidades ("cluster") /DIT 86b/. O mapeamento

do DODM para o IODM coloca todas as entidades diretamente componentes de um objeto estruturado em um agrupamento lógico (figura 3.12). Desta forma, a complexidade de implementar hierarquias arbitrárias de objetos e relacionamentos é reduzida ao problema de implementar conjuntos de objetos e relacionamentos simples /DIT 86b/.

No nível mais abaixo do IODM, os agrupamentos lógicos são divididos em agrupamentos físicos disjuntos que, por sua vez, levam a registros de tamanho variável. Estes níveis de abstração da informação tratada em cada camada resulta no acesso eficiente aos dados, à medida que informações de agrupamento são passadas das camadas mais externas para as mais internas.

O tratamento dos dados a nível de registro ou mesmo segmentos e páginas é realizado pelas camadas mais internas da arquitetura de forma similar aos SGBDs tradicionais. No entanto estas camadas utilizam extensivamente as informações de agrupamento vindas das camadas superiores para a implementação de mecanismos de agrupamentos físicos dos dados em disco.

4 O MODELO BD_PAC

Um modelo de dados é definido como um formalismo para expressar a estrutura lógica dos dados e ainda fornecer a base para a manipulação do banco de dados /MCL 81/. O modelo de dados deve ser uma ferramenta poderosa para declarar propriedades estáticas e dinâmicas da aplicação. Propriedades estáticas envolvem objetos, características de objetos e seus relacionamentos. Propriedades dinâmicas dizem respeito às operações sobre objetos e transações como um conjunto de operações relacionadas. Um modelo deve suportar também regras de integridade sobre propriedades estáticas e propriedades dinâmicas. Um modelo de dados consiste especificamente de três componentes /BRO 84/:

Componente de Dados: um conjunto de conceitos para a estruturação da informação. Mantém informação sobre a organização lógica e estrutural dos dados no banco de dados;

Conjunto de Operações: Envolve uma linguagem genérica de manipulação dos dados. Permite que os dados sejam atualizados e recuperados da base de dados;

Conjunto de Restrições de Integridade: Especificação de restrições de integridade para os relacionamentos entre os dados e metadados do sistema.

O modelo de dados do Sistema DAMOKLES - "Design Object Data Model" (DODM) - apresenta poucos recursos para a modelagem semântica de aplicações. O DODM está baseado no Modelo Entidade/Relacionamento /CHE 76/ com o suporte para objetos complexos e versões. Esta proposta estende o DODM com os conceitos de abstração: generalização, agregação molecular e associação (o sistema de tipos do DODM suporta o conceito de classificação), proporcionando maior poder de modelagem ao sistema, a fim de trazer para o banco de dados grande parcela da semântica da aplicação.

Durante o transcorrer deste capítulo são mostrados vários

exemplos que demonstram a modelagem BD_PAC de objetos de um sistema de auxílio ao desenvolvimento de software /MAC 89a/. Os exemplos demonstram a utilização dos conceitos do BD_PAC em uma aplicação de projeto.

Nas seções seguintes são apresentados os conceitos de abstração suportados pelo BD_PAC. Classificação e relacionamentos genéricos são diretamente importados do DODM, enquanto que generalização, agregação molecular e associação são peculiares ao BD_PAC.

4.1 Classificação

Classificação é o conceito de abstração que estabelece um relacionamento entre objetos do esquema e objetos da base de dados /BRO 84/. O conceito fornece um mecanismo para a especificação de tipos de objetos, os quais funcionam como um "template" ou moldura que determina a forma e o meio de interpretação das informações para todos os objetos do tipo na base de dados. *Instanciação* é o processo que relaciona um tipo às suas ocorrências na base de dados. Este relacionamento é estabelecido no momento em que o objeto é inserido no sistema.

Um tipo é uma declaração que descreve as propriedades comuns a todos os seus objetos. As propriedades são representadas pelos atributos que são mapeados para domínios (tipos de atributos) do sistema. Os atributos podem ser simples ou estruturados, de acordo com o domínio a que correspondem. Os domínios são todos aqueles já descritos na seção 3.3.2, no entanto destaque especial é dedicado a tipos de atributos estruturados e campos longos.

- Atributos estruturados são particularmente importantes para aplicações de projeto. Domínios como cadeias de caracteres, vetores e registros são freqüentemente utilizados em aplicações não convencionais para armazenar informações estruturadas.

Vetores de inteiros são excelentes para armazenar coordenadas espaciais, enquanto que registros possibilitam a construção de atributos parciais de um atributo;

- Campos longos permitem o armazenamento de grande volume de informação cuja estrutura não é controlada pelo sistema gerenciador. O sistema oferece operadores especiais para este tipo de atributo, de forma que seja possível atualizar e recuperar todo o atributo ou suas partes. Em um atributo do tipo campo longo pode-se guardar, por exemplo, todo um módulo de software ou um "layout" de circuito eletrônico. É possível ainda armazenar todo este texto, caso a aplicação fosse o controle de bancos de dados de dissertações.

Um tipo é declarado em um esquema BD_PAC como mostra o exemplo da figura 4.1. A descrição exata da sintaxe da linguagem de definição do BD_PAC encontra-se no Anexo 1. O tipo declarado no exemplo é chamado simples porque contém somente propriedades descritivas (ver seção 3.1) . A declaração de tipos com estrutura complexa é realizada utilizando-se um dos construtores de tipo, explicados mais adiante.

```
OBJECT TYPE INTERFACE
    ATTRIBUTES
        Autor : STRING[30];
        N_Func : INT;
        Tipos : BOOL;
        .
        .
        .
END INTERFACE;
```

Figura 4.1: Declaração de tipo simples

Um tipo, simples ou complexo, pode ser declarado como versionado. As características de versões de objetos dos tipos versionados são explicadas em detalhe no capitulo 5. A figura 4.2 mostra um exemplo de um tipo simples versionado.

```
OBJECT TYPE IMPLEMENTACAO
    ATTRIBUTES
        Autor:     STRING[30];
        Linguagem: STRING[20];
        Descricao: LONG_FIELD;
        Codigo:    LONG_FIELD
    VERSIONS TREELIKE
END IMPLEMENTACAO;
```

Figura 4.2: Declaração de tipo com versão

No BD_PAC, cada objeto pertence a somente um tipo. Um objeto qualquer da base não poderia ao mesmo tempo ser ocorrência dos tipos INTERFACE e IMPLEMENTACAO, por exemplo.

O modelo tem um conjunto de operadores básicos para a manutenção de objetos e versões de objetos. Operadores para manipulação de objetos são aqueles já citados na seção 3.3.3.1. Eles permitem navegar, buscar e atualizar objetos e seus atributos. O modelo oferece um conjunto de operadores especiais para manipular por partes ou completamente atributos do tipo campo longo. Para versões de objetos, o modelo suporta operadores para atualizar versões individualmente e ainda uma coleção de operadores para tratamento de grafo de versões.

4.2 Agregação

Agregação é a abstração na qual um relacionamento entre entidades é considerado uma entidade de nível mais alto /SMI 77/. O conceito, ao longo dos anos, tem evoluído para se tornar um poderoso meio de composição de objetos através das chamadas *hierarquias de agregação*. /BAT 85/ define o conceito de agregação molecular como sendo um conjunto de entidades heterogêneas e seus inter-relacionamentos visto como uma entidade única de nível mais elevado, chamada de objeto composto ou complexo. No Modelo Relacional já pode-se perceber claramente a aplicação da abstração.

O conceito de agregação corresponde à noção de propriedade no sentido de composição /MAT 88/. Ele descreve propriedades que o objeto deve ter a fim de existir consistentemente, portanto objetos compostos não podem existir sem seus componentes. A semântica de exclusão em cascata é uma das características da agregação molecular /BAT 85/. A exclusão de um objeto composto implica a retirada dos seus componentes da base. No entanto, a exclusão em cascata não é necessariamente uma norma. Um modelo pode suportar o compartilhamento de objeto em hierarquias superpostas, o que impede a exclusão automática de um objeto compartilhado. Na proposta do BD_PAC a exclusão em cascata é opcionalmente oferecida à aplicação.

No Modelo BD_PAC, o conceito de agregação molecular permite a formação de objetos complexos e o tratamento destes como uma unidade lógica de processamento. O conceito estabelece um relacionamento implícito de composição entre o objeto agregado e seus componentes. O modelo suporta a declaração de hierarquias de objetos superpostas, portanto um objeto componente pode ser compartilhado por várias hierarquias de agregação.

A declaração de um agregado molecular em um esquema BD_PAC é mostrada no exemplo da figura 4.3. O tipo agregado MODULO tem três componentes: dois tipos (INTERFACE e IMPLEMENTACAO) e um relacionamento (INT_IMPL), este último associa objetos dos dois

tipos componentes (INTERFACE e IMPLEMENTACAO). Um tipo agregado pode, portanto, ter tipos de objetos e tipos de relacionamentos em sua composição.

```
AGGREGATION TYPE MODULO
    ATTRIBUTES
        Autor:    STRING[30];
        Data:     DATE;
        Descricao: LONG_FIELD
    COMPONENTS
        INTERFACE     (AT LEAST 1),
        IMPLEMENTACAO (AT MOST  5),
        INT_IMPL
END MODULO;
```

Figura 4.3: Agregação molecular

Uma agregação pode ser declarada no esquema como tipo componente de si mesma, direta ou indiretamente. Observe-se a declaração do tipo PRODUCAO na figura 4.4. No exemplo, o tipo agregado PRODUCAO é composto por dois outros tipos: CABECALHO e CORPO. O tipo CORPO, por sua vez, é um agregado que tem como componentes objetos do tipo PRODUCAO, caracterizando uma recursão. Neste ponto é importante afirmar que recursões são possíveis, no entanto precisam ser acíclicas a nível de dados da base. Recursões são possíveis somente a nível de definição do esquema.

Uma característica importante para a agregação molecular é a possibilidade de restringir a quantidade de objetos componentes. Por exemplo, se uma pessoa é formada por cabeça, tronco e membros, não faz sentido dizer que uma única pessoa tem duas cabeças. No Modelo

BD_PAC a participação dos objetos na composição pode ser restrita de três formas: (1) declara-se o número máximo de objetos componentes possíveis; (2) determina-se o número mínimo de objetos componentes; (3) associa-se as duas formas anteriores, o que estabelece os limites mínimo e máximo dos objetos componentes. O exemplo da figura 4.3 define que um objeto do tipo MODULO tem que ter pelo menos um objeto componente do tipo INTERFACE, e no máximo cinco objetos componentes do tipo IMPLEMENTACAO. A cardinalidade do relacionamento implícito de composição é opcional, logo se não é importante para a aplicação controlar o número de partes de um objeto composto, o modelador pode simplesmente não determinar qualquer cardinalidade na definição do agregado.

```
AGGREGATION TYPE PRODUCAO
    COMPONENTS
        CABECALHO,
        CORPO
END PRODUCAO;

AGGREGATION TYPE CORPO
    ATTRIBUTES
        N_atrib:  INT;
        Tem_term: BOOL
    COMPONENTS
        PRODUCAO
END CORPO;
```

Figura 4.4: Agregação recursiva

Existem duas formas de inserir um objeto como componente de um agregado: (1) inclui-se o objeto ao mesmo tempo em

que se diz quem é o seu agregado; (2) inclui-se o objeto e, através de uma outra operação que estabelece o relacionamento implícito de composição, associa-o ao agregado molecular. Esta segunda opção é particularmente importante para a inclusão de componentes de agregações moleculares superpostas. O modelo também fornece operadores de navegação que caminham na estrutura hierárquica das agregações em ambos os sentidos - baixo para cima e cima para baixo. Há ainda operadores que buscam objetos de acordo com o relacionamento implícito de composição, assim dado um componente é possivel buscar o agregado e dado o agregado é possível buscar o componente.

4.3 Generalização

Generalização é a forma de abstração em que um conjunto de objetos similares é abstraído em um objeto de nível mais alto, onde são agrupadas as propriedades comuns aos objetos /SMI 77/. Na generalização, diferenças entre objetos semelhantes são ignoradas para formar um tipo de ordem mais elevada em que as similaridades podem ser enfatizadas /PEC 88/. O conceito possibilita a formação de hierarquias de supertipos e subtipos, onde estes últimos são ditos *tipos especializados*.

Os supertipos, chamados também de tipos generalizados, agrupam propriedades comuns aos seus respectivos tipos especializados. Embora as propriedades dos supertipos não estejam definidas nos subtipos, elas também fazem parte do conjunto de propriedades dos subtipos através de um processo chamado de *herança de propriedades*. O mecanismo de herança determina que objetos de subtipos herdam propriedades dos objetos dos supertipos /BAN 87/.

No Modelo BD_PAC, um relacionamento implícito de generalização é mantido entre objetos do supertipo e objetos de seus subtipos. O processo de herança é inerente ao mecanismo de generalização. A herança de atributos para hierarquias de tipos ocorre

logicamente tanto a nível de definição (esquema) como a nível de objeto da base. Isto quer dizer que a estrutura descritiva (atributos) do supertipo vale para um subtipo, juntamente com os respectivos valores de atributos. Suponha que um sistema de auxílio ao projeto de software armazene informações sobre o pessoal que trabalha nos projetos. FUNCIONARIOS pode ser modelado como uma generalização de ANALISTAS e PROGRAMADORES, como mostra a figura 4.5. Se NOME e NUM_IDENT são atributos de FUNCIONARIOS, por herança, eles também são atributos dos subtipos ANALISTAS e PROGRAMADORES. Suponha agora que um funcionário (tipo FUNCIONARIOS) de nome "JOAO" (NOME="JOAO") e que tem identidade número "1.234.567" (NUM_IDENT=1.234.567) é programador (tipo PROGRAMADORES) especialista na linguagem C (LINGUAGEM="C"). Visto através do tipo FUNCIONARIOS, este funcionário tem nome "JOAO" e número de identidade "1.234.567". Visto através do tipo PROGRAMADOR este mesmo funcionário tem nome "JOAO", número de identidade "1.234.567" e é especialista em linguagem "C".

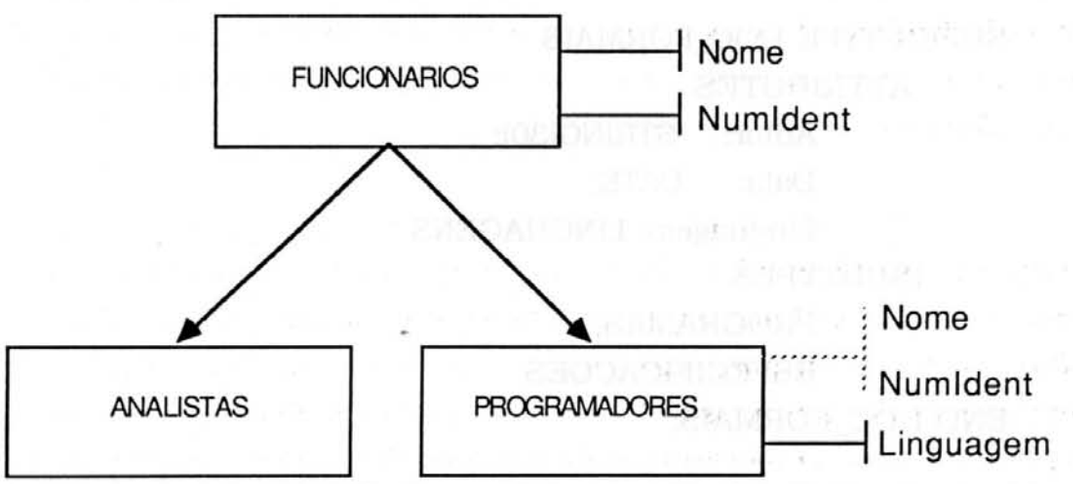

Figura 4.5: Generalização

No BD_PAC, um objeto de um supertipo pode aparecer em mais de um subtipo. Isto quer dizer que o funcionário de nome "JOAO"

do exemplo anterior poderia ser um analista (tipo ANALISTAS) em um projeto e, ao mesmo tempo, um programador (tipo PROGRAMADORES) em outro.

A declaração de uma hierarquia de generalização em um esquema BD_PAC é mostrada no exemplo da figura 4.6. No exemplo, DOC_FORMAIS é supertipo de PROGRAMAS e ESPECIFICACOES, portanto eles herdam as propriedades de DOC_FORMAIS. O processo de herança não se restringe somente às propriedades descritivas (atributos), mas também alcança a característica de ser um tipo versionado. Se DOC_FORMAIS fosse um tipo com versões, também o seriam os subtipos PROGRAMAS e ESPECIFICACOES. Os atributos das versões do supertipo são herdados para as versões dos subtipos automaticamente. Quando os subtipos são versionados não pelo processo de herança mas por definição, os atributos das versões do supertipo somam-se aos atributos próprios das versões dos subtipos, formando uma única estrutura de versão.

```
SUPER TYPE DOC_FORMAIS
        ATTRIBUTES
                Autor:    STRING[30];
                Data:     DATE;
                Linguagem: LINGUAGENS
        SUBTYPES
                PROGRAMAS,
                ESPECIFICACOES
END DOC_FORMAIS;
```

Figura 4.6: Declaração de generalização

A inclusão de objetos em hierarquias de generalização pode

ser realizada tanto através dos supertipos como através dos subtipos. Quando um objeto é inserido em um supertipo, o sistema trata somente os atributos do supertipo. Por outro lado, quando um objeto é inserido em um subtipo, o sistema automaticamente cria um novo objeto no supertipo e associa os dois objetos através de um relacionamento implícito de generalização. Neste caso, são atualizados os valores dos atributos do objeto a nível de subtipo, bem como os valores dos atributos do objeto criado automaticamente a nível de supertipo. Para o caso em que o objeto no nível de supertipo já exista na base de dados, o modelo oferece um operador para inclusão de objeto a nível de subtipo sem que haja a criação automática no nível de supertipo.

Exclusões de objetos dos subtipos não são propagadas para objetos dos supertipos. No entanto, quando se exclui um objeto do supertipo, todos os objetos dos subtipos que mantém um relacionamento implícito de generalização também são automaticamente excluídos. No exemplo da figura 4.5, caso o funcionário "JOAO" fosse despedido, não faria sentido mantê-lo como programador de um projeto.

Quanto a busca de objetos, somente os atributos no nível de supertipo são buscados numa consulta através do supertipo. Quando a consulta é realizada através do subtipo, tanto os atributos a nível de supertipo quanto os atributos a nível de subtipo ficam disponíveis para a aplicação.

4.4 Associação

A associação é a forma de abstração em que um conjunto de objetos (*membros*) é considerado um objeto conjunto de nível mais alto (*conjunto*). O conceito é usado para associar ou particionar um número variável de objetos de uma dada classe ou de classes distintas. /BRO 81/ restringe a aplicação da associação a objetos de um único tipo, como forma de particionar o conjunto de objetos do tipo. /ATW 85/ também restringe o particionamento aos objetos de um tipo, mas vai mais longe. Ele determina que o particionamento divida o conjunto em subconjuntos disjuntos de acordo com um atributo definidor de partição. /HAM 81/ e /MAT 88/ não restringem o particionamento a um único tipo, afirmando que o conjunto pode derivar de ocorrências de vários tipos distintos. Ambos compartilham com /ATW 85/ na idéia do atributo definidor, porém /MAT 88/ chama de *função de pertinência* uma característica que elementos precisam ter para pertencer ao conjunto. A função de pertinência é um conceito bem mais flexível do que o atributo definidor. /HAM 81/ chama de *predicado* o conceito de função de pertinência de /MAT 88/, mas sua proposta também suporta a formação de conjuntos controlados pelo usuário, no sentido de que este determina quem são os membros do conjunto. Conjuntos controlados pelo usuário não exigem que todos os tipos base da associação tenham uma característica comum, permitindo a formação de conjuntos sobre qualquer combinação de tipos definidos no esquema.

No conceito de associação não existe herança de atributos ou métodos. Um conjunto associação não define as propriedades nem a estrutura dos seus membros. Estes têm que ser necessariamente objetos de tipos do esquema. No Modelo BD_PAC, estes tipos podem ser diversos e não é necessário que eles tenham um atributo comum. A formação dos conjuntos acontece pela enumeração dos elementos realizada pelo usuário.

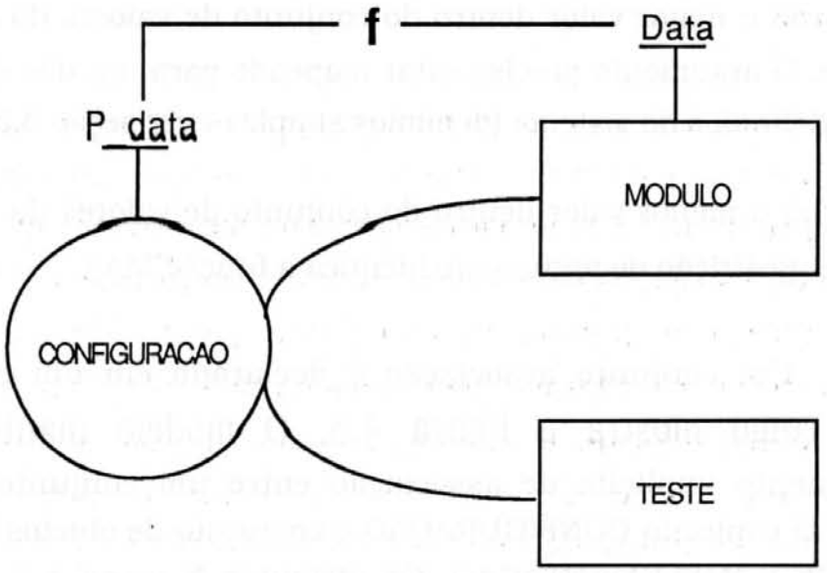

Figura 4.7: Associação

As propriedades dos conjuntos são utilizadas para expressar características do conjunto como um todo. No BD_PAC, estas características podem ser independentes ou derivadas das propriedades dos tipos membros. Os atributos independentes são próprios dos conjuntos, sem qualquer relação de dependência com os atributos dos membros. Eles diferem dos derivados porque estes últimos são calculados pelo sistema a partir dos membros dos conjuntos e não podem ser diretamente atualizados por operações de usuário. Atributos derivados são calculados a partir de funções pré-definidas do sistema e declaradas na definição da associação (figura 4.7). As funções pré-definidas são as mesmas fornecidas pela SQL /DAT 86/:

COUNT: calcula o número de objetos do tipo (funciona como o COUNT(*) do SQL). Tem como argumento um dos tipos membros;

SUM: soma os valores do atributo especificado. Seu argumento precisa ser numérico e pertencer a um dos tipos membros;

AVG: calcula a média aritmética dos valores do atributo base. Argumento tem que ser um atributo dos tipos membros com domínio numérico;

MAX: retorna o maior valor dentro do conjunto de valores do atributo base. O argumento precisa estar mapeado para um dos domínios pré-definidos do sistema (domínios simples - ver seção 3.3.2);

MIN: retorna o menor valor dentro do conjunto de valores do atributo base. Restrição de argumento idêntica a função MAX.

Um conjunto associação é declarado em um esquema BD_PAC como mostra a figura 4.8. O modelo mantém um relacionamento implícito de associação entre um conjunto e seus membros. O conjunto CONFIGURACAO é composto de objetos dos tipo INTERFACE e IMPLEMENTACAO. Os atributos N_interf e N_funcoes são derivados respectivamente pela contagem de todos os objetos do tipo INTERFACE no conjunto e pela soma dos valores do atributo N_func dos objetos do tipo INTERFACE do conjunto. Para melhor descrição da sintaxe da declaração de associações ver Anexo 1 que contém a descrição sintática da Linguagem de Definição de Dados do BD_PAC.

```
SET TYPE CONFIGURACAO
        ATTRIBUTES
                Data_cria:  DATE;
                Anal_teste: STRING[30];
                Data_teste: DATE;
                N_interf:   COUNT ( INTERFACE );
                N_funcoes:  SUM ( INTERFACE.N_func )
        MEMBERS
                INTERFACE,
                IMPLEMENTACAO
END CONFIGURACAO;
```

Figura 4.8: Declaração de associação

O modelo fornece operadores para inclusão, exclusão e consulta a objetos construídos por associação. Operadores para caminhamento nas hierarquias de conjunto são baseados nos relacionamentos implícitos de associação. Dado um conjunto, pode-se buscar um membro e dado um membro pode-se também buscar um conjunto.

A atualização dos atributos dos objetos associação pode ser realizada somente para atributos independentes. Os atributos derivados são sistema-dependentes, portanto são automaticamente atualizados pelo sistema.

4.5 Relacionamento

Tipos de relacionamentos estabelecem uma ligação genérica entre tipos de objetos. A existência dos objetos relacionados é condição necessária para a existência do relacionamento. O significado semântico do relacionamento fica sob a responsabilidade da aplicação.

Os relacionamentos no BD_PAC, assim como no DAMOKLES, são um tipo especial de objeto. Relacionamentos, a exemplo dos objetos, podem possuir atributos, podem fazer parte na composição de agregações moleculares e possuem identificador único atribuído pelo sistema.

Papéis podem ser declarados para cada tipo de objeto que participa do tipo de relacionamento. O BD_PAC também suporta a declaração de cardinalidade para os tipos de objetos participantes.

Em um esquema BD_PAC, um relacionamento é declarado de forma idêntica ao Sistema DAMOKLES como mostra o exemplo da figura 4.9. O relacionamento INT_IMPL relaciona objetos dos tipos INTERFACE e IMPLEMENTACAO. Como a descrição de papéis é opcional, não é necessário que seja especificada, como no exemplo.

```
RELSHIP TYPE INT_IMPL
    ATTRIBUTES
        Param: VET_PARAM;
    RELATES
        INTERFACE,
        IMPLEMENTACAO
END INT_IMPL;
```

Figura 4.9: Declaração de relacionamento

A fim de manter relacionamentos, o sistema fornece operadores de inclusão e exclusão de objetos relacionamentos. Na operação de inclusão, há a necessidade de existência e de identificação dos objetos que compõem o relacionamento nos devidos papéis. A exclusão de qualquer um dos objetos participantes de um relacionamento implica diretamente na exclusão do relacionamento (integridade referencial explicada na seção 3.1).

Operadores de navegação localizam objetos a partir de relacionamentos e relacionamentos a partir de objetos participantes. No primeiro caso, o sistema tem a sua disposição um objeto relacionamento e precisa buscar as informações dos objetos que estão relacionados. No segundo, um objeto participante está disponível, tornando possível a busca de um relacionamento que o relaciona a um outro objeto.

4.6 Integração dos Conceitos

Os conceitos de abstração podem ser usados em conjunto na modelagem dos dados de uma aplicação. É possível, por exemplo, generalizar tipos agregados ou agregar supertipos ou ainda fazer

associações de agregados moleculares.

Hierarquias de generalização podem envolver tipos simples e tipos construídos. Herança de propriedades ocorre sempre dos supertipos para os subtipos. Quando um agregado molecular participa da hierarquia como subtipo, a herança ocorre somente até o objeto agregado, como mostra o exemplo da figura 4.10. Semelhantemente, se o subtipo é um conjunto, seus membros não herdam as propriedades dos supertipos.

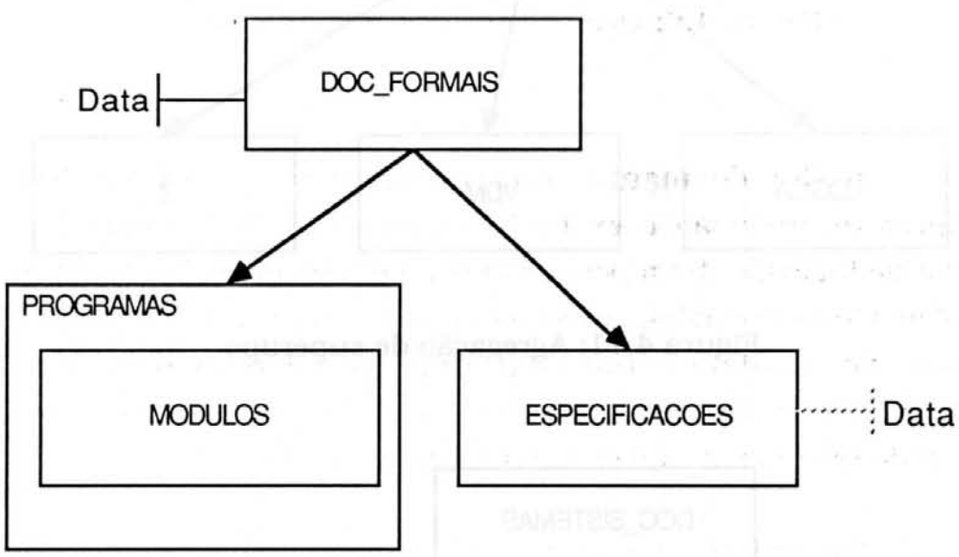

Figura 4.10: Generalização com agregação

O modelo também suporta a formação de agregados moleculares onde os componentes são tipos generalizados (figura 4.11). As operações de manutenção destes componentes seguem todas as especificações da hierarquia supertipo/subtipo. No entanto, na recuperação de um objeto do supertipo, o objeto a nível de subtipo não é automaticamente buscado. Quando um subtipo participa diretamente como componente de um agregado, a capacidade de herdar as propriedades do supertipo continuam valendo, portanto os atributos do supertipo são automaticamente recuperados quando houver acesso ao

subtipo componente (figura 4.12). Por outro lado, a exclusão de um objeto agregado não implica a exclusão também do objeto no nível de supertipo.

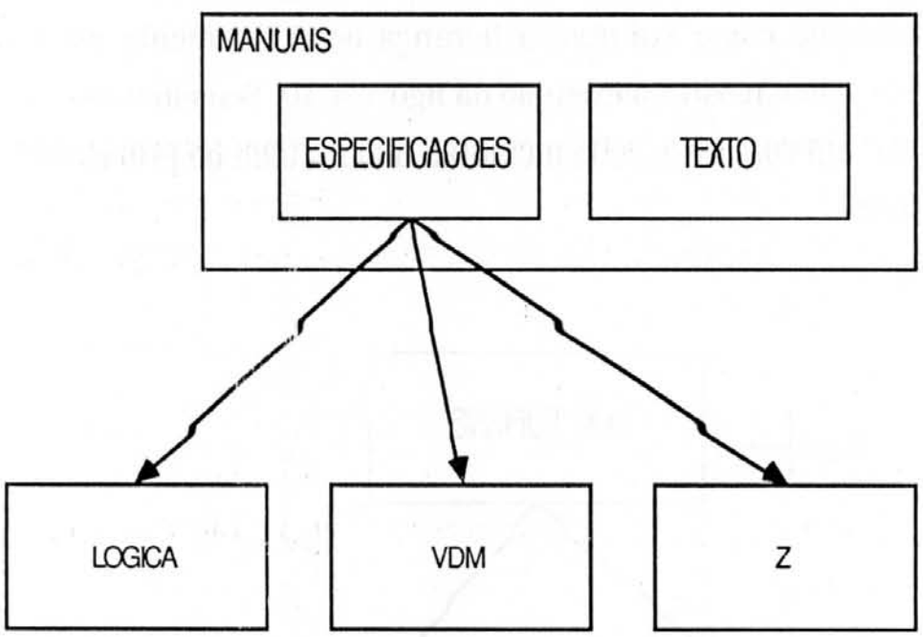

Figura 4.11: Agregação de supertipo

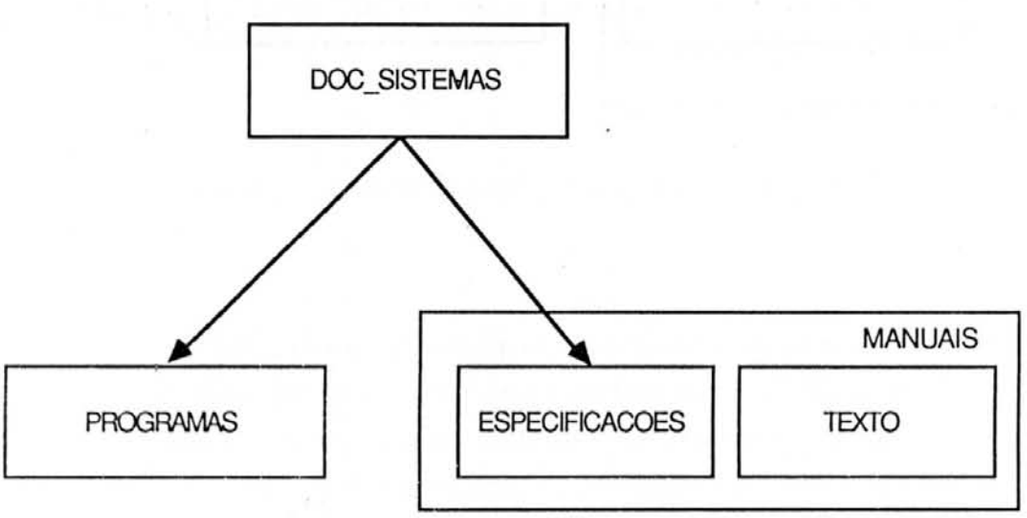

Figura 4.12: Agregação de subtipo

Associações, assim como generalizações, também podem

ser componentes de agregações. Neste caso os objetos componentes são conjuntos formados a partir de objetos dos tipos declarados no esquema como membros.

Associações de supertipos são perfeitamente possíveis no BD_PAC. Objetos dos supertipos formam o conjunto associação da mesma maneira que os objetos de um tipo básico. Por seu vez, um conjunto também pode ser formado por subtipos de uma hierarquia de generalização. Neste caso, são válidos para efeito de cálculo de atributos derivados do conjunto, todos os atributos do subtipo, sejam eles próprios ou herdados.

Quanto às hierarquias de agregação, não existe qualquer restrição de participação como membro de conjuntos. Um tipo declarado como membro tanto pode ser um tipo agregado como um tipo componente.

Relacionamentos podem ligar objetos simples, agregados moleculares, objetos generalizados e conjuntos. Os conceitos não precisam ser homogêneos, ou seja, o modelo suporta relacionamento entre supertipos e tipos básicos, agregações moleculares e subtipos, associações e componentes, entre outros.

5 O PLANO DE VERSÕES

Projetar qualquer objeto industrial é sempre uma atividade evolutiva. Para se chegar ao ponto em que um protótipo ou um modelo do que se quer construir esteja pronto, os projetistas provavelmente já terão tentado vários caminhos alternativos de construir o objeto ou mesmo terão consertado diversas falhas que poderiam ter ocorrido durante a concepção do objeto. Em aplicações de suporte a projeto é de alta importância o registro e o acesso ao histórico evolutivo de concepção daquilo que se quer projetar. A relevância advém de três aspectos importantes da aplicação:

- Alternativas tentadas e abandonadas podem ser mais tarde retomadas para continuar o desenvolvimento do mesmo objeto de projeto ou de um outro que se quer construir;
- É necessário, algumas vezes, manter formas diferentes de se construir um objeto de projeto, a fim de possibilitar liberdade de escolha de um único produto com pequenas diferenças ou mesmo obter o mesmo produto com condições básicas de construção diferentes;
- O registro de todas as fases de projeto reúne um acervo importante de informações para quem constrói o objeto de projeto.

A fim de suprir as necessidades de controle e armazenamento da evolução de projeto no domínio das aplicações não convencionais - CAD, CAM, CASE - propostas de Sistemas de Bancos de Dados (/KAT 83/, /KAT 86b/, /ZDO 84/, /ZDO 86/, /ATW 85/, /BAT 85/, /DIT 86b/, /KUO 86/, /CHO 88/, /BER 88/, /CEL 90/) têm incluído, entre outras novas funções, a *gerência de versões* de objetos de projeto. Alguns desses sistemas apresentam um modelo dedicado à aplicação para a qual foram desenvolvidos, como é o caso de /KUO 86/ para Ambientes de Desenvolvimento de Software, enquanto que /KAT 83/ e /BAT 85/ suportam o projeto de sistemas digitais.

Em /BAT 85/ versões são objetos de projeto que

compartilham uma mesma interface. Uma interface é considerada um tipo de objeto e suas versões, ocorrências do tipo. Pelo conceito de *Generalização de Versões*, propriedades da interface são herdadas para cada uma de suas versões. A interface, além de abstrair características comuns à todas as suas versões, também determina a forma de interconexão entre objetos para formar objetos moleculares.

/KAT 83/ também propõe um modelo para sistemas de projeto mas não implementa a generalização de versões. Para /KAT 83/ a atividade de projeto é evolucionária e as versões são melhorias ou correções de objeto de projeto. Sua proposta divide claramente o espaço estrutural, onde estão incluídas as versões, e o espaço representacional, onde ficam as ocorrências de objeto propriamente ditas.

No Sistema ORION /BAN 87/, que implementa um modelo genérico, objetos podem ser declarados como versionáveis ou não /CHO 88/. Quando o objeto é não versionado, ele é a propria ocorrência do objeto na classe. Um objeto versionável, por outro lado, é um objeto genérico que descreve seu grafo de derivação de versões, portanto a ocorrência de um objeto se traduz por uma de suas versões específicas do grafo. Todos os objetos pertencentes a uma classe são descritos pelo mesmo conjunto de atributos e métodos. As versões dos objetos versionados herdam a estrutura declarada na classe, ou seja, cada versão é descrita pelo mesmo conjunto de atributos e métodos que descreve os objetos da classe.

/ATW 85/ propõe forma semelhante para o tratamento de versões. Em sua abordagem, um objeto de tipo versionado armazena alguns dados comuns à todas as versões do objeto e permite o tratamento das versões como um conjunto unitário de objetos. /ATW 85/ define o conceito de *percolação*, onde a criação de uma versão de objeto é automaticamente propagada para cima na hierarquia de composição de objetos. Semelhante característica é encontrada também nas propostas de /KAT 83/, /ZDO 84/ e /CEL 90/, esta última adotada no sistema O_2 /BAN 88b/.

A abordagem para o plano de versões de /DIT 86b/ já foi mostrada na seção 3.2. Aqui é importante destacar a peculiaridade do tratamento de versões no Sistema DAMOKLES (/DIT 86b/, /ABR 88/). Versões são objetos que podem assumir características totalmente diferentes daquelas de seu objeto genérico. São objetos como qualquer outro dentro do modelo de dados, com uma única restrição de existência dependente da existência do objeto genérico.

/BER 88/ propõe uma abordagem diferente das demais citadas acima. /BER 88/ advoga que as versões devem ser modeladas diretamente pelos construtores de tipos do modelo de dados. Neste caso, as versões são objetos especializados dentro de hierarquias de generalização. Esta proposta elimina a idéia do objeto genérico encontrado em várias outras acima citadas e traz o plano de versões para dentro do plano de conceitos de abstração do modelo.

5.1 Versões no BD_PAC

O plano de versões do Modelo BD_PAC possibilita à aplicação manter diversas representações de um mesmo objeto, oferecendo assim um mecanismo interno para o armazenamento e manipulação do histórico evolutivo ou alternativas de projeto.

Após a criação de um objeto de projeto, novas versões podem ser derivadas do objeto e ainda novas versões podem ser derivadas das versões do objeto, formando um *grafo de derivação de versões* . Um grafo de derivação de versões captura a evolução do projeto e indica a ordem parcial das versões do objeto. O projeto de um circuito integrado, por exemplo, pode envolver um número de diferentes versões durante o desenvolvimento do circuito. Seguindo o projeto inicial, novas versões do projeto do circuito podem ser derivadas para, por exemplo, reduzir-lhe o tamanho ou mesmo diminuir o consumo de energia. Em adição a isto, pode ser necessário gerar versões paralelas a fim de experimentar diferentes alternativas de projeto.

Quando um tipo é declarado no esquema BD_PAC como versionado, o grafo de derivação das versões dos objetos do tipo segue a forma que consta da declaração: linear, árvore ou acíclico. Este grafo representa um relacionamento que define uma ordem parcial entre as versões de um objeto (figura 5.1). No grafo linear, uma versão pode ter somente uma predecessora e uma sucessora. No grafo em árvore, uma versão pode ter somente uma predecessora, mas várias sucessoras. Para o grafo acíclico, versões podem ter várias antecessoras e sucessoras.

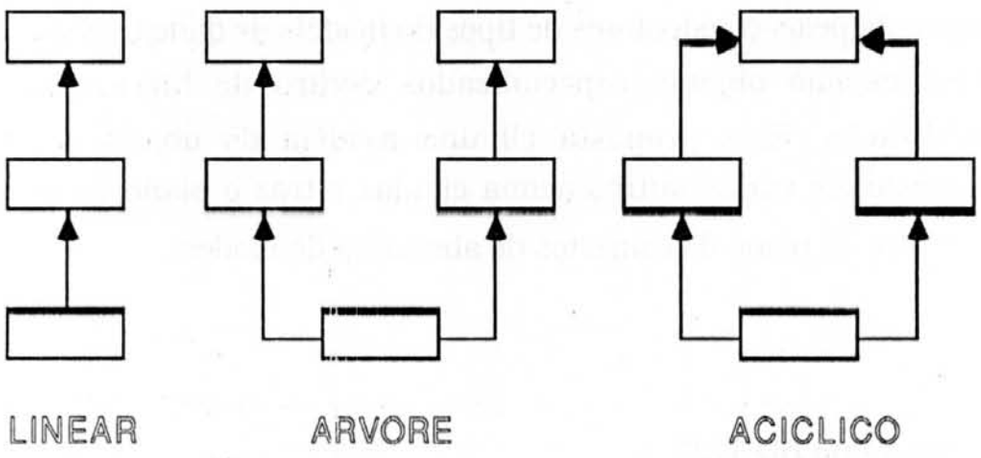

Figura 5.1: Grafos de derivação de versões.

A inclusão das versões no grafo de derivação é de responsabilidade do projetista ou da ferramenta que utiliza o Sistema de Banco de Dados. O sistema oferece operadores para manipulação de grafo de derivação que, uma vez utilizados pelo projetista/ferramenta, montam um grafo enquanto que verificam a operação de acordo com a declaração do grafo no esquema.

No BD_PAC um tipo é versionado ou não-versionado. Um tipo versionado contém objetos abstratos. Cada objeto abstrato representa o conjunto de suas versões. O suporte para versões se aplica somente a tipos versionados. Neste capítulo, o termo *objeto genérico* se refere a um objeto de tipo versionado, por outro lado o termo *objeto* trata de um objeto de um tipo não-versionado. O termo *versão* se refere a uma versão específica do objeto genérico.

A abordagem do controle de versões no BD_PAC é semelhante àquelas propostas por /ATW 85/ e /CHO 88/. Os objetos de um tipo versionado são objetos abstratos que possibilitam o acesso ao conjunto das suas versões. As versões dos tipos versionados têm a estrutura declarada no tipo. Isto implica que os relacionamentos semânticos declarados através dos construtores de tipo (ver capítulo 4: O MODELO BD_PAC) se aplicam diretamente nas versões de tipos versionados. A figura 5.2 mostra um exemplo para os três níveis de informação dos tipos. Os tipos T1 e T2, assumidamente versionados, contêm respectivamente, os objetos OB1 e OB2, genéricos. O objeto genérico OB1 tem duas versões, V1 e V2, enquanto que o objeto genérico OB2 tem uma versão, V3. Por outro lado, o tipo T3, que, suponha-se, não foi declarado versionado, contém somente um objeto, OB3, sem qualquer versão.

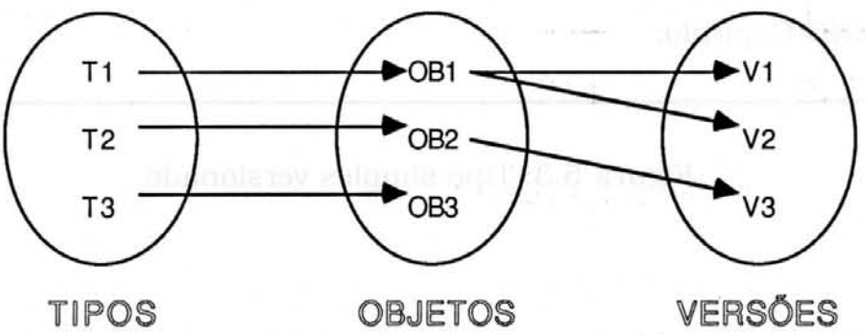

Figura 5.2: Níveis de informação.

Versões diferem dos objetos pelas seguintes características:

- Existe um relacionamento implícito entre as versões de um objeto genérico que as organiza de acordo com um grafo declarado no esquema BD_PAC.;
- Versões dependem diretamente do objeto genérico, isto quer dizer que uma vez excluído o objeto, suas versões necessariamente também o são;

Para que um objeto possa ter versões, o seu tipo deve ser declarado no esquema BD_PAC como mostra o exemplo da figura 5.3. A ordem parcial entre as versões pode assumir três declarações distintas: "LINEAR", "TREELIKE" e "ACYCLIC". O anexo 1 contém a descrição detalhada da gramática de um esquema BD_PAC, portanto pode ser consultado para se obter a sintaxe exata da declaração de tipos versionados.

```
OBJECT TYPE Capitulo
    ATTRIBUTES
        Nome: String[30];
        Num_ordem: INT;
        Texto: LONG_FIELD
    VERSIONS TREELIKE
END Capitulo;
```

Figura 5.3: Tipo simples versionado

5.1.1 Versões e generalização

Numa hierarquia de generalização do BD_PAC, os tipos construídos são chamados de supertipos, enquanto que os tipos de mais baixo nível, ou seja os tipos construtores, são chamados de subtipos (ver capítulo 4: O MODELO BD_PAC). A principal característica das hierarquias de generalização do BD_PAC é a herança de propriedades dos supertipos para os subtipos. Esta herança acontece não somente a nível de atributos, mas também a nível de propriedade de ser um tipo versionado. Observe-se as declarações da figura 5.4. Os tipos SB_B e SB_C, embora não tenham sido declarados como tipos versionados, pela propriedade de herança, também o são, com grafo de derivação de

versões semelhante ao grafo declarado para o supertipo SP_A.

```
SUPER TYPE SP_A
        ATTRIBUTES
                Att1 : DOUBLE;
                Att2 : LONG_FIELD
        VERSIONS LINEAR
        SUBTYPES
                SB_B,
                SB_C
END SP_A;

OBJECT TYPE SB_B
        ATTRIBUTES
                Att3 : CHAR;
                Att4 : INT
END SB_B;

OBJECT TYPE SB_C
        ATTRIBUTES
                Att5 : FLOAT
END SB_C;
```

Figura 5.4: Declaração generalização versionada

A recíproca para os subtipos não é verdadeira, isto é, quando um subtipo tem versões, não necessariamente também terão versões seus supertipos. A herança somente se dá dos supertipos para os subtipos e nunca no sentido contrário.

O plano de versões é restrito pelos relacionamentos semânticos em que os objetos genéricos participam. Observe-se o

exemplo da figura 5.5. Sejam os objetos genéricos OB1 e OB2 objetos do supertipo SP_A declarado na figura 5.4 e os objetos genéricos OB3 e OB4 objetos dos subtipos SB_B e SB_C, respectivamente. Considerando-se que OB3 e OB4 são subobjetos de OB1 (representado na figura 5.5 pela seta mais grossa), o relacionamento de generalização das versões de OB3 e OB4 está restrito às versões de OB1 somente. Uma versão de OB2, por exemplo, não poderia especializar-se em versões de OB4, porque OB2 não é superobjeto de OB4,

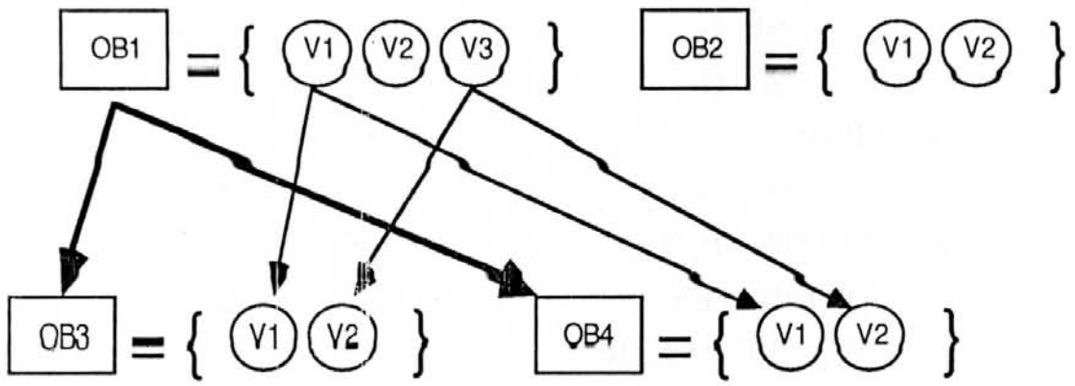

Figura 5.5: Versões de superobjetos e subobjetos

A herança de propriedades que ocorre entre supertipos e subtipos alcança também as versões dos objetos desta hierarquia. Desta forma, todos os atributos das versões de nível mais alto na hierarquia são passados para os níveis inferiores recursivamente até que se chegue a camada elementar da estrutura hierárquica.

A cada versão do subtipo corresponde somente uma versão do supertipo. Esta característica é necessária devido à herança de propriedades entre versões dentro de uma hierarquia de generalização. Observe-se novamente a figura 5.5. Cada versão dos objetos genéricos OB3 e OB4 corresponde a somente uma versão do objeto genérico OB1. Não existe a possibilidade de que duas versões do supertipo sejam especializadas para uma única versão do subtipo.

```
AGGREGATION TYPE TESE
        ATTRIBUTES
                Titulo: STRING[30];
                Autor: STRING[30];
                Data: DATE
        VERSIONS ACYCLIC
        COMPONENTS
                CAPA,
                CAPITULO,
END TESE;

OBJECT TYPE CAPA
        ATTRIBUTES
                Orientador: STRING[30]
END CAPA;

OBJECT TYPE CAPITULO
        ATTRIBUTES
                Titulo: STRING[40];
                Texto: LONG_FIELD
        VERSIONS ACYCLIC
END CAPITULO;
```

Figura 5.6: Tipo agregação com versão

5.1.2 Versões e Agregação

Versões de objetos compostos por agregação, diferentemente de versões em hierarquias de generalização, aparecem com freqüência nas propostas de sistemas de bancos de dados para suporte ao projeto

(/ZDO 84/, /ATW 85/, /DIT 86b/, /CHO 88/). Muitas dessas propostas fornecem ao projetista o suporte para manutenção de versões de objetos de projeto e deixam para o projetista a responsabilidade da semântica dos relacionamentos entre as diversas versões de objetos no plano de versões. Esta abordagem, embora possibilite total liberdade ao projetista de compor versões indistintamente, resulta em planos de versões muitas vezes confusos para o usuário acompanhar e que não levam em consideração os relacionamentos semânticos dos objetos genéricos.

No BD_PAC, cada versão de um tipo agregado só pode ter versões de um tipo definido como componente. Isto quer dizer que no plano de versões de hierarquias de agregação são observados os relacionamentos de composição entre os objetos genéricos. Um exemplo de declaração de uma agregação com versões é mostrado na figura 5.6. Suponha-se que um objeto TESE1 do tipo TESE seja composto pelo objeto CAPA1 do tipo CAPA e pelos objetos CAPIT1, CAPIT2 e CAPIT3 do tipo CAPITULO (figura 5.7). Uma versão do objeto TESE1 poderia compor-se somente do objeto CAPA1 e de quaisquer versões dos objetos CAPIT1, CAPIT2 e CAPIT3. Esta abordagem, embora restrinja um pouco a liberdade de projetistas, é mais intuitiva e possibilita ao sistema de banco de dados um controle melhor do plano de versões de objetos.

No BD_PAC, uma versão pode ser agregado de muitas outras e, ainda, uma versão pode ser componente de várias outras. Observe-se o exemplo da figura 5.8. No exemplo, o objeto genérico TESE1, do tipo TESE declarado na figura 5.6, é composto por um capítulo CAPIT1 do tipo CAPITULO. TESE1 tem duas versões V1 e V2, enquanto que CAPIT1 tem três versões V1, V2 e V3. A versão TESE1.V1 poderia compor-se das versões CAPIT1.V1 e CAPIT1.V2 e, ao mesmo tempo, a versão TESE1.V2 poderia compor-se das versões CAPIT1.V2 e CAPIT1.V3. A versão CAPIT1.V2 seria portanto compartilhada pelas versões TESE1.V1 e TESE1.V2. Um cuidado especial é dedicado às versões compartilhadas no momento de excluir versões agregadas da base. Quando da exclusão de uma versão agregada como um todo, aquelas versões compartilhadas não são excluídas da base de dados.

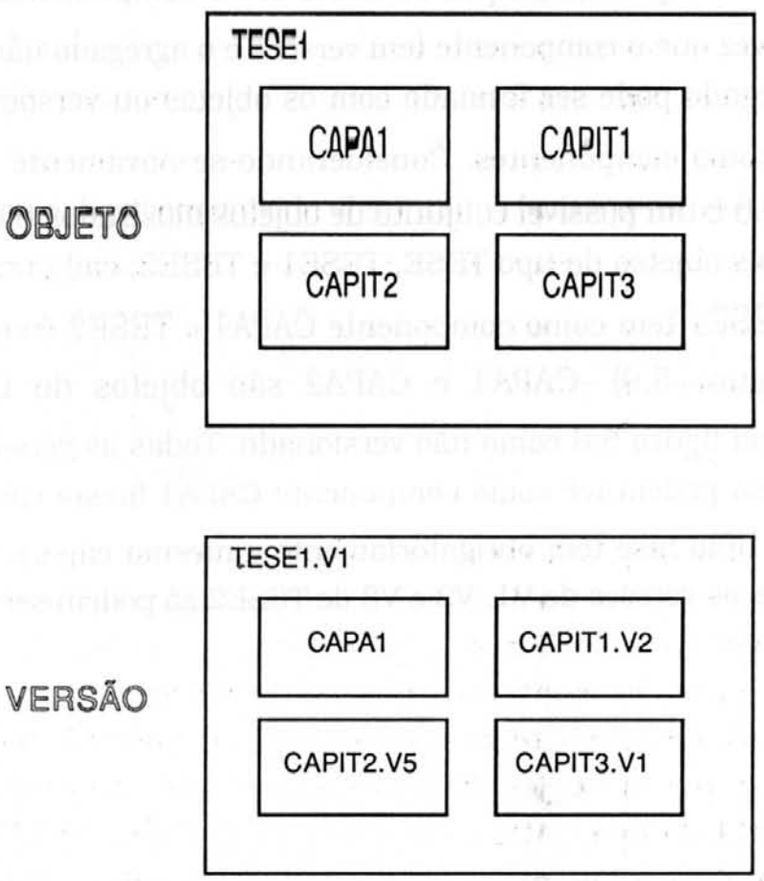

Figura 5.7: Versões de agregações.

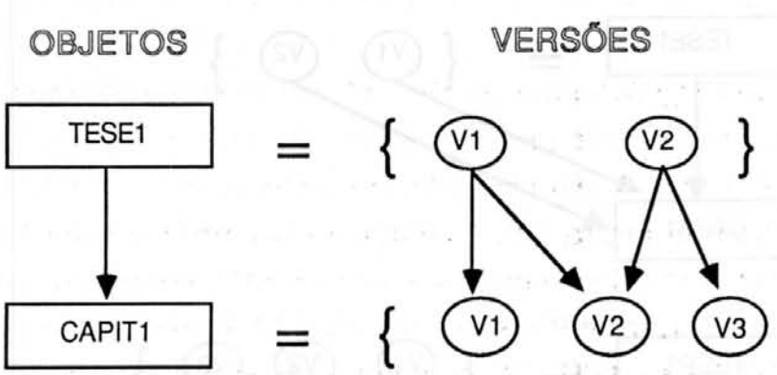

Figura 5.8: Compartilhamento de versões em agregações

A Quando o agregado é um objeto versionado e seus componentes não são, todas as versões dos objetos agregados são

formadas com objetos dos tipos definidos como componentes. Por outro lado, uma vez que o componente tem versões e o agregado não tem, cada objeto agregado pode ser formado com os objetos ou versões dos tipos definidos como componentes. Considerando-se novamente o esquema da figura 5.6 e um possível conjunto de objetos mostrados na figura 5.9. Existem dois objetos do tipo TESE, TESE1 e TESE2, cada um contendo versões. TESE1 tem como componente CAPA1 e TESE2 é composto de CAPA2 (figura 5.9). CAPA1 e CAPA2 são objetos do tipo CAPA, declarado na figura 5.6 como não versionado. Todas as versões V1 e V2 de TESE1 só podem ter como componente CAPA1 (neste caso todas as versões de uma tese têm obrigatoriamente a mesma capa). Do mesmo modo, todas as versões de V1, V2 e V3 de TESE2 só podem ser compostas por CAPA2.

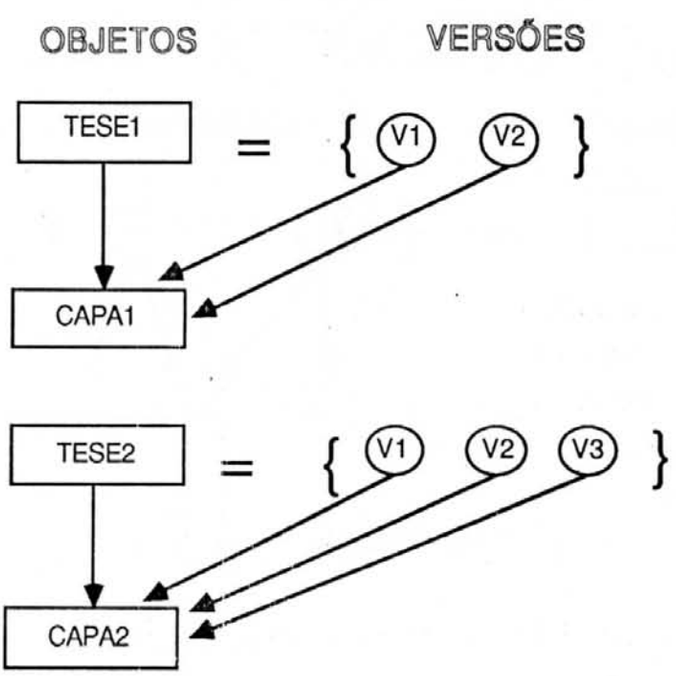

Figura 5.9: Versões de agregados e componentes

A restrição do relacionamento implícito de composição também se aplica às hierarquias de agregações no plano de versões. Versões de tipos agregados devem obedecer às restrições impostas ao tipo correspondente. Por exemplo, quando um tipo MOTOCICLETA está restrito a ter duas rodas, todas as versões de MOTOCICLETA também estarão restritas a somente duas rodas.

5.1.3 Versões e Associação

Assim como os tipos construídos pela generalização e pela agregação podem ser tipos versionados, os conjuntos criados pela associação também podem ter versões.

```
SET TYPE CONJ
    ATTRIBUTES
        Attrib1: LONG;
        Attrib2: AVG ( ELEM . Att1 )
    VERSIONS TREELIKE
    MEMBERS
        ELEM
END CONJ

OBJECT TYPE ELEM
    ATTRIBUTES
        Att1: INT
    VERSIONS TREELIKE
END ELEM;
```

Figura 5.10: Declaração de um conjunto

Uma versão de uma associação é formado possivelmente pelas versões dos objetos do tipo declarado como membro. A figura 5.10 mostra a declaração de um tipo versionado. Suponha-se que num determinado instante existem objetos e versões na base de dados como mostra a figura 5.11. O objeto OBJ1 do tipo associação CONJ tem como membros os objetos OBJ2 e OBJ3 do tipo ELEM. As versões de CONJ têm, como membros, versões dos objetos OBJ2 e OBJ3.

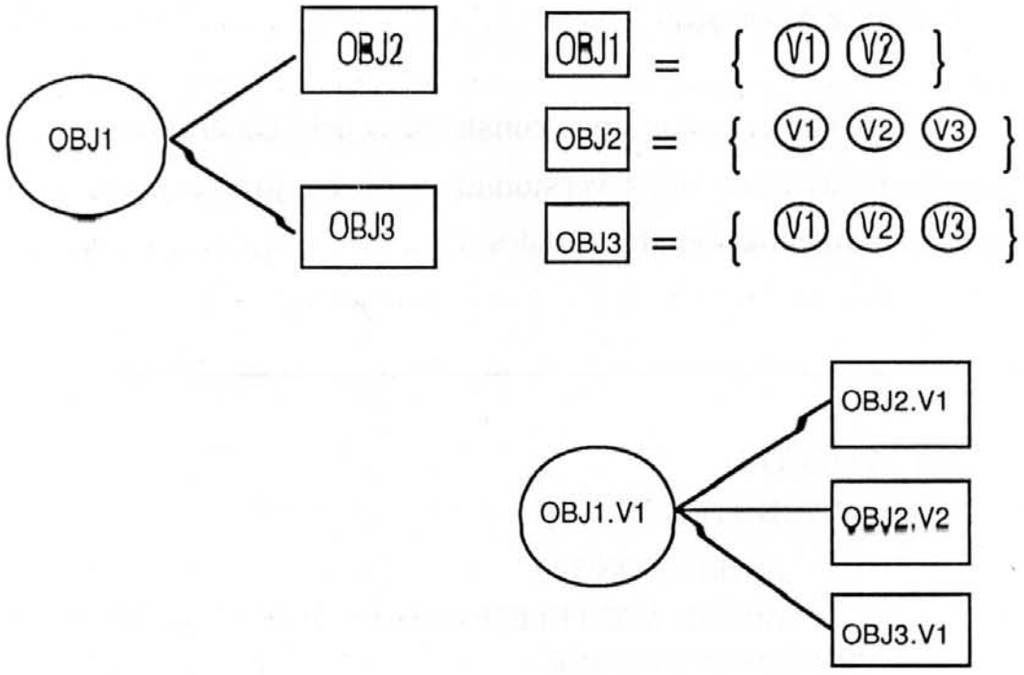

Figura 5.11: Objetos de conjuntos versionados

Para conjuntos definidos como versionados e que têm elementos não versionados, as versões do conjunto terão, como elementos, objetos do tipo declarado como membro. Imagine-se que o tipo ELEM da declaração acima não seja tipo versionado. Neste caso todas as versões do objeto OBJ1 da figura 5.11 seriam conjuntos que teriam como domínio apenas o conjunto formado pelos objetos OBJ2 e OBJ3 do tipo ELEM.

Dentro do plano de versões de uma hierarquia de associação, versões do conjunto podem ligar-se com versões dos

elementos indistintamente. Isto quer dizer que uma versão de um elemento pode participar de várias versões do conjunto e, ainda mais, uma versão de um conjunto pode ter um número ilimitado de versões dos seus elementos.

A derivação de atributos do conjunto, possível para associações de objetos, também é perfeitamente aplicável às versões dos conjuntos associação. Na declaração da figura 5.10, pela herança de estrutura dos objetos para as suas versões (ver seção 5.1), o atributo "Attrib2" é válido para todas as versões de objetos do tipo CONJ e deverá ter valor calculado pela média - função pré-definida "AVG" - dos valores do atributo "Att1" para as versões que são elementos do conjunto específico. Observe-se o exemplo da figura 5.11. O atributo "Attrib2" da versão OBJ1.V1 é calculado levando-se em consideração os valores do atributo "Att1" das versões que formam o conjunto.

5.2 Considerações Finais

Neste capítulo foi discutida a semântica do plano de versões e a sua integração com o modelo de dados. Inicialmente, foi feita uma breve revisão de algumas propostas de planos de versões em sistemas de bancos de dados para ambientes de projeto. Em seguida, os aspectos conceituais da abordagem do BD_PAC para o plano de versões foram detalhadamente descritos. O plano de versões foi mostrado de acordo com o modelo de dados do BD_PAC. Para cada conceito de abstração do modelo foram relatadas as características próprias das versões de objetos correspondentes à abstração.

O resultado desta proposta é um plano de versões flexível, mas que não é tão liberal quanto algumas propostas de versões de sistemas de bancos de dados orientados a objetos vistos no início deste capítulo (/CHO 88/, /DIT 86b/). Por outro lado, o mecanismo não é tão rígido quanto às abordagens de modelos dedicados de /BAT 85/ e /KAT 83/. Uma vez que as características semânticas dos objetos

genéricos são levadas em consideração no plano de versões, a fidelidade na representação semântica de uma versão de um objeto é mantida com mais facilidade pelo sistema de banco de dados e pelo próprio usuário do sistema que tem numa versão uma alternativa para o objeto genérico.

6 IMPLEMENTAÇÃO

O Modelo BD_PAC foi implementado como uma camada sobre a versão 2.0 do Sistema de Gerência de Banco de Dados DAMOKLES. O protótipo executa sobre o sistema operacional SUNOS versão 4.0 (compatível com o sistema operacional UNIX da Universidade de BERKELEY) da empresa SUN MICROSYSTEMS. O ambiente de execução é o fornecido por uma estação de trabalho tipo SUN SPARCSTATION 1. O protótipo foi totalmente desenvolvido na Linguagem C /KER 78/.

O Sistema DAMOKLES possui dois componentes principais. Um componente de reconhecimento da linguagem de definição dos dados formado pelo compilador do DAMOKLES e o componente da linguagem de manipulação, formado por uma biblioteca de operadores de banco de dados que têm interface embutida na linguagem C /KER 78/. O BD_PAC estende a gramática do compilador para que sejam reconhecidos os novos construtores de tipos explicados no capítulo 4, e implementa, seja por redefinição dos operadores já existentes do DAMOKLES, seja por criação de novos operadores, uma biblioteca de procedimentos da linguagem C que permite a manipulação dos objetos do novo modelo.

As seções seguintes fornecem um quadro geral dos aspectos de implementação do BD_PAC. Inicialmente, é mostrada a arquitetura do sistema. A próxima seção detalha a implementação do compilador da linguagem de definição do novo modelo. Em seguida, a linguagem de manipulação é explicada através de uma descrição suscinta dos operadores da linguagem. Finalmente, são traçados alguns aspectos de operacionalidade, onde se explica como utilizar o novo sistema.

6.1 Arquitetura do Sistema

O BD_PAC é implementado como uma camada sobre o

Sistema DAMOKLES. Esta camada é formada por um dicionário de dados, onde são armazenadas informações semânticas sobre os objetos manipulados no sistema, e uma biblioteca de operadores que interagem diretamente com os operadores do próprio DAMOKLES e fazem acesso às informações do dicionário de dados do BD_PAC (figura 6.1)

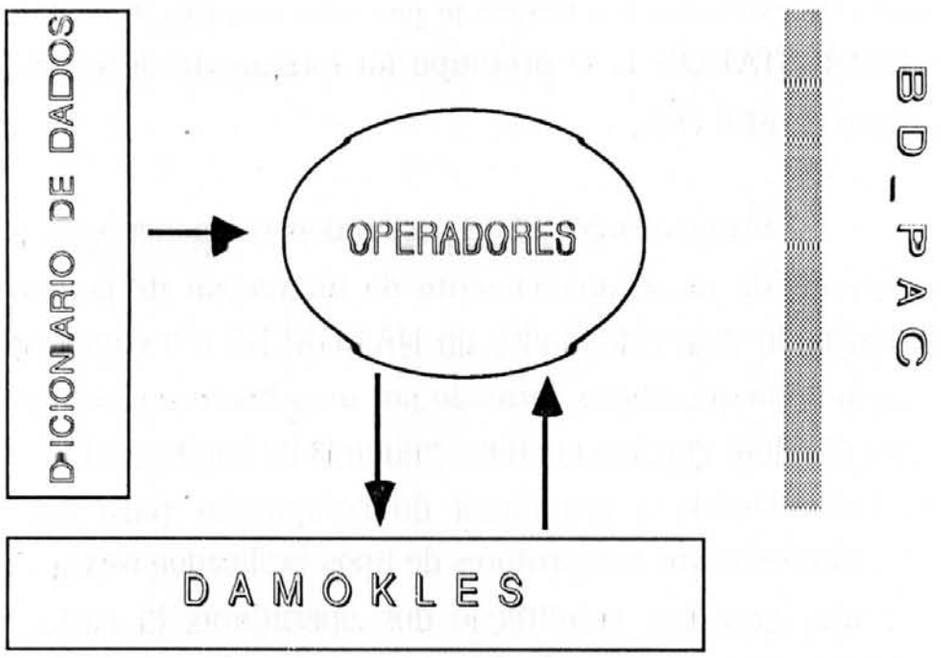

Figura 6.1: Arquitetura do BD_PAC

Embora se utilizem largamente da funcionalidade do DAMOKLES, os operadores do BD_PAC percebem este sistema como uma "caixa preta", enxergando apenas a interface do sistema. Este aspecto resulta numa arquitetura modular e fornece um grau considerável de independência na implementação do sistema.

Todos os tipos e objetos do BD_PAC são traduzidos para tipos e objetos do DAMOKLES. Os operadores do BD_PAC funcionam como um filtro entre a aplicação e os operadores do DAMOKLES. Quando acionados, os operadores do BD_PAC buscam informações do dicionário de dados e chamam os operadores do DAMOKLES, fornecendo ao usuário o objeto com o significado semântico adequado.

6.1.1 O Dicionário de Dados

As informações sobre os tipos de um esquema BD_PAC ficam armazenadas no dicionário de dados do sistema. Este repositório de dados tem como função armazenar todas as ligações semânticas estruturais dos tipos de um esquema de aplicação.

O dicionário de dados é implementado como um banco de dados do DAMOKLES. As entradas do dicionário são realizadas no momento do reconhecimento de um esquema de aplicação (ver seções seguintes). Não há necessidade de manter todas as informações sobre os tipos do esquema já que grande parte delas fica guardada no dicionário de dados do DAMOKLES. No dicionário do BD_PAC são mantidas informações necessárias apenas à correta interpretação dos tipos e construtores de tipos fornecidos pelo modelo.

O dicionário de dados é gerado automaticamente pelo sistema quando este é configurado para utilização. Neste instante, o compilador interpreta as informações do esquema do dicionário de dados, chamado de metaesquema, e o sistema cria um banco de dados associando-o ao esquema compilado. O metaesquema é mostrado em detalhe no anexo 2 deste volume. A figura 6.2 mostra uma descrição diagramática suscinta do metaesquema.

Edbschema: tipo estruturado que mantém informações gerais sobre os esquemas da aplicação. Um objeto deste tipo armazena um identificador, um nome e a localização do esquema representado. Um campo especial do tipo "LONG_FIELD" (capítulo 3) guarda uma representação compacta de todo o metaesquema. Este campo deverá ser mais tarde trazido para a memória no momento da utilização do banco de dados, otimizando assim, consultas ao metaesquema. Cada esquema compilado pelo usuário do BD_PAC vai ter um objeto correspondente neste tipo com o mesmo nome do esquema.

Object_type: este tipo contém objetos que armazenam informações

específicas de cada tipo de objeto do esquema. Cada objeto contém o nome, o identificador, uma indicação de tipo versionado e qual o tipo de versão, um campo para o tipo do tipo (simples, relacionamento, construído por agregação, generalização ou associação) e o comprimento do "buffer" de atributos do tipo representado.

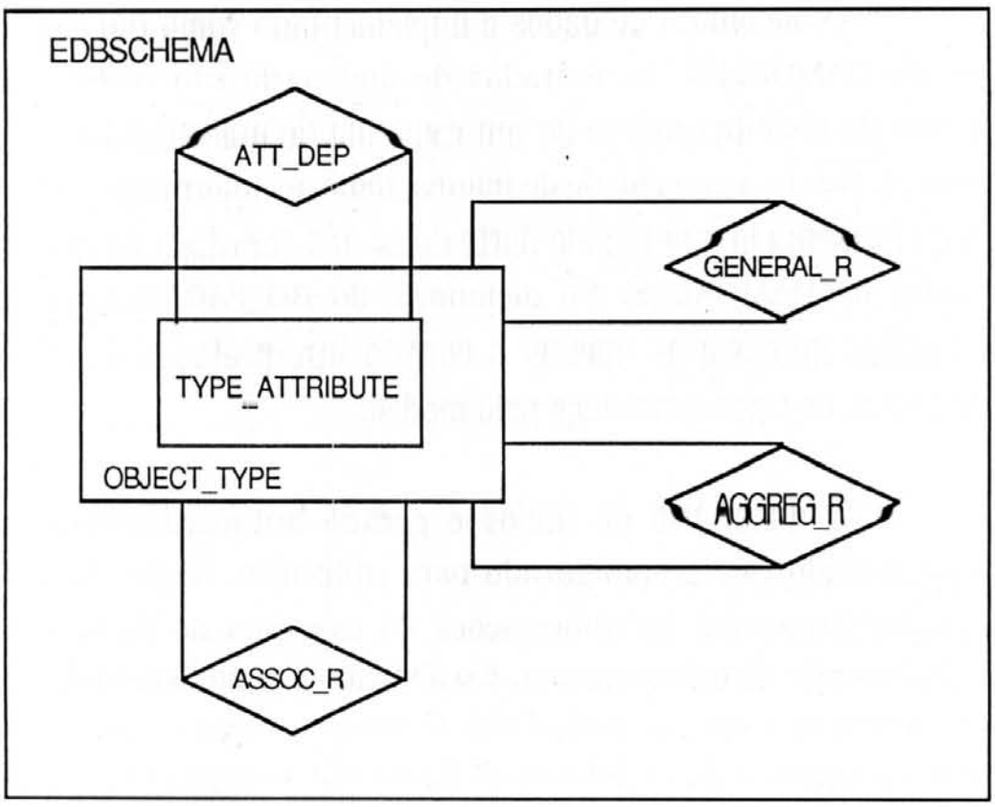

Figura 6.2: O metaesquema

Type_attribute: os objetos de *type_attribute* armazenam informações sobre todos os atributos para cada tipo do esquema. Os objetos mantêm o nome, o identificador, o domínio e uma indicação de atributo calculado automaticamente pelo sistema.

Generalization_r: armazena dados de relacionamentos estruturais de hierarquias de generalização. É um auto-relacionamento de *Object_type* onde um objeto participa com o papel de supertipo e o outro com o papel de subtipo.

Aggregation_r: guarda relacionamentos semânticos de agregação entre tipos do esquema. Um dos objetos do relacionamento é o agregado e o outro é o componente. Mantém informações sobre a cardinalidade do relacionamento semântico e o tipo do componente (objeto ou relacionamento).

Association_r: mantém informações sobre os relacionamentos de associação entre os tipos. Um objeto participa com o papel de conjunto, enquanto que o outro é o membro.

Att_dependence_r: auto-relacionamento de *type_attribute* que determina dependência entre atributos para aqueles que são calculados automaticamente pelo sistema. Um objeto do tipo *type_attribute* assume o papel de definidor enquanto que o outro é o definido. O relacionamento armazena, ainda, a função de cálculo do atributo.

6.2 A Linguagem de Definição dos Dados

A estrutura dos dados de um banco de dados do BD_PAC é declarada por um esquema que descreve os tipos de objetos e os relacionamentos semânticos entre os tipos. Um esquema é uma descrição textual formal que segue a gramática do Anexo 1. O esquema BD_PAC compõe-se de três grandes partes:

- *Declaração de constantes*: declaração de todas as constantes utilizadas no esquema;
- *Declaração de domínios*: permite a construção de domínios (tipos de variáveis na nomenclatura de linguagens de programação) de atributos a partir daqueles já fornecidos pelo sistema;
- *Declaração de tipos*: componente onde são declarados todos os tipos, simples ou construídos, do esquema. Os atributos dos tipos podem ter domínios pré-definidos ou construídos na declaração de domínios.

O esquema é uma estrutura estática. Uma vez definido e associado a um banco de dados, um esquema não pode mudar. No BD_PAC, os programas de aplicação importam algumas informações do esquema em tempo de compilação. Se o esquema for posteriormente alterado, todos os programas que utilizam o banco de dados associado àquele esquema precisam ser alterados e recompilados. Quanto aos dados já cadastrados, eles perdem sua estrutura original, sem que seja possível recuperá-los através do novo esquema. Quando houver necessidade de alterar o esquema de um banco de dados já existente, pode-se criar um novo banco de dados e associá-lo ao novo esquema. A partir daí, um programa da aplicação transforma e transfere os dados do velho para o novo banco de dados. Quanto aos programas, é impossível fugir da recompilação.

6.2.1 O Reconhecimento

O reconhecedor do BD_PAC é gerado automaticamente a partir de uma especificação gramatical, semelhante àquela mostrada no anexo 1. A gramática é descrita na forma de uma BNF (Backus-Naur Form) estendida com ações semânticas e um mecanismo de passagem de atributos entre as produções. Esta gramática serve então de entrada para um utilitário que gera reconhecedores, chamado YACC ("Yet Another Compiler-Compiler") /JOH 75/. O YACC cria uma função capaz de reconhecer um texto que obedece a especificação gramatical. Porém esta função não é capaz de ler o arquivo fonte e buscar os *tokens* do texto. Por este motivo, existe uma segunda especificação que complementa a gramática do YACC e serve de entrada para um outro utilitário, chamado LEX ("Lexical Analyser Generator") /LES 75/. O LEX gera uma função que trabalha em conjunto com aquela gerada pelo YACC, buscando os *tokens* do texto fonte. A figura 6.3 mostra o funcionamento dos dois utilitários juntos.

As funções geradas pelo LEX e YACC são incorporadas ao compilador do BD_PAC, onde estão codificadas as ações semânticas.

Estas têm as seguintes funções:

- Alimentar uma estrutura de dados (chamada doravante de estrutura de dados do reconhecedor) a partir do fonte, contendo todas as informações do esquema;
- Fazer a análise semântica do texto reconhecido.

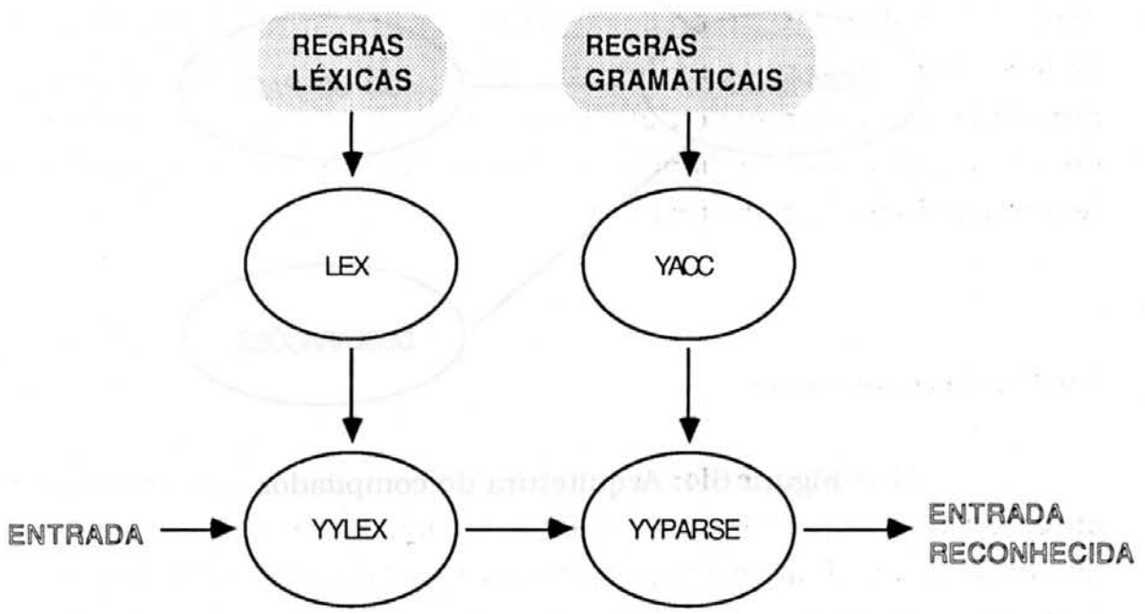

Figura 6.3: LEX e YACC

Uma vez que o esquema fonte foi reconhecido com sucesso, o sistema então percorre a estrutura de dados do reconhecedor e alimenta os dicionários de dados do DAMOKLES e do BD_PAC. Grande parte das informações da estrutura de dados do reconhecedor fica no dicionário de dados do DAMOKLES. Uma outra parte, que implementa a extensão do modelo de dados, vai para o dicionário do BD_PAC.

Uma vez que o dicionário de dados do BD_PAC é um banco de dados DAMOKLES, seus objetos são inseridos utilizando-se a

linguagem de manipulação do DAMOKLES.

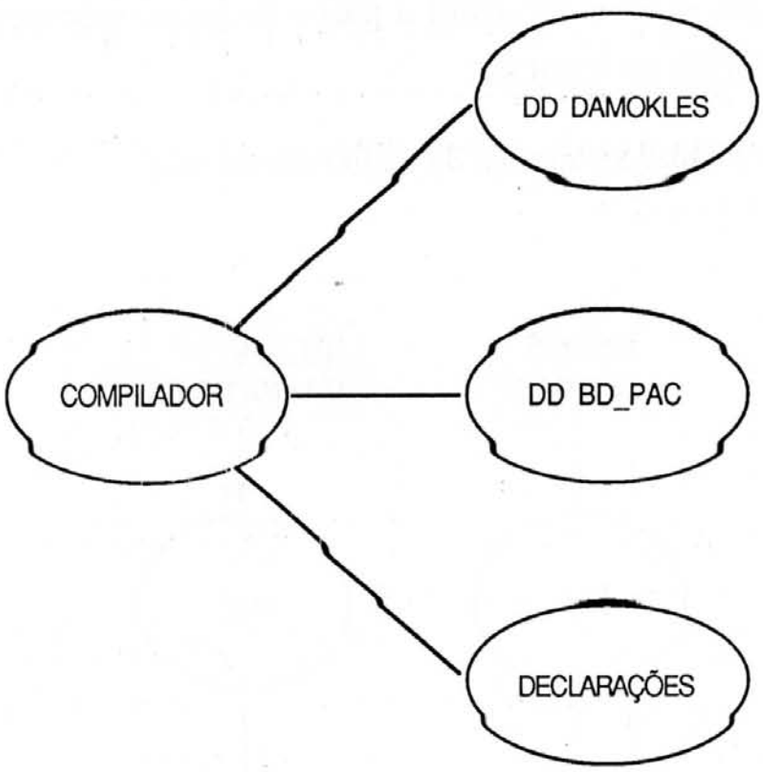

Figura 6.4: Arquitetura do compilador

6.2.2 Expansão das Definições dos Subtipos

O compilador do BD_PAC tem como saída as informações dos dicionários de dados do DAMOKLES e do BD_PAC, bem como a criação de um arquivo fonte em linguagem C contendo declarações para as constantes, domínios de atributos e tipos de objetos do esquema compilado (figura 6.4). Este arquivo fonte C precisa, mais tarde, ser incluído nos programas de aplicação que utilizam o banco de dados referente ao esquema. O anexo 3 contém a listagem de um arquivo de declarações gerado pelo compilador.

O arquivo de declarações, entre outra coisas, contém declarações *typedef* da linguagem C, uma para cada tipo do esquema.

Com isto, um tipo do esquema é mapeado diretamente para um tipo construído da linguagem C, formando um registro com estrutura idêntica àquela do tipo do banco de dados. O programa de aplicação tem que declarar uma variável de programa do tipo definido pelo *typedef* para ser utilizada como área de comunicação entre o espaço de dados do programa e o banco de dados. A figura 6.5 mostra um fragmento de um arquivo de declarações.

ESQUEMA	DECLARAÇÕES
OBJECT TYPE Exemplo	typedef struct
ATTRIBUTES	{
Att1: Dom1;	Dom1 Att1;
Att2: Dom2	Dom2 Att2;
END Exemplo;	} Exemplo

Figura 6.5: Um exemplo de declaração

Ocorre que, numa hierarquia de generalização, os subtipos herdam as propriedades dos supertipos. Por outro lado, a geração do arquivo de declarações leva em consideração apenas os atributos de cada tipo especificamente. Isto torna a definição de um tipo pelo *typedef*, uma declaração de espaço insuficiente para receber todos os atributos do subtipos, aqueles próprios e os herdados. A solução veio com a implementação de um algoritmo recursivo para a expansão das declarações dos subtipos no arquivo C gerado.

A herança de propriedades não se materializa somente no nível mais baixo da hierarquia de generalização. A cada nível, os tipos são expandidos pelos atributos declarados até aquele ponto da hierarquia, de forma recursiva, até que seja alcançado o nível mais

baixo, onde não há mais herança de propriedades.

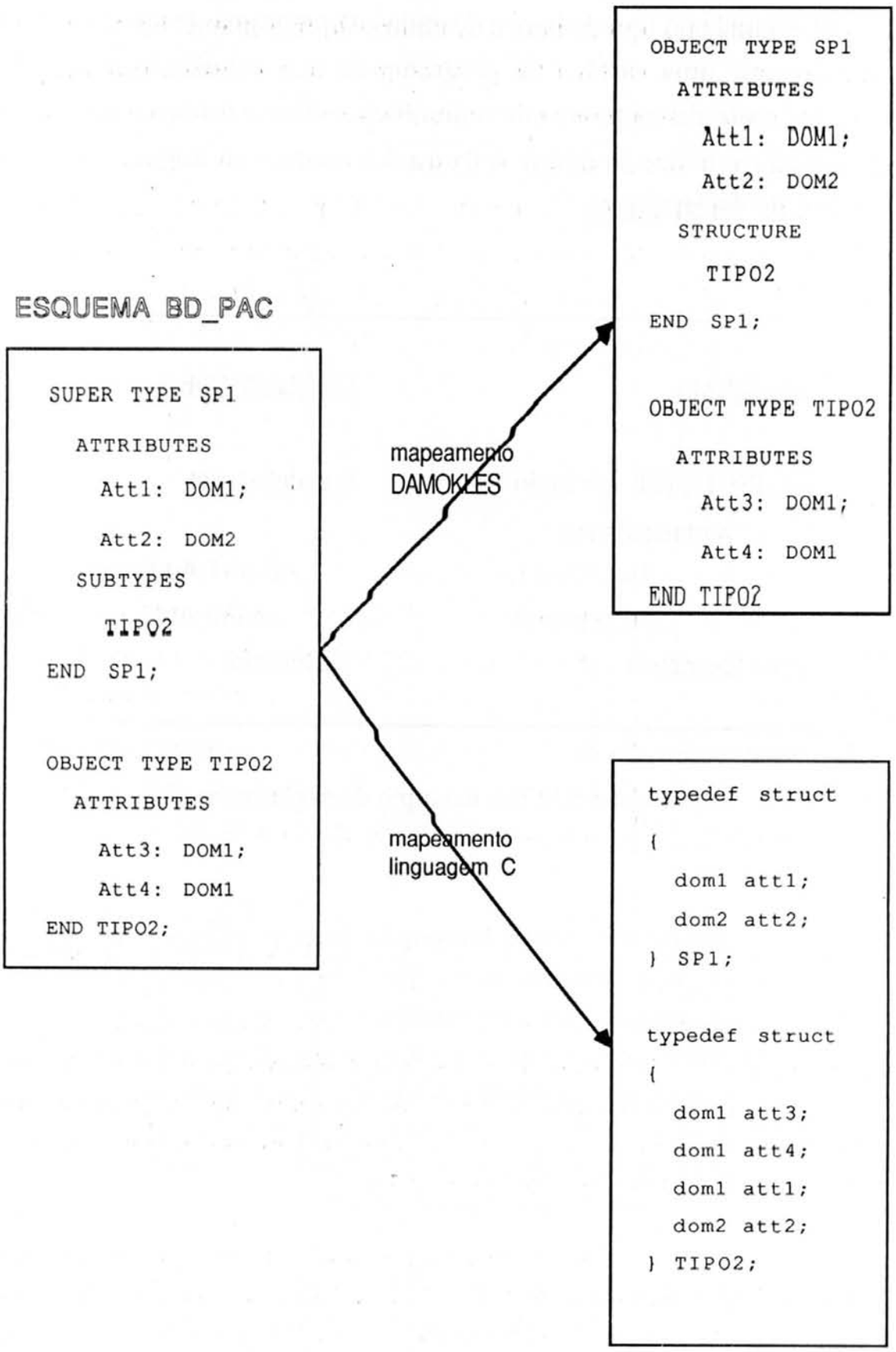

Figura 6.6: Mapeamento de generalizações

A expansão do arquivo de declarações é necessária devido à forma como está implementado o conceito de generalização. Os tipos definidos numa hierarquia de generalização são mapeados para tipos estruturados do DAMOKLES, mantendo a sua estrutura original (observe-se a figura 6.6). No momento da recuperação dos objetos dos subtipos, o sistema caminha automaticamente em direção ao nível mais alto, buscando os valores dos atributos em cada nível. Todos os dados coletados são, então, passados para a área de comunicação fornecida pelo programa de aplicação - a variável de programa declarada como sendo do subtipo específico.

6.3 A Linguagem de Manipulação

A linguagem de manipulação é caracterizada como uma interface "um objeto por vez". Isto quer dizer que cada operação da linguagem retorna apenas um objeto ao programa chamador. A aplicação pode fazer acesso e manipular os dados armazenados em um banco de dados pela navegação de objeto para objeto através de relacionamentos semânticos ou genéricos do modelo.

Os objetos são identificados no BD_PAC através de uma chave interna, única em todo o sistema e por todo o tempo, chamada de *surrogate* (o conceito de *surrogate* é implementado pelo DAMOKLES). Operações que localizam objetos retornam, para o programa chamador, o *surrogate* do objeto encontrado. A aplicação tem a possibilidade, então, de armazenar o *surrogate* em uma variável de programa no seu próprio espaço de endereços e utilizá-lo para leitura do objeto ou como referência em passos futuros de navegação.

A linguagem de manipulação do BD_PAC funciona como uma aplicação avançada do DAMOKLES. Todos os tipos e objetos são traduzidos para tipos e objetos do DAMOKLES. Os operadores da linguagem funcionam como um filtro, obtendo informações do dicionário de dados BD_PAC, buscando os objetos do DAMOKLES e

fornecendo ao usuário o objeto com o significado semântico adequado.

A maioria dos operadores foi redefinida e implementada como um conjunto de operadores do DAMOKLES. O mapeamento, no caso de hierarquias de generalização, é invariavelmente de 1 para N, onde 1 representa o número de operações do BD_PAC - a interface externa do modelo - e N o número de operações no DAMOKLES. A recuperação de um objeto que herda atributos de outros objetos é um ótimo exemplo para esse mapeamento. A uma operação destas no BD_PAC correspondem várias do DAMOKLES, cada uma buscando os objetos da hierarquia de generalização.

O BD_PAC não somente redefine alguns operadores do DAMOKLES, mas também fornece alguns operadores novos. Um deles é o operador para pesquisa por valor de atributo - busca associativa. O usuário, agora, tem a sua disposição uma forma de pesquisar os objetos baseada em valores dos atributos do objeto, por exemplo, uma chave externa (conjunto de atributos definido pelos usuários, cujos valores identificam univocamente um objeto na base de dados). No DAMOKLES, a recuperação dos objetos tem por base o identificador interno, chave que não faz sentido como valor para o usuário. Neste caso, ou se busca diretamente o objeto uma vez que o *surrogate* está disponível, ou se faz uma pesquisa seqüencial até que seja achado o objeto que se está procurando. Utilizando-se o operador de busca associativa, a aplicação pode recuperar qualquer objeto que tenha qualquer atributo com um valor especificado.

O operador de pesquisa associativa leva em consideração a hierarquia em que o objeto está inserido. Isto quer dizer que o conjunto de objetos pesquisados pode ser restrito a apenas um subconjunto do conjunto de todos os objetos do tipo. Portanto podem-se fazer duas formas de pesquisa associativa: (1) dentre todos os objetos, quais aqueles que satisfazem determinada condição; (2) dentre os objetos componentes de um agregado específico, quais os que satisfazem determinada condição.

O Sistema DAMOKLES fornece um meio para pesquisar objetos de acordo com valores de atributos (capítulo 3). O sistema oferece operadores que recebem uma expressão de pesquisa (uma cadeia de caracteres contendo uma expressão parecida com expressões de uma SQL) e devolvem um *cursor* - conjunto de *surrogates* dos objetos que satisfazem a expressão passada como parâmetro. As expressões de pesquisa obedecem a uma gramática especial definida no manual do Sistema DAMOKLES versão 2.0 /ABR 88/. O operador de pesquisa associativa do BD_PAC é mais simples de ser utilizado porque o usuário não necessita aprender uma nova linguagem (aquela definida pela gramática das expressões de pesquisa do DAMOKLES), e ainda assim dispõe de um meio poderoso para pesquisar objetos por valor, pois a pesquisa no BD_PAC leva em consideração os relacionamentos hierárquicos semânticos entre os objetos.

Em seguida são mostradas, para cada grupo de operadores, as diferenças básicas entre os operadores do DAMOKLES e do BD_PAC. Esta classificação dos operadores é realizada no manual de utilização do Sistema DAMOKLES versão 2.0 /ABR 88/. A lista abaixo não tem como objetivo mostrar todos os detalhes de implementação de cada operador do BD_PAC, mas sim mostrar um quadro genérico sobre o que mudou do DAMOKLES para o BD_PAC em termos de operadores. Uma descrição detalhada da implementação dos operadores é dada em /FOR 90/.

Operadores sobre banco de dados: foram redefinidos porque há a necessidade de abrir o dicionário de dados do BD_PAC e buscar suas informações quando da utilização de um banco de dados;

Operadores sobre objetos: operadores de inclusão e exclusão de objetos foram totalmente modificados. Eles precisam buscar informações semânticas sobre o tipo de objeto manipulado e agir conforme a semântica de cada tipo. Os operadores para inclusão e exclusão do relacionamento semântico entre objetos foram, a partir de um só operador no DAMOKLES, expandidos para três operadores, um para cada construtor de tipo do BD_PAC. Os operadores de navegação em hierarquias de objetos também

foram expandidos numa relação de 1 para 3, ou seja, para cada operador do DAMOKLES, 3 foram criados a fim de refletir a semântica do objeto manipulado;

Operadores sobre relacionamentos: esses operadores manipulam relacionamentos genéricos entre objetos. Este tipo de relacionamento é suportado no BD_PAC de forma idêntica ao DAMOKLES. A princípio, não há qualquer necessidade de mudar estes operadores;

Operadores sobre atributos: precisaram ser redefinidos a fim de perceberem a herança de atributos de supertipos para subtipos e suportarem a derivação de atributos dos tipos associação. O cálculo de um atributo em hierarquias de associação é realizado no momento da recuperação do atributo;

Operadores sobre campos longos: não sofrem modificações porque os campos longos têm manipulação independente dos demais atributos de um objeto, salvo o operador que abre um campo longo, devido à herança de atributos. Todo o acesso a um campo longo é feito através de um *descritor de campo longo* criado pelo sistema;

Operadores de navegação baseada em relacionamentos: estes operadores estão divididos em dois grandes grupos: (1) aqueles que apenas procuram um objeto tendo como base identificadores internos. Estes retornam somente o identificador. (2) Aqueles que buscam o objeto e têm como retorno o identificador interno e os atributos do objeto. Os operadores de (2) foram redefinidos devido à diferença semântica de atributos entre os construtores de tipos - herança nas hierarquias de generalização e cálculo nas hierarquias de associação;

Operadores sobre versões: o grafo de versões do BD_PAC é mantido de forma similar ao DAMOKLES. Por outro lado, os operadores para inserção e remoção de versões foram totalmente modificados. Eles agora levam em consideração as propriedades semânticas de cada tipo declarado como versionado;

Operadores sobre múltiplos bancos de dados: as operações de início e de fim de transação de projeto continuam as mesmas. Isto não acontece com as operações de cópia e transferência de objetos entre bancos de dados e as operações de requisição e liberação de objeto. O raio de alcance desses operadores sobre os objetos depende do tipo do objeto, variando de um tipo construído para outro. As operações de requisição e liberação também sofrem influência direta da semântica do objeto que está sendo manipulado, logo foram totalmente redefinidas.

6.4 Aspectos de Operacionalidade

Esta seção tem como objetivo fornecer um guia geral para o desenvolvimento de aplicações baseadas no BD_PAC. Inicialmente, serão abordadas algumas características de inicialização do sistema. Em seguida, o objeto de discussão passa a ser o desenvolvimento de aplicações que utilizam o BD_PAC como sistema gerenciador de banco de dados. O objetivo aqui não é fornecer um manual de utilização do sistema, mas traçar um quadro geral que possibilite a utilização dos conceitos básicos do BD_PAC. Quando houver necessidade de maiores detalhes, o leitor pode consegui-los no manual da versão 2.0 do Sistema DAMOKLES /ABR 88/.

6.4.1 Inicialização do Sistema

O BD_PAC é dividido em três grandes componentes: dois programas, chamados de DB_ADMIN e DB_USER, e uma biblioteca de procedimentos que contém os operadores de banco de dados.

DB_ADMIN: o DB_ADMIN é um programa para ser utilizado somente pelo administrador de banco de dados. Ele contém funções de configuração e inicialização do sistema, bem como procedimentos para controle de acesso aos bancos de dados. Estes

últimos possibilitam o cadastramento de usuários e grupos de usuários;

DB_USER: é o compilador do BD_PAC. O DB_USER tem funções para compilar esquemas e criar bancos de dados;

DAMLIB: conjunto de procedimentos que realiza o gerenciamento do banco de dados. O usuário de aplicação deve incluir esta biblioteca na compilação do seu programa a fim de poder utilizar os operadores de banco de dados.

A operação de inicialização do sistema deve ser o primeiro passo a ser realizado para que o BD_PAC seja utilizado. Durante a inicialização, são definidos alguns parâmetros globais do sistema e criados os dicionários de dados do DAMOKLES e do BD_PAC. O sistema cria, no diretório corrente, um arquivo de configuração e um diretório chamado DAM_DATA, onde serão armazenados todos os dados da aplicação.

A inicialização do sistema deve ser realizada somente por pessoal autorizado - o administrador de banco de dados - porque, caso haja quaisquer dados da aplicação previamente gravados, eles serão todos destruídos.

Uma vez inicializado o sistema, o administrador deve cadastrar os usuários autorizados a fazer acesso aos dados da aplicação e os seus respectivos grupos, quando houver necessidade destes. O sistema então estará pronto para ser utilizado pelos usuários da aplicação.

6.4.2 Utilizando o Sistema

Após a finalização do trabalho do administrador de banco de dados, o usuário de aplicação precisa realizar os seguintes passos, a fim de poder trabalhar com os seus dados:

- (1) Compilar o esquema do banco de dados;
- (2) Criar um ou mais bancos de dados de acordo com o esquema;
- (3) Compilar o programa de aplicação e ligá-lo à biblioteca de funções **damlib.a**.

O usuário de aplicação utiliza o programa DB_USER para realizar os passos (1) e (2). A biblioteca **damlib.a** é fornecida pelo sistema. A aplicação estará pronta para entrar em operação.

Um programa de aplicação tem que incluir três arquivos para fazer acesso e manipular dados armazenados em um banco de dados BD_PAC:

- O arquivo "edb.h" contém definições de constantes e tipos que representam conceitos genéricos da linguagem de manipulação (*surrogate*, descritores, códigos de retorno);
- O arquivo "edb_com.e" contém os protótipos - declaração da interface de uma função da linguagem C a fim de que seja possível utilizá-la em outros módulos - de cada função da interface que representa um operação da linguagem de manipulação. Cada função dessas retorna um valor indicando a finalização correta da operação ou o tipo de erro ocorrido;
- O arquivo "db_<nome_esquema>.h" contém o resultado da tradução do esquema, ou seja, o mapeamento dos tipos declarados no esquema BD_PAC para estruturas de dados da linguagem C (ver seção 6.2.2).

Para obter maiores detalhes sobre o conteúdo destes arquivos, o leitor deve ir direto ao manual do DAMOKLES - versão 2.0 /ABR 88/.

7 APLICAÇÃO DO MODELO

Este capítulo tem por objetivo mostrar a validade do modelo BD_PAC através da modelagem dos dados de dois ambientes de projeto em desenvolvimento na UFRGS, o Sistema AMPLO (AMbiente de Projeto LÓgico de Sistemas Digitais) (/BOK 87/, /WAG 87a/, /GOL 88/, /BEC 88/, /WAG 87b/, /GOL 89a/ /WAG 89/, /GOL 89b/), para projeto de sistemas digitais, e o Sistema AMADEUS (Ambiente e Metodologias Adaptáveis de DEsenvolvimento Unificado de Software) (/PRI 85/, /FAV 87/, /PRI 88/, /MEL 88/, /MEL 89/, /PRI 89/, /MAC 89a/), um ambiente integrado de desenvolvimento de software.

O capítulo é organizado da seguinte forma: cada um dos sistemas de projeto é apresentado em suas características gerais e, em seguida, é feita a modelagem dos dados através dos conceitos suportados pelo BD_PAC. Uma segunda modelagem é realizada utilizando-se os conceitos do DODM (veja capítulo 3), o modelo de dados suportado pelo DAMOKLES. A modelagem é realizada em duas etapas: (1) Os dados são descritos de forma diagramática possibilitando uma visão global e integrada das informações dos ambientes; (2) Logo após, e em um nível mais detalhado, um esquema descreve os dados apresentados na notação diagramática. Esta notação tem sido usada extensivamente nos capítulos 4 e 5 para ilustrar os conceitos suportados pelo modelo. Aqui, devido à importância da notação para a compreensão do capítulo, será realizada uma descrição de cada diagrama utilizado na modelagem e seu significado semântico.

A modelagem dos dados através dos dois modelos certamente fornecerá subsídio para avaliação da utilidade e poder de modelagem semântica de cada um dos modelos.

7.1 A Notação Diagramática

O modelo BD_PAC suporta tipos simples e tipos

construídos. Os simples são os *tipos de objetos* e os *tipos relacionamentos*; os construídos derivam dos tipos simples pela aplicação de um dos construtores de tipos: *agregação*, *generalização* ou *associação*. A notação diagramática deve diferenciar precisamente os tipos representados. Para isso, cada tipo tem uma representação diagramática própria.

A notação diagramática é idêntica em vários aspectos à notação do Modelo Entidade/Relacionamento /CHE 76/. Foram realizadas extensões nesta notação para a representação correta dos conceitos de abstração e versões, explicados nos capítulos 4 e 5, respectivamente. Estes capítulos definem a semântica exata de cada conceito suportado pelo BD_PAC.

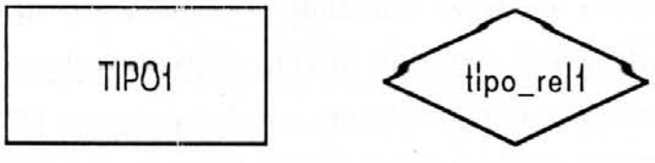

Figura 7.1: Tipo simples

- Os tipos simples são representados através de retângulos e losangos, retângulos para os tipos de objetos e losangos para os tipos relacionamentos, como mostra a figura 7.1. Os tipos relacionamentos têm a função de ligar tipos de objetos (simples ou construídos), portanto não têm existência própria. Esta característica é refletida na notação pelas ligações do tipo relacionamento com os tipos de objetos;

- Tipos construídos por generalização formam hierarquias de supertipos e subtipos. Esta hierarquia é representada pela ligação de tipos de objetos através de setas que partem do supertipo para os subtipos (figura 7.2);

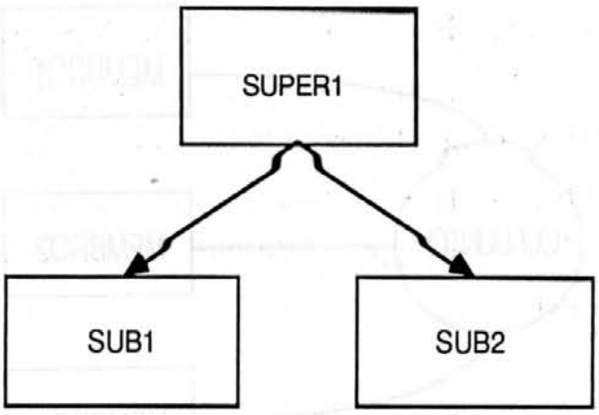

Figura 7.2: Generalização

- Agregações são a composição de tipos por outros tipos. Para representar esta semântica, um tipo construído por agregação é modelado por uma caixa (retângulo ou quadrado) que contém outros tipos, simples ou construídos (veja a figura 7.3);

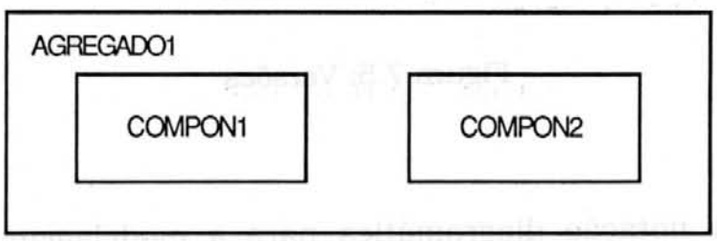

Figura 7.3: Agregação

- Conjuntos de objetos formados por associações são representados por círculos, de onde partem linhas indicando os tipos básicos para a formação do conjunto (figura 7.4);
- Com exceção dos relacionamentos, quaisquer dos tipos descritos acima podem ser versionados. Um tipo versionado é modelado como mostra a figura 7.5.

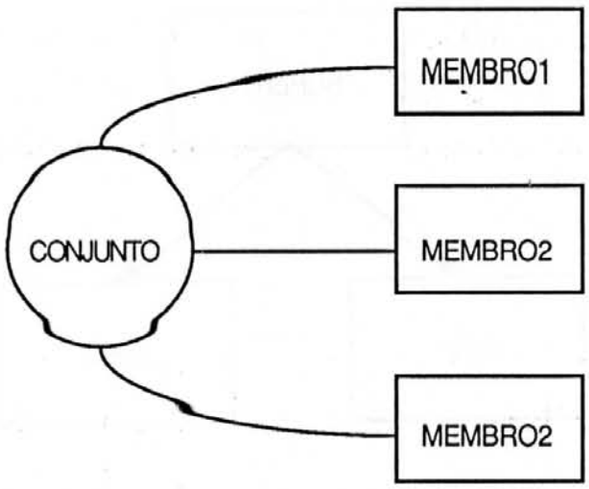

Figura 7.4: Associação

Figura 7.5: Versões

A notação diagramática para a modelagem de objetos através do DODM precisa ser um pouco diferente, embora também seja uma extensão da notação do modelo Entidade/Relacionamento. A semântica exata dos tipos representados está descrita no capítulo 3, onde foi discutido o Sistema DAMOKLES.

- Os tipos simples são representados de forma idêntica àquela do BD_PAC - retângulos para objetos e losangos para relacionamentos (figura 7.1);

- Objetos complexos ou estruturados têm notação semelhante a de um tipo construído por agregação no BD_PAC (ver figura 7.3);

- Versões no DAMOKLES podem ter estrutura totalmente diferente daquela do tipo versionado. Por esta razão, a representação não

pode ser tão simples como na notação do BD_PAC. Versões são então representadas como tipos de objetos, mas que necessariamente estão ligados ao tipo versionado por uma linha tracejada (observe a figura 7.6);

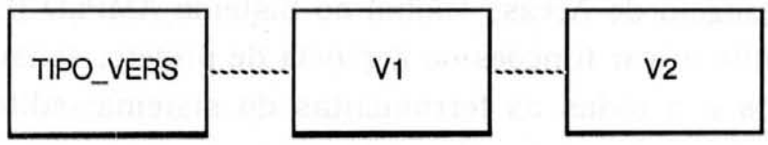

Figura 7.6: Versões no DAMOKLES

- O tipo *union* do DODM (ver capítulo 3) é representado por uma caixa dividida em partes (figura 7.7). Cada parte representa um tipo de objeto da união.

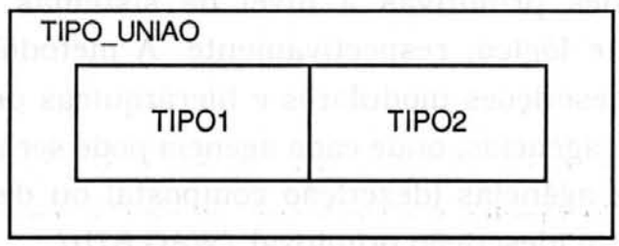

Figura 7.7: União no DAMOKLES

7.2 O Sistema AMPLO

O Sistema AMPLO é um ambiente integrado para o projeto de sistemas digitais. O ambiente apresenta as seguintes características principais /WAG 87b/:

- Um banco de dados unificado armazena todas as informações de projeto sobre as descrições dos módulos e modelos de simulação e ainda oferece a possibilidade de manutenção de *alternativas* e *versões* de módulos;

- A interação com o usuário é mantida por uma interface denominada Linguagem de Acesso Global ao Sistema AMPLO (LAGO), a qual permite ativar funções de gerência de projeto, consulta à base de dados e a todas as ferramentas do sistema: editores gráficos, compiladores e simuladores /LUZ 90/;

- Conceitos de gerência de projeto bem definidos são suportados inteiramente por todas as linguagens de descrição de hardware (HDLs) e por ferramentas do ambiente de projeto.

São em número de quatro as linguagens de descrição de hardware suportadas pelo ambiente: uma delas, a linguagem REDES /WAG 87a/, é dedicada a descrições compostas dos sistemas digitais; as outras três, as linguagens LAÇO, KAPA e NILO /WAG 87b/, são utilizadas para descrições primitivas a nível de sistemas, transferência de registradores e lógico, respectivamente. A metodologia de projeto é baseada em descrições modulares e hierárquicas de sistemas digitais como redes de agências, onde cada agência pode ser refinada como uma outra rede de agências (descrição composta) ou descrita em um dos níveis de projeto (descrição primitiva) /WAG 87b/.

Além das ferramentas de descrição de sistemas digitais, o ambiente ainda oferece uma família de simuladores para a validação do projeto dos sistemas. Os simuladores trabalham através de modelos de simulação. Um modelo de simulação é um conjunto de descrições primitivas de agências, obtidas pelo caminhamento hierárquico nas descrições compostas. Modelos de simulação podem ter um único nível ou podem ser multiníveis. No primeiro caso, todas as agências do modelo são descritas no mesmo nível de projeto. Nos modelos multiníveis, as agências podem ser descritas nos diferentes níveis de projeto - sistemas, transferência entre registradores e lógico.

Cada definição de agência, seja por descrição composta ou primitiva, deve ter um nível de descrição associado. O AMPLO permite a uma agência conter ocorrências de tipos de agências definidos em qualquer um dos níveis de projeto, desde que haja compatibilidade entre os sinais de interfaces das agências conectadas. Descrições primitivas de agências podem ser obtidas textualmente ou graficamente, havendo equivalência entre as duas formas /WAG 87a/.

7.2.1 Modelagem dos Dados - BD_PAC

Os dados tratados no Sistema AMPLO são armazenados em uma base de dados comum a todo o sistema /BEC 88/. As ferramentas de projeto - compiladores, editores gráficos, construtores de modelos e simuladores - e o usuário, através de uma linguagem de comandos global do ambiente que permite consulta e remoção de objetos, manipulam esta base de dados. O acesso à base de dados é realizado através de uma interface orientada a objeto, onde objetos são agências, alternativas, versões, ocorrências de alternativas e versões, e modelos de simulação /GOL 89a/ (observe a figura 7.8).

Uma agência pode ter inúmeras alternativas, cada uma delas constituindo uma interface distinta. Alguns atributos da agência, por exemplo, o nome, são herdados para as suas alternativas, enquanto que outros não, porque são características que descrevem a agência como um todo (ex. projetista, data_projeto). AGENCIA é, portanto, um agregado de AG_ALT numa relação de composição onde, para cada objeto do tipo AGENCIA, existe somente um objeto do tipo AG_ALT. AG_ALT é uma generalização de ALT_VER. Estes tipos possibitam a herança de propriedades das agências para as alternativas.

Uma alternativa pode estar associada a diversas versões, que apresentam a mesma interface. O tipo ALTERNATIVA é modelado como uma agregação de dois outros tipos, ALT_VER e SINAIS_I. ALT_VER é subtipo de AG_ALT, logo recebe seus atributos por herança;

ao mesmo tempo, também é supertipo de VER_GEN, passando para os níveis mais baixos da hierarquia os atributos herdados. Uma alternativa é compostas por vários sinais (de entrada, saída ou entrada/saída), que definem a interface. O tipo SINAIS_I é uma generalização do tipo SINAIS_V que, por sua vez, é uma generalização dos subtipos SIN_VS_P e SIN_VS_C, componentes das versões primitivas e compostas respectivamente (veja a figura 7.8).

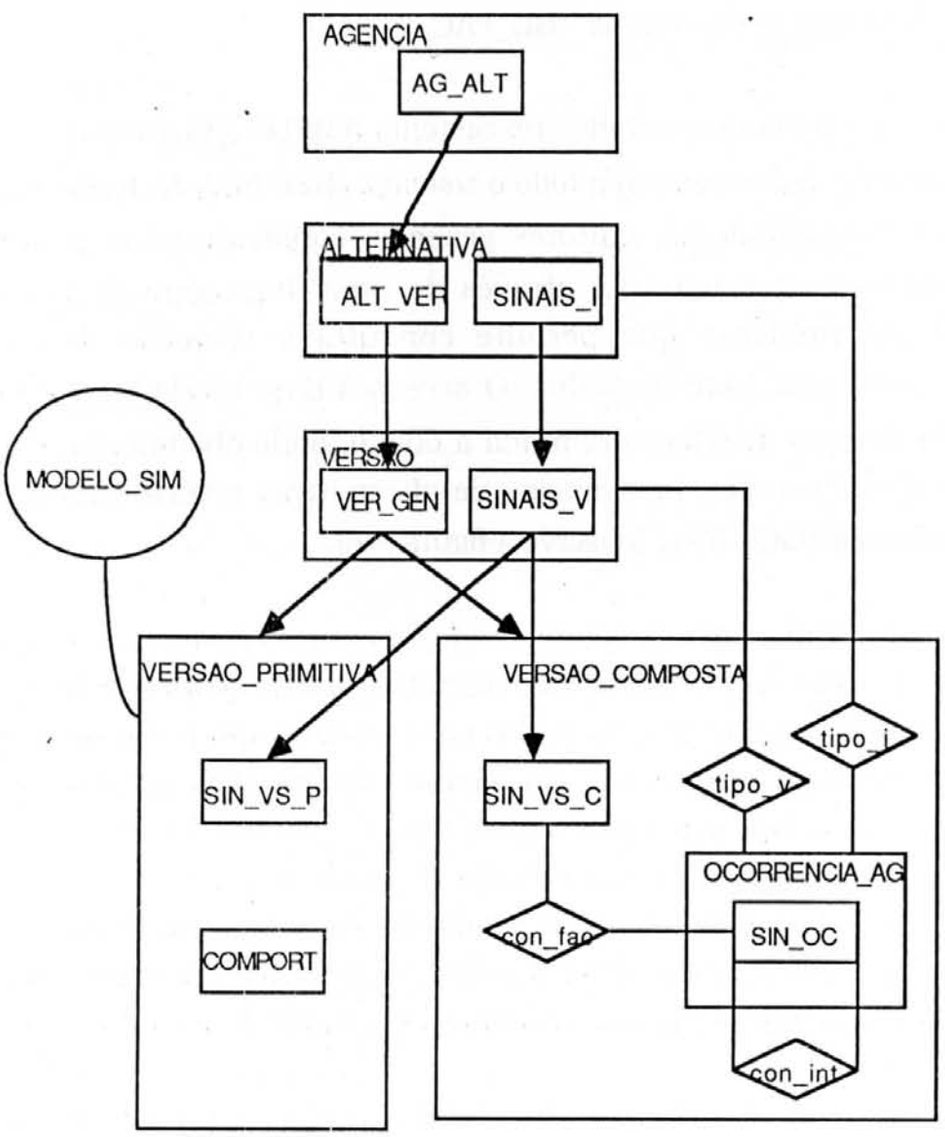

Figura 7.8: Modelagem do AMPLO com BD_PAC

Uma versão se especializa em primitiva ou composta, fato modelado pelo supertipo VER_GEN e subtipos VERSAO_PRIMITIVA e VERSAO_COMPOSTA. Os objetos destes dois subtipos têm em comum os sinais de interface e algumas propriedades herdadas através da hierarquia, no entanto apresentam estruturas totalmente distintas. Uma versão primitiva descreve uma agência em um dos três níveis de projeto - lógico, transferência entre registradores e sistemas. Esta descrição, que caracteriza o comportamento da versão da agência, é armazenada em objetos do tipo COMPORT, componente do tipo VERSAO_PRIMITIVA.

Uma VERSAO_COMPOSTA consiste de uma rede de agências. Cada agência componente é uma ocorrência de alternativa ou versão de agência /GOL 89b/. Na modelagem, o tipo OCORRENCIA_AG tem dois relacionamentos, um com o tipo ALTERNATIVA indicando que a ocorrência de agência corresponde a uma alternativa, o outro com o tipo VERSAO indicando que a ocorrência de agência corresponde a uma versão. Quando uma ocorrência de agência corresponde a uma alternativa, então somente a interface da agência componente é definida na descrição composta. Caso as ocorrências de agências correspondam a versões, a descrição composta está estaticamente definida, isto é, está completamente declarada e pronta para ser utilizada em modelos de simulação. O uso de ocorrências de agências como alternativas permite a definição de um *template* /BAT 85/, onde o projetista pode mudar as versões selecionadas dinamicamente, construindo uma variedade de modelos de simulação do sistema digital a ser analisado /GOL 89a/.

Numa descrição composta, os sinais das interfaces das agências definem as conexões entre as agências componentes e a agência composta, que se está descrevendo. Os sinais de interface da versão composta são herdados pela hierarquia de generalização iniciada na interface pelo tipo de objeto SINAIS_I. Cada ocorrência de agência também tem o seu conjunto de sinais de interface armazenados nos objetos do tipo SIN_OC. Eventualmente, uma versão composta pode ter sinais internos, representados pelo auto-relacionamento CON_INT. As

conexões entre sinais de interface, seja da versão composta ou de ocorrências de agências, são armazenadas no relacionamento CON_FAC.

O tipo de objeto VERSAO é declarado como um tipo versionado, portanto, por herança, também são versionados os subtipos VERSAO_PRIMITIVA e VERSAO_COMPOSTA.

O esquema BD_PAC que descreve os tipos da figura 7.8 é mostrado no Anexo 4.

7.2.2 Modelagem dos Dados - DAMOKLES

A modelagem dos objetos do Sistema AMPLO através do modelo de dados do DAMOKLES é mostrada na figura 7.9. A modelagem poderia ser feita de duas formas diferentes: (1) Os tipos de objetos AGENCIA, ALTERNATIVA e VERSAO seriam modelados como tipos do modelo de dados e ligados através de relacionamentos, deixando para a aplicação o controle das relações entre as agências, suas alternativas e versões; (2) O tipo VERSAO seria modelado como versão do tipo ALTERNATIVA e este modelado como versão do tipo AGENCIA. A segunda opção foi a escolhida porque representa melhor a dependência hierárquica das alternativas para as agências e das versões para as alternativas.

Os objetos do tipo ALTERNATIVA são armazenados como versões do tipo AGENCIA. Para cada agência, neste contexto tratada como um objeto genérico, pode haver diversas versões que têm dependência direta da existência do objeto genérico. Não ocorre herança de propriedades entre os tipos AGENCIA e suas versões. Atributos que, no AMPLO, são herdados das agências para as alternativas, precisam ser explicitamente declarados. A aplicação também deve fazer a cópia dos valores dos atributos herdados e manter a consistência entre os valores.

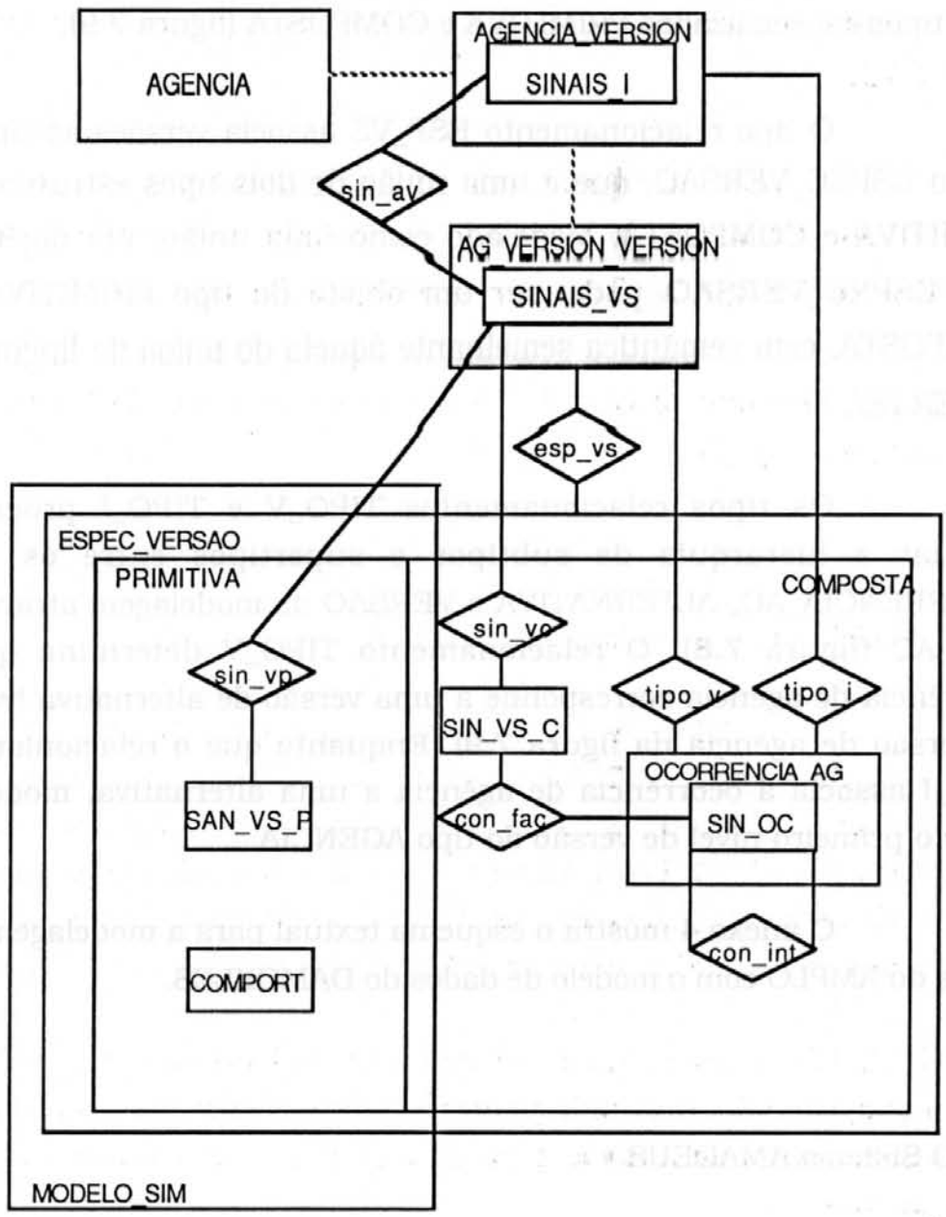

Figura 7.9: Modelagem do AMPLO com DAMOKLES

As versões das alternativas são armazenadas no segundo nível de versão do tipo AGENCIA (figura 7.9). Os sinais da interface, que são herdados das alternativas para as versões na modelagem do BD_PAC, têm um relacionamento de equivalência através do tipo de relacionamento SIN_AV. Este relacionamento possibilita o controle da correspondência entre os sinais de interface das alternativas e versões.

Fato semelhante ocorre entre os sinais de interface das versões e dos seus tipos especializados PRIMITIVA e COMPOSTA (figura 7.9).

O tipo relacionamento ESP_VS associa versões ao tipo de objeto ESPEC_VERSAO, que é uma união de dois tipos estruturados, PRIMITIVA e COMPOSTA. Modelado como uma união, um objeto do tipo ESPEC_VERSAO pode *ser* um objeto do tipo PRIMITIVA *ou* COMPOSTA, com semântica semelhante àquela do *union* da linguagem C /KER 78/.

Os tipos relacionamentos TIPO_V e TIPO_I procuram simular a hierarquia de subtipos e supertipos entre os tipos OCORRENCIA_AG, ALTERNATIVA e VERSAO da modelagem através do BD_PAC (figura 7.8). O relacionamento TIPO_V determina que a ocorrência de agência corresponde a uma versão de alternativa (versão de versão de agência na figura 7.9). Enquanto que o relacionamento TIPO_I associa a ocorrência de agência a uma alternativa, modelada como o primeiro nível de versão do tipo AGENCIA.

O anexo 4 mostra o esquema textual para a modelagem dos dados do AMPLO com o modelo de dados do DAMOKLES.

7.3 O Sistema AMADEUS

O AMADEUS, um ambiente de desenvolvimento de software em construção na UFRGS, é composto de várias ferramentas integradas que compartilham uma base de dados comum. As ferramentas interagem entre si através da base de dados que armazena as diversas representações e versões das especificações e programas produzidos no ambiente.

O Editor Dirigido por Sintaxe (EDS) é um *editor de textos programável*. A sintaxe e semântica da linguagem a ser processada pelo EDS é descrita em LDE (Linguagem de Especificação). A LDE permite a

especificação sintática e semântica de uma linguagem, através de uma *gramática de atributos* estendida por *variáveis globais* e *ações semânticas*. O texto em edição é armazenado internamente na forma de uma *Árvore Sintática Abstrata*. Um módulo do sistema cria uma descrição interna (tabelas do reconhecedor) da linguagem a partir da especificação feita em LDE. O EDS e as demais ferramentas que ficam acopladas a ele (depurador, formatador, "browser", compilador) utilizam constantemente as tabelas do reconhecedor na edição, submissão e formatação do texto.

O Editor Diagramático Generalizado (EDG) é uma ferramenta que tem a capacidade de gerar Editores Diagramáticos Especializados. O EDG é dividido em dois módulos principais: O Meta Editor Diagramático (MED) e o Editor Diagramático Específico (EDE). Através do MED são descritos os tipos de nodos, arcos e regras de ligação de uma técnica particular. Utilizando o EDE o usuário da técnica interage com o EDG criando e mantendo diagramas. O EDE suporta somente a edição de diagramas de técnicas especificadas no MED. A interface do EDE é uniforme para todas as técnicas suportadas divergindo somente quanto aos tipos de diagramas que podem ser editados.

O MED é utilizado para a definição dos objetos primitivos da técnica diagramática. Os objetos primitivos são então inseridos num contexto de uma gramática de atributos, especificada em LDE, onde eles são tratados como os elementos léxicos da gramática. A fim de ligar os níveis léxico e sintático é criado um cabeçalho para todo elemento gráfico /PRI 89/.

Todas as ferramentas do AMADEUS geram documentos que, ou fazem parte diretamente do software em desenvolvimento ou são saídas de fases intermediárias do processo de produção do software. Os documentos podem ser especificações gráficas e textuais, diagramas de projeto conceitual do sistema, programas, manuais, massas de testes.

7.3.1 Modelagem dos Dados - BD_PAC

O tipo de objeto GRAMATICA é mostrado na figura 7.10. Uma gramática definida em LDE é formada por objetos dos seguintes tipos: ATRIBUTO, ACAO, LEXICO, PRODUCAO e alguns relacionamentos associando esses objetos.

A gramática de atributos permite a especificação de atributos associados às produções e a passagem desses atributos entre as produções. Cada produção pode ter, associadas ao seu cabeçalho, variáveis de importação (atributos herdados), variáveis de exportação (atributos sintetizados) e variáveis de trabalho. Os atributos herdados são recebidos pela produção no momento em que esta é ativada pelo reconhecedor da gramática. Já os valores dos atributos sintetizados retornam quando a produção acaba de ser reconhecida. Variáveis de trabalho são variáveis auxiliares utilizadas durante o reconhecimento da produção. Os atributos são responsáveis por grande parte da verificação semântica estática do texto submetido /FAV 87/. Cada produção pode ou não ter atributos declarados em seu cabeçalho (relacionamento P_ATR da figura 7.10).

Ações semânticas são procedimentos ativáveis pelo reconhecedor. As ações são associadas às produções da gramática através do relacionamento P_AC (figura 7.10). Durante a submissão de texto, no momento em que uma produção é reconhecida, todas as ações associadas à produção são ativadas. As ações são utilizadas para manipulação de tabelas de símbolos, formatação de texto, atribuição de variáveis, avaliação de expressões. Elas são responsáveis também pela verificação da semântica estática do texto.

Os objetos léxicos são divididos em dois grupos: os léxicos textuais e os léxicos diagramáticos (figura 7.10). Os léxicos textuais são constituídos pelo conjunto básico de caracteres e cadeias de caracteres, formando as unidades léxicas de uma linguagem. Os léxicos diagramáticos são as unidades léxicas de uma técnica diagramática.

Eles contêm todas as características de desenho e formatação dos diagramas, que são definidos através do Meta Editor Diagramático.

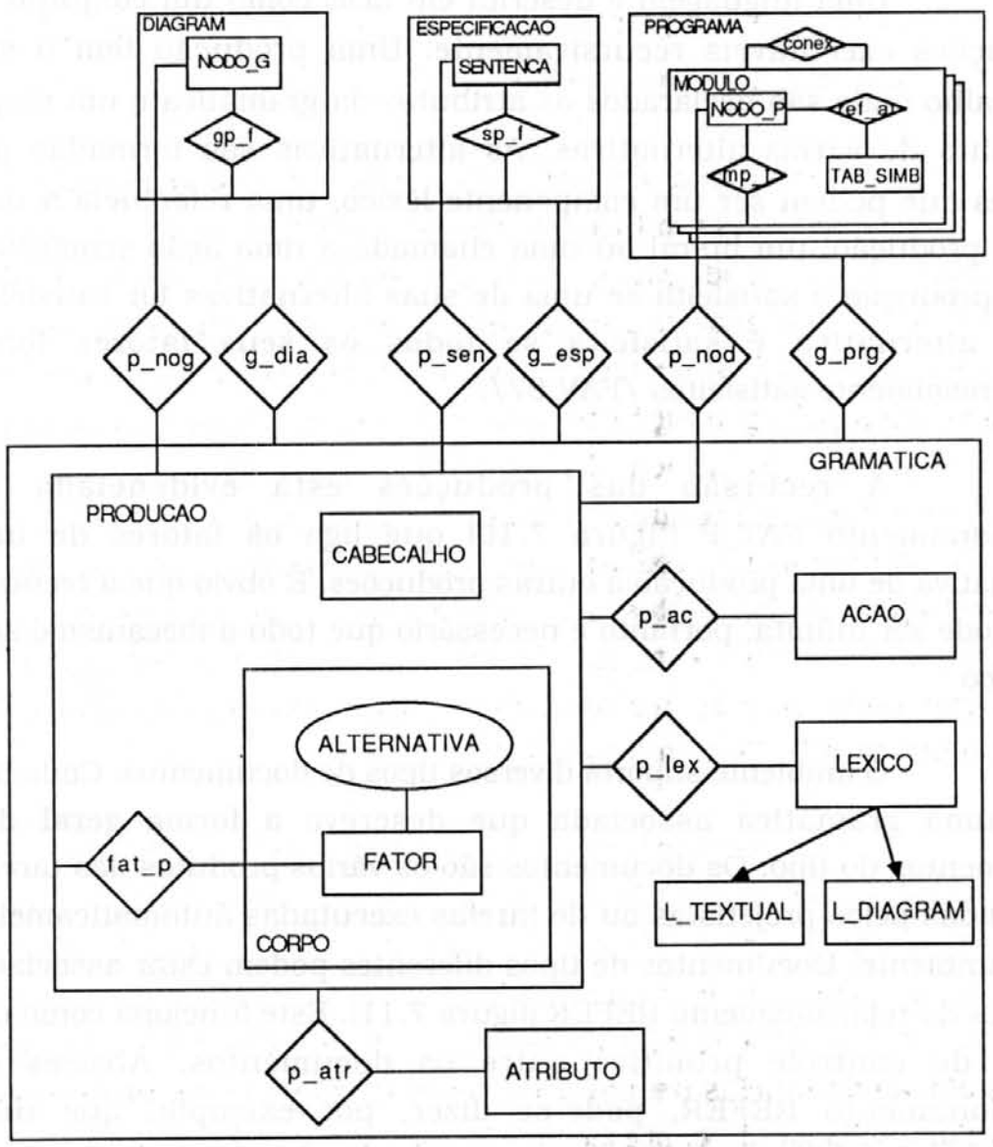

Figura 7.10: A gramática e seus relacionamentos

Os léxicos formam os símbolos terminais da sintaxe da linguagem ou notação diagramática. Se a gramática é aplicada a

documentos textuais, os léxicos são os textuais. Se a gramática é aplicada a documentos diagramáticos, então os léxicos são diagramáticos.

Uma linguagem é descrita em LDE como um conjunto de produções executáveis recursivamente. Uma produção tem o seu cabeçalho onde são declarados os atributos da gramática e um corpo, conjunto de várias alternativas. As alternativas são formadas por fatores que podem ser um componente léxico, uma referência a uma outra produção, um literal ou uma chamada a uma ação semântica. Uma produção é satisfeita se uma de suas alternativas for satisfeita. Uma alternativa é satisfeita se todos os seus fatores forem seqüencialmente satisfeitos /FAV 87/.

A recursão das produções está evidenciada no relacionamento FAT_P (figura 7.10) que liga os fatores de uma alternativa de uma produção a outras produções. É óbvio que a recursão não pode ser infinita, portanto é necessário que todo o mecanismo seja *acíclico*.

O ambiente suporta diversos tipos de documentos. Cada tipo tem uma gramática associada que descreve a forma geral dos documentos do tipo. Os documentos são os vários produtos das tarefas realizadas pelos projetistas ou de tarefas executadas automaticamente pelo ambiente. Documentos de tipos diferentes podem estar associados através do relacionamento REFER (figura 7.11). Este funciona como um grafo de controle primitivo entre os documentos. Através do relacionamento REFER, pode-se dizer, por exemplo, que uma especificação algébrica se refere a um programa do sistema, ou que um programa implementa determinado processo de um Diagrama de Fluxo de Dados. Os documentos podem ser textuais ou diagramáticos.

Os documentos textuais são aqueles que utilizam uma linguagem escrita como forma de descrever os documentos do projeto. Eles incluem especificações escritas tais como VDM, português estruturado, especificações algébricas, manuais, programas, dados de

teste (figura 7.11).

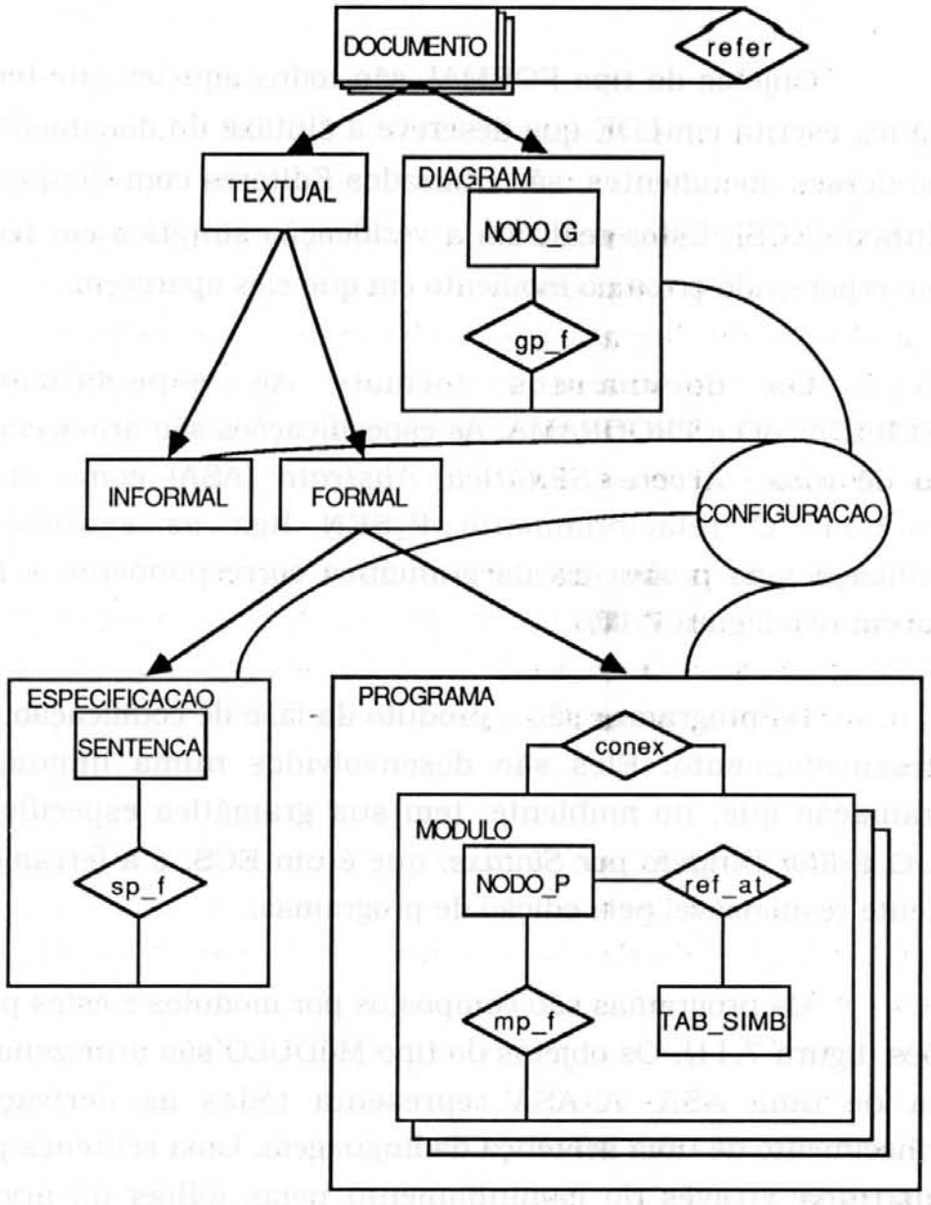

Figura 7.11: Os documentos gerados pelo AMADEUS

Os documentos textuais são ainda divididos em INFORMAL e FORMAL. A diferença básica entre eles é a ausência de descrição gramatical nos documentos informais.

Os documentos informais não possuem gramática livre de contexto descrevendo suas propriedades sintáticas. São editados livremente e são armazenados sob a forma de um campo longo.

Objetos do tipo FORMAL são todos aqueles que tem uma gramática escrita em LDE que descreve a sintaxe do documento. Para edição desses documentos, são utilizados Editores com Conhecimento de Sintaxe (ECS). Estes realizam a verificação sintática em tempo de edição, reportando erros no momento em que eles aparecem.

Os documentos formais se especializam em ESPECIFICACAO e PROGRAMA. As especificações são armazenadas na forma de uma *Árvore Sintática Abstrata* (ASA) como mostra a figura 7.11. O relacionamento P_SEN liga as sentenças das especificações às produções da gramática correspondente à notação formal em uso (figura 7.10).

Os programas são o produto da fase de codificação do ciclo de desenvolvimento. Eles são desenvolvidos numa linguagem de programação que, no ambiente, tem sua gramática especificada em LDE. O *Editor Dirigido por Sintaxe*, que é um ECS, é a ferramenta do ambiente responsável pela edição de programas.

Os programas são compostos por módulos e estes possuem versões (figura 7.11). Os objetos do tipo MóDULO são armazenados na forma de uma ASA. A ASA representa todas as derivações do reconhecimento de uma sentença da linguagem. Uma sentença pode ser reconstruída através do caminhamento pelas folhas da árvore. Os nodos da árvore são ligados às produções da gramática através do relacionamento P_NOD. O relacionamento CONEX armazena a conexão entre os módulos que formam um programa (figura 7.11).

Os diagramas são documentos compostos de objetos geométricos. Uma técnica diagramática também possui sintaxe e restrições de ligação entre os vários objetos da técnica. Essas características são especificadas em LDE, onde um mecanismo de

gramática de atributos é utilizado. O Meta Editor Diagramático (MED) é utilizado para descrever os objetos diagramáticos da técnica a ser utilizada. O MED possui figuras geométricas que são utilizados como molde para a descrição dos objetos da técnica. O Editor Diagramático Específico utiliza o mecanismo de gramática de atributos para garantir a correta construção de diagramas da técnica. Várias técnicas diagramáticas podem ser especificadas, cada uma delas com uma gramática específica associada.

Os diagramas são armazenados internamente através de uma estrutura de árvore - Árvore Sintática Abstrata (figura 7.11). Da mesma forma que os programas, os nodos da ASA são associados às produções da gramática através de um relacionamento chamado P_NOG. Os léxicos da gramática são os objetos diagramáticos definidos no Meta Editor Diagramático.

Uma configuração é um conjunto de objetos dos tipos DIAGRAM, INFORMAL, ESPECIFICACAO e PROGRAMA (figura 7.11). Em geral o pessoal de desenvolvimento trabalha em determinadas partes do projeto e não necessita de todos os objetos de projeto disponíveis numa sessão de trabalho. Uma configuração determina um bloco de documentos que são aglomerados explicitamente pelos projetistas.

7.3.2 Modelagem dos Dados - DAMOKLES

Os objetos do AMADEUS modelados no BD_PAC e mostrados na figura 7.10 têm estrutura semelhante aos da modelagem através do modelo do DAMOKLES. Os objetos estruturados ali representados são mapeados diretamente para hierarquias de agregações, enquanto que os relacionamentos são mapeados para relacionamentos do BD_PAC. Exceção ocorre com o tipo de objeto LEXICO, porque é um supertipo de TEXTUAL e DIAGRAM. Ele é modelado no DAMOKLES como um objeto estruturado contendo os tipos

TEXTUAIS e DIAGRAM (figura 7.12).

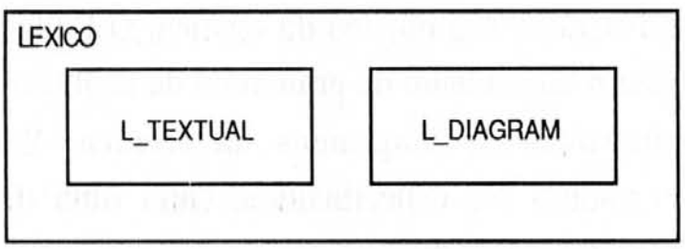

Figura 7.12: Léxicos no DAMOKLES

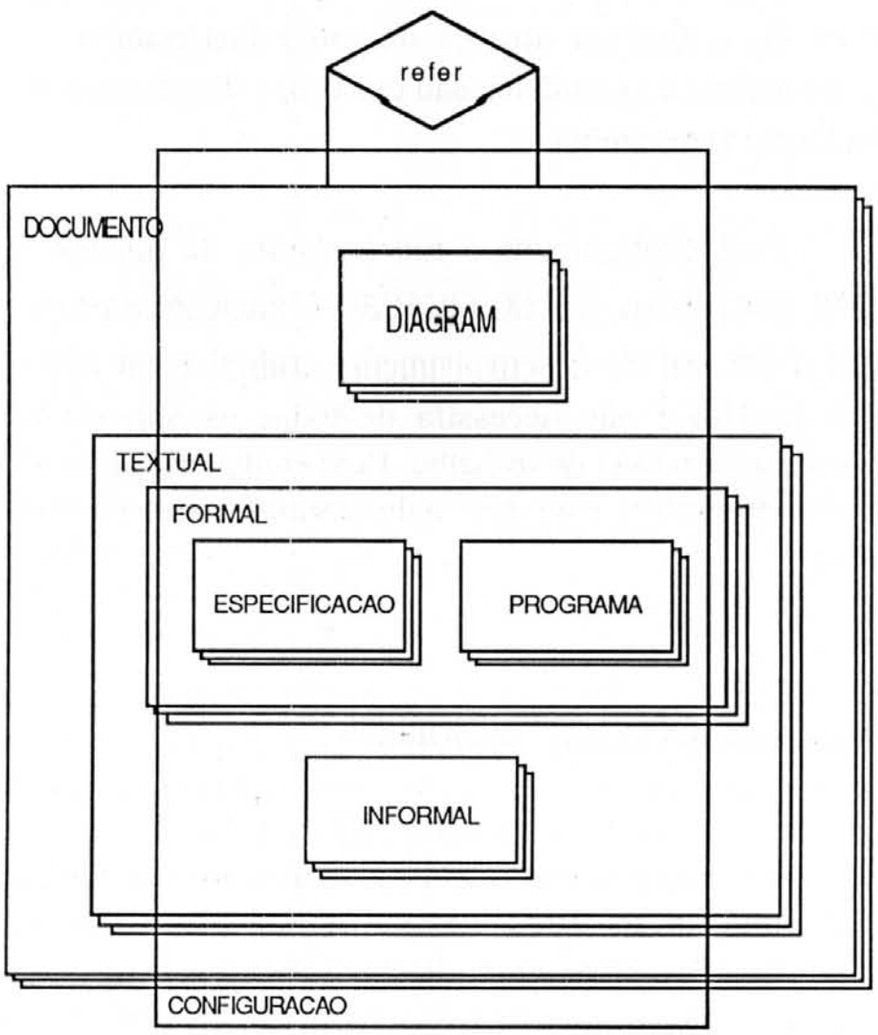

Figura 7.13: Hierarquia de documentos no DAMOKLES

A modelagem da estruturação dos documentos gerados pelo AMADEUS foi realizada através de uma hierarquia de generalização. Esta hierarquia foi transportada para o modelo do DAMOKLES como uma hierarquia de objetos estruturados (figura 7.13).

O tipo CONFIGURACAO, modelado como um conjunto no BD_PAC, passou a ser um tipo estruturado, agregando os tipos DIAGRAM, INFORMAL, ESPECIFICACAO e PROGRAMA.

O esquema DAMOKLES que descreve os dados do AMADEUS é mostrado no Anexo 4.

7.4 Considerações Finais

Os dados modelados através do BD_PAC têm conteúdo semântico bem definido devido ao uso de construtores de tipos específicos para cada abstração que se quer representar. O mesmo não acontece com os dados modelados utilizando-se do modelo de dados do DAMOKLES. O Sistema DAMOKLES suporta somente dois construtores de tipo - tipo estruturado e união - e dois tipos simples - tipo de objeto e tipo relacionamento. Com isto, tipos construídos pelos construtores de tipos do BD_PAC precisam ser mapeados para os conceitos do modelo do DAMOKLES mais simples, resultando na perda de semântica dos dados representados no banco de dados. As restrições de integridade inerentes a cada tipo construído pelo BD_PAC passa a ser realizada pela aplicação, a fim de manter correta a representação dos dados.

No mapeamento dos conceitos do BD_PAC para os conceitos do DAMOKLES, muitas das facilidades fornecidas pelos construtores foram perdidas:

- Todos os documentos do AMADEUS, desde o tipo DOCUMENTO até os tipos ESPECIFICACAO e PROGRAMA, têm versões. Na modelagem com o BD_PAC, declara-se o tipo DOCUMENTO como

versionado e todos os seus subtipos herdam esta propriedade. Na modelagem com o DAMOKLES, todos os tipos da hierarquia de documentos têm que ser explicitamente declarados como versionados. Mesmo assim, o Sistema DAMOKLES não mantém qualquer restrição entre as várias associações de versões dos documentos;

- As propriedades declaradas no tipo DOCUMENTO do AMADEUS e na hierarquia de sinais do AMPLO não são herdadas para os subtipos, forçando a redeclaração e atualização dos valores pela aplicação;

- A correspondência semântica de que um objeto de um subtipo *é* um objeto do supertipo (objeto especializado) não é inerente a hierarquia de objetos estruturados do DAMOKLES. Esta característica precisa ser mantida pelos editores do sistema;

- Alguns relacionamentos entre tipos e supertipos ou entre conjuntos e membros tiveram que ser transformados em relacionamentos genéricos, com a conseqüente perda de controle semântico pelo sistema de banco de dados;

- Uma produção do AMADEUS tem somente um cabeçalho e corpo, característica modelada no BD_PAC, mas impossível de modelar no DAMOKLES;

- A especialização do tipo VERSAO em PRIMITIVA e COMPOSTA foi modelada no DAMOKLES como um relacionamento e uma união. O relacionamento existe de fato na aplicação - representa a especialização - mas o tipo união ESPEC_VERSAO foi artificialmente inserido para representar o *ou exclusivo* entre os tipos PRIMITIVA e COMPOSTA.

Os esquemas gerados pela modelagem através do BD_PAC são menores e mais compreensíveis que o equivalente gerado pela modelagem com o DAMOKLES. Verificações de restrições de integridade são realizadas automaticamente pelo BD_PAC, liberando as ferramentas dos ambientes de realizar estas tarefas.

8 CONCLUSÃO

A integração das ferramentas é uma propriedade necessária nos modernos ambientes de projeto. O compartilhamento dos dados através de uma interface comum e padronizada é de vital importância para a integração das ferramentas. Por outro lado, sistemas de banco de dados fornecem esta facilidade além de muitas outras /DAT 86/ na gerência de informação de sistemas. No entanto, os dados gerados em aplicações de projeto apresentam um conjunto de características que não são satisfatoriamente modeladas nos sistemas de bancos de dados disponíveis comercialmente /SID 80/. Este trabalho propôs, então, um modelo de dados para ambientes de projeto. O modelo proposto, chamado de BD_PAC, foi projetado para satisfazer o tanto quanto possível às necessidades de estruturação de dados dos modernos ambientes de projeto de software ou de hardware.

O projeto do BD_PAC poderia ser realizado de duas formas diferentes:

- A partir de análise de requisitos realizada em ambiente de projeto, definir as propriedades do novo modelo e implementar um sistema de gerência de banco de dados capaz de suportá-lo;
- Realizar extensões no modelo de dados de um sistema de banco de dados disponível, com o objetivo de satisfazer o máximo possível as necessidades de dados dos ambientes de projeto.

A segunda opção foi a escolhida. Diversos fatores guiaram esta escolha:

Tempo: o desenvolvimento de todo um sistema de banco de dados que suporta o modelo demandaria muito tempo de implementação. No entanto, o objetivo era expor as novas idéias tão rápido quanto possível a ambientes de projeto;

Pessoal: os recursos de pessoal para a implementação do sistema eram escassos;

Disponibilidade do DAMOKLES: o Sistema DAMOKLES, um SGBD voltado para ambientes de desenvolvimento de software, poderia ser utilizado como base para a construção do BD_PAC.

O BD_PAC foi então projetado como uma extensão do modelo de dados do Sistema DAMOKLES, chamado DODM. O BD_PAC estende o DODM (explicado em detalhe no capítulo), nas seguintes características: (1) suporte para três construtores semânticos de tipos, chamados de conceitos de abstração /MAT 88/; (2) novas propriedades para o plano de versões.

O construtor de tipo agregação permite a definição de objetos complexos, formando uma hierarquia de agregados e componentes. Os componentes podem ser compartilhados por diversos agregados. O modelo suporta a declaração de cardinalidade dos objetos componentes.

O construtor de tipo generalização suporta a criação de hierarquias de supertipos e subtipos, onde as propriedades dos supertipos são herdadas para os subtipos.

O construtor de associação possibilita a formação de conjuntos, onde os elementos são objetos de outros tipos. Atributos do conjunto podem ser derivados diretamente dos valores dos atributos dos objetos membros, através da aplicação de uma função de cálculo previamente definida.

Os construtores de tipos podem-se compor indistintamente, permitindo a formação de hierarquias mistas em conceitos. Por exemplo, um agregado pode ter componentes que são supertipos de conjuntos. No entanto, devido a algumas limitações do modelo básico - o modelo do DAMOKLES - um tipo não pode ser ao mesmo tempo mais do que um conceito de abstração (por exemplo, ser ao mesmo tempo um supertipo e um agregado, como é possível no Sistema KRISYS /MAT 89/). Isto refletiu diretamente nos estudos de caso descritos no capítulo 7, onde houve a necessidade de se criar tipos de objetos que na

realidade não existiam na aplicação. Cada um desses tipos "artificiais" representavam relacionamentos semânticos diferentes entre dois tipos.

O plano de versões no BD_PAC reflete as qualidades semânticas dos tipos versionados. Esta característica tem como resultado um plano de versões fácil de ser acompanhado por parte do usuário, ao mesmo tempo que restringe o escopo das associações semânticas entre versões de objetos.

Na extensão do DODM, foi especificada uma nova linguagem de definição de dados e implementado um compilador capaz de reconhecer todas as construções gramaticais definidas. O BD_PAC mantém ainda um dicionário de dados auxiliar onde guarda informações semânticas sobre os tipos. Os operadores da linguagem de manipulação de dados foram alterados e oferecidas novas operações.

O sistema de banco de dados que suporta o BD_PAC herdou ainda as seguintes características do DAMOKLES:

- Facilidades de construção de domínios de atributos estruturados, inclusive campos longos;
- Suporte ao conceito de transação de projeto, com mecanismos de controle de concorrência e reconstrução em caso de falha;
- Manutenção de bancos de dados de grupos e individuais, permitindo áreas de dados privativas e áreas de dados públicas a projetistas de mesmo grupo.

Todos os tipos, simples ou construídos, do BD_PAC são traduzidos para os tipos, simples ou estruturados, do DAMOKLES. Desta forma, aproveitam-se os mecanismos de otimização ("cluster" lógico) na implementação de objetos.

Ambientes de projeto em desenvolvimento na UFRGS para as áreas de software e hardware são aplicações que podem utilizar as facilidades oferecidas pelo BD_PAC. Certamente esta utilização

resultará na identificação de novos requerimentos ou mesmo possíveis falhas na implementação do protótipo, o que garante evolução contínua para o modelo proposto.

8.1 Sugestões para Trabalhos Futuros

O acesso aos dados do BD_PAC é realizado somente através de um conjunto de operadores que têm interface com a Linguagem C /KER 78/. Uma linguagem de consulta de alto nível permitiria acesso direto aos dados sem a necessidade de se utilizar um programa escrito em Linguagem C. Muitas ferramentas dos modernos ambientes de projeto são gráficas, e o mesmo padrão poderia ser seguido para os sistemas de banco de dados. A implementação de uma linguagem diagramática para a definição de dados para o BD_PAC nos moldes de /REG 90/, bem como uma linguagem de consulta igualmente gráfica são excelentes tópicos de investigação.

O sistema não tem facilidades para a evolução de esquemas. Caso haja a necessidade de mudar a estrutura dos objetos, a aplicação precisa realizar todos os procedimentos para a transformação estrutural dos objetos e compatibilizar as ferramentas do ambiente à nova estrutura.

Uma evolução natural do modelo é a aproximação cada vez maior do paradigma de objeto. Para isto, é necessário um estudo no sentido de estender o BD_PAC para suportar os conceitos de métodos para os tipos e transferência de mensagens entre objetos.

8.2 Análise do Trabalho

O modelo proposto e implementado satisfaz os objetivos traçados. O BD_PAC suporta os principais conceitos de abstração encontrados na literatura especializada, sendo estes, ainda, reconhecidamente necessários à modelagem de dados de ambientes de

apoio a projeto. A expressividade semântica do modelo foi conseguida sem qualquer prejuízo às características específicas do Sistema DAMOKLES para ambientes de projeto - transações de projeto e múltiplos bancos de dados.

O novo plano de versões reflete a preocupação de trazer maior parcela do significado semântico da aplicação para o sistema de banco de dados.Isto torna mais simplificado o trabalho de ferramentas de verificação e simulação, já que uma parcela significativa da interconexão entre versões de objetos é realizada pelo sistema de banco de dados.

Minimizar alterações no código do Sistema DAMOKLES foi uma premissa básica na implementação do protótipo. O DAMOKLES tem código fonte bem documentado e relativamente fácil de entender, embora escrito em linguagem C, no entanto é muito extenso (9Mb de código), o que impossibilitou a análise de todo o código. Este fato gerou insegurança quanto a grandes modificações internas no código do DAMOKLES. Desde o início houve a preocupação de compatibilizar o BD_PAC com possíveis novas versões do DAMOKLES. Mais um bom motivo para que a implementação do BD_PAC fosse o máximo possível independente do código do DAMOKLES.

Uma parcela significativa do tempo despendido para a realização deste trabalho e que não está esplícita no volume foi o esforço realizado na compreensão necessária do funcionamento interno do DAMOKLES, a fim de possibilitar a implementação da extensão. A documentação disponível tratava apenas das facilidades do sistema no suporte ao projeto, sem entrar em detalhes sobre implementação. O código em liguagem C foi a única fonte de recurso para o estudo do funcionamento interno do DAMOKLES.

A modelagem dos dados dos ambientes AMPLO e AMADEUS realizada no estudo de caso delineou um quadro comparativo entre o BD_PAC e o DAMOKLES, ao mesmo tempo que demonstrou problemas na ortogonalidade dos construtores de tipo do

BD_PAC. Expressividade e concisão são características presentes nos esquemas BD_PAC em relação aos esquemas DAMOKLES. A impossibilidade de definir um tipo através de dois ou mais construtores diferentes é uma restrição forte no BD_PAC. Na modelagem do AMPLO isto fica claro pela uso de artifícios para suportar satisfatoriamente os dados do ambiente. A restrição no BD_PAC veio como herança do DAMOKLES, já que este não permite redefinição de tipo no esquema e os tipos do BD_PAC são traduzidos internamente para tipos do DAMOKLES numa relação unívoca. Esta forma de implementação maximizou o reuso e minimizou as alterações no código fonte do DAMOKLES.

ANEXO 1

SINTAXE DA LINGUAGEM DE DEFINIÇÃO DOS DADOS

A NOTAÇÃO

A sintaxe da DDL é descrita por uma gramática livre de contexto, através de uma notação BNF estendida. A gramática consiste de uma seqüência de regras. O lado esquerdo de cada regra - cabeçalho da regra - é separado do lado direito correspondente - corpo da regra - pelo sinal "::=". Cada regra é terminada por um ponto (".") . Os símbolos terminais da gramática são identificadores ou aparecem entre aspas simples ("'").

Regras diferentes com um mesmo cabeçalho são agrupadas em uma única regra com os corpos alternativos separados por "|". Por exemplo:

```
declaration ::= constant_decl    |
                value_set_decl   |
                object_kind_decl |
                relship_type_decl .
```

Partes opcionais de um corpo vêem entre colchetes.

```
factor ::= [ '-' ] constant .
```

A repetição de um elemento *uma* ou mais vezes é expressa por "(" e ")+", no entanto *zero* ou mais vezes por "(" e ")*".

```
constant_decl ::= CONST ( const_decl )+ .
```

Grupos de elementos aparecem entre parênteses. Listas são repetições onde os elementos são

separados por um delimitador. As listas são caracterizadas pelo símbolo "//" seguido do delimitador e consistem de pelo menos um elemento.

```
value_set ::= STRUCT ( attribute // ; ) END .
```

ELEMENTOS LÉXICOS

Identificadores

```
id      ::=
        letter ( letter | digit )* .

name    ::=
        letter ( letter | digit )* .

letter      ::=
        'a'..'z' | 'A'..'Z' | '_' .

digit ::=
        '0'..'9' .
```

Números

```
int_constant    ::=
        decimal_constant  |
        hex_constant              .

decimal_constant     ::=
        ( digit )+ .

hex_constant    ::=
        'Ox' ( hex_digit )+ .

hex_digit       ::=
        digit                   |
        'a'..'f'                |
        'A'..'F'                .
```

Cadeias de caracteres

```
string_constant     ::=
      '"' ( char )* '"' .

char_constant  ::=
      ''' char ''' .
```

Datas

```
date_constant  ::=
      '@' [ day '.' ] month '.' year
      [ '.' hour ':' minute ] '@' .

day               ::=
      decimal_constant  .

month             ::=
      decimal_constant  .

year              ::=
      decimal_constant  .

hour              ::=
      decimal_constant  .

minute                ::=
      decimal_constant  .
```

ESQUEMA

```
schema          ::=
      SCHEMA  schema_id
           ( declaration )+
      END  schema_name  .

declaration     ::=
      constant_decl          |
      value_set_decl         |
      object_kind_decl   |
      relship_type_decl .

schema_id  ::=
      id .

schema_name     ::=
      name  .
```

DECLARAÇÃO DE CONSTANTES

```
constant_decl  ::=
     CONST ( c_decl )+ .

c_decl              ::=
     constant_id '=' constant_expr ';' .

constant_expr  ::=
     term          |
     constant_expr ( '+' | '-' ) term .

term           ::=
     factor    |
     term ( '*' | '/' | 'MOD' ) factor .

factor              ::=
     [ '-' ] constant .

constant       ::=
     '(' constant_expr ')' |
     constant_name                 |
     int_constant          |
     char_constant                 |
     string_constant       |
     date_constant                 .

constant_id         ::=
     id .

constant_name  ::=
     name .
```

DECLARAÇÃO DE DOMÍNIOS DE ATRIBUTOS

```
value_set_decl ::=
     VALUE_SET ( vs_decl )+ .

vs_decl        ::=
     value_set_id ':' value_set ';' .

value_set      ::=
     value_set_name
|
     ENUM '{' ( constant_id // ',' ) '}'      |
     STRING '[' constant_expr ']'
        [ MATCHES string_constant ]
|
     BYTES '[' constant_expr ']'
|
     value_set_name SUBR
        '[' constant_expr '..' constant_expr ']'
|
     value_set_name ARRAY '[' constant_expr ']'
|
     STRUCT ( attribute // ';' ) END
|
     UNION  ( attribute // ';' ) END
.

value_set_id   ::=
     id .

value_set_name ::=
     name .
```

DECLARAÇÃO DE TIPOS DE OBJETOS

```
object_kind_decl    :=
        supertype_decl          |
        aggregation_decl        |
        set_decl                |
        object_type_decl        .

supertype_decl :=
        SUPER [ TYPE ] object_type_id
            super_desc END object_type_name ';' .

super_desc              :=
        [ attribute_clause ]
        [ version_clause   ]
        [ subtype_clause   ]
          cardinality_clause .

aggregation_decl    ::=
        AGGREGATION [ TYPE ] object_type_id
            aggregation_desc END object_type_name
';' .

aggregation_desc    ::=
        [ attribute_clause ]
        [ version_clause   ]
        [ component_clause ]
        cardinality_clause .

set_decl        ::=
        SET [ TYPE ] object_type_id
            set_desc END object_type_name ';' .

set_desc        ::=
        [ attribute_set_clause ]
        [ version_clause       ]
        [ member_clause        ]
        cardinality_clause     .

object_type_decl    ::=
        OBJECT [ TYPE ] object_type_id
            object_desc END object_type_name ';' .

object_desc             ::=
        [ attribute_clause ]
        [ version_clause   ]
        cardinality_clause .

object_type_id ::=
        id .

object_type_name    ::=
        name .
```

Attributos

```
attribute_clause    ::=
     ATTRIBUTES  ( attribute // ';' ) [ UNIQUE
          ( '(' unique_attribute ')' // ';' ) ] .

attribute           ::=
     attribute_id  ':'  value_set  .

unique_attribute    ::=
     ( attribute_name // ',' ) .

attribute_id    ::=
     id .

attribute_name ::=
     name .


attribute_set_clause    ::=
     ATTRIBUTES  ( attribute_set  ';' )  [ UNIQUE
          ( '(' unique_attribute ')' // ';' ) ] .

attribute_set   ::=
     attribute_id  ':'  value_set |
     attribute_id  ':'  att_function   .

att_function    ::=
     function_name  '('  object_type_name  '.'
          attribute_name  ')'  .

function_name   ::=
     COUNT     |
     SUM       |
     AVG       |
     MIN       |
     MAX       .
```

Versões

```
version_clause ::=
     VERSIONS  version_kind  .

version_kind    ::=
     LINEAR    |
     TREELIKE  |
     ACYCLIC   .
```

Subtipos

```
subtype_clause ::=
     SUBTYPES ( object_type_name // ',' ) .
```

Componentes

```
component_clause          ::=
     COMPONENTS ( comp_denotation // ',' ) .

comp_denotation           ::=
     object_type_denot  |
     relship_type_name  .

object_type_denot    ::=
     object_type_name  [  implicity_card  ]  .

implicity_card       ::=
     [  AT  LEAST  card_const  ]     |
     [  AT  MOST   card_const  ]     .

card_const                ::=
     int_constant  .

relship_type_name    ::=
     name  .
```

Membros

```
member_clause  ::=
     MEMBERS  ( object_type_name  //  ','  )  .
```

Cardinalidade em Relacionamentos

```
cardinality_clause  ::=
     [  AT  LEAST  ONCE  '('  ( role_denot  //  ','  )
')'  ]  |
     [  AT  MOST  ONCE   '('  ( role_denot  //  ','  )
')'  ]  .

role_denotation          ::=
     relship_type_name  [  '.'  role_name  ]  .

role_name           ::=
     name  .
```

DECLARAÇÃO DE RELACIONAMENTOS

```
relship_type_decl   ::=
     RELSHIP  [ TYPE ]  relship_type_id
          relship_type
     END  relship_type_name  ';'  .

relship_type        :=
     RELATES  ( role  //  ','  )
     [  attribute_clause  ]  .

role                :=
     [  role_id  ':'  ]  object_type_name  .
```

ANEXO 2

ESQUEMA DO DICIONÁRIO DE DADOS BD_PAC

```
SCHEMA    emeta

   /* *** constant definitions *** */

   CONST
      NAME_LENGTH = 32 ;


      EDD_CH = 0 ;    /* character   */
      EDD_IN = 1 ;    /* integer     */
      EDD_TI = 2 ;    /* time        */
      EDD_BO = 3 ;    /* boolean     */
      EDD_LF = 4 ;    /* long field  */
      EDD_SG = 5 ;    /* string      */
      EDD_BY = 6 ;    /* bytes       */
      EDD_AR = 7 ;    /* array       */
      EDD_SR = 8 ;    /* subrange    */
      EDD_EN = 9 ;    /* enum        */
      EDD_ST = 10 ;   /* structure   */
      EDD_UN = 11 ;   /* union       */
      EDD_LO = 12 ;   /* long        */
      EDD_FL = 13 ;   /* float       */
      EDD_DO = 14 ;   /* double      */
      EDD_NA = 15 ;   /* name        */
      EDD_ND = 16 ;   /* not defined */

      EDD_MINIMUM = 0 ;
      EDD_MAXIMUM = 1 ;
      EDD_ABSOLUTE = 2 ;


      /* functions for calculating the derivated */
      /* attributes of association */

      EFUN_COUNT = 0;   /* count    */
      EFUN_SUM   = 1;   /* sum      */
      EFUN_AVG   = 2;   /* average  */
      EFUN_MIN   = 3;   /* maximum  */
      EFUN_MAX   = 4;   /* minimum  */
```

```
/* *** value set declarations *** */

VALUE_SET
   Name : STRING [ NAME_LENGTH ] ;
   Id : INT ;
   Location : ENUM { SERVER, WS } ;


/* *** object type declarations *** */


OBJECT TYPE   edbschema

   ATTRIBUTES

      name : Name;/* name of schema as it has       */
                  /* been declared in DDL-program  */
     id : Id ;    /* system defined unique key     */
      repr : LONG_FIELD ;
                  /* packed representation of the  */
                  /* defined schema.               */
      location : Location
                  /* location of the schema :      */
                  /* server or workstation         */

   STRUCTURE IS  /* structural representation of   */
                 /* defined schema as a complex    */
                 /* DODM object                    */

      object_type,
      generalisation_r,
      aggregation_r,
      association_r,
      att_dependence_r

END   edbschema ;
```

```
OBJECT TYPE   object_type

  ATTRIBUTES
    name : Name;
    id : Id;
    has_version : BOOL;
    version_kind : INT;
    obj_kind : INT;
    obj_att_len : INT

  STRUCTURE
    type_attribute

END object_type ;


OBJECT TYPE   type_attribute

  ATTRIBUTES
    name : Name ;      /*user attribute name          */
    id : Id ;          /*  system attribute identi-*/
                       /*  fier. This identif. is   */
                       /*  unique in object- or re-*/
                       /*  lationship type decl.    */
    kind : INT ;       /*  value set flags for :    */
                       /*  CHAR, INT, BOOLEAN,..  ..*/
                       /*  TIME, LONG_FIELD,        */
                       /*  SUBRANGE, ARRAY,         */
                       /*  BYTES, STRING, ENUM,     */
                       /*  STRUCT, UNION,           */
                       /*  LONG, FLOAT, DOUBLE      */
    defined : BOOL;    /*  explicitely defined      */
                       /*  or "herdado"            */
    calc : BOOL

END type_attribute ;
```

```
/* *** relationship type declarations *** */

RELSHIP TYPE generalisation_r

  RELATES
     super : object_type,
     sub : object_type

END generalisation_r ;


RELSHIP TYPE aggregation_r

  RELATES
     aggreg : object_type,
     compon : object_type

  ATTRIBUTES
      card_min : INT;
      card_max : INT;
      comp_type : INT

END aggregation_r ;


RELSHIP TYPE association_r

  RELATES
     set : object_type,
     memb : object_type

END association_r ;


RELSHIP TYPE att_dependence_r

  RELATES
      definer : type_attribute,
      defined : type_attribute

  ATTRIBUTES
     function : INT   /* functions flags for :    */
                      /*  EFUN_COUNT,             */
                      /* EFUN_SUM,                */
                      /*  EFUN_AVG,               */
                      /*  EFUN_MIN,               */
                      /*  EFUN_MAX                */

END att_dependence_r ;

END emeta
```

ANEXO 3

ARQUIVO DE DECLARAÇÕES EM LINGUAGEM C

```
/*****************************************************/
/*     D A M O K L E S        (version 2.0)          */
/*     DDL - Compiler                                */
/*****************************************************/
/*      Compiled at: Sat Oct 13 22:59:39 1990        */
/*****************************************************/

#ifndef  TRUE
#define  TRUE  1
#endif

#ifndef  FALSE
#define  FALSE  0
#endif


/*****************************************************/
/*           constant    definitions                 */
/*****************************************************/

/*****************************************************/
/*           enum    constant    definitions          */
/*****************************************************/

#define    BRANCO    16
#define    PRETO     17
#define    AMARELO   18
#define    AZUL      19
#define    VERMELHO  20
#define    VERDE     21
#define    CINZA     22
#define    MAGENTA   23
#define    LACO      24
#define    KAPA      25
#define    NILO      26
#define    IN   27
#define    OUT  28
#define    INOUT     29
#define    BUS  30
#define    CLOCK     31
#define    CONTROL   32
#define    INTERGER  33
#define    TERMINAL  34
```

```
/*****************************************************/
/*      object-,                        relationship-,   */
/*      attribute-,                   role-              */
/*      and     synonym-                name    definit. */
/*****************************************************/

#define   AGENCIA  35
#define   AG_ALT   38
#define   AG_NOME  39
#define   ALTERNATIVA   41
#define   ALT_VER  40
#define   ARQ_FONTE     56
#define   ARQ_OBJETO    57
#define   COMPORT  64
#define   COMPORTAMENTO      65
#define   CON_FAC  71
#define   CON_INT  72
#define   DATA     70
#define   DATA_CRIACAO  37
#define   DEF_INT_ARQK  43
#define   DEF_INT_ARQL  42
#define   DEF_INT_ARQN  44
#define   DESC_GRAF_COR      46
#define   DESC_GRAF_ESC      68
#define   DESC_GRAF_PONTOS   45
#define   DESC_GRAF_POS      53
#define   DESC_GRAF_POS_TXT  47
#define   DIMENSAO  50
#define   MODELO_SIM    69
#define   NIVEL     55
#define   NOME      49
#define   OCORRENCIA_AG      67
#define   PROJETISTA    36
#define   SENTIDO   51
#define   SINAIS_I  48
#define   SINAIS_V  60
#define   SIN_OC    66
#define   SIN_VS_C  62
#define   SIN_VS_P  61
#define   TIPO_I    74
#define   TIPO_SIN  52
#define   TIPO_V    73
#define   VERSAO    63
#define   VERSAO_COMPOSTA    59
#define   VERSAO_PRIMITIVA   58
#define   VER_GEN   54

/*****************************************************/
/*          value set  definitions                   */
/*****************************************************/

typedef  struct
         {
```

```
            short dia;
            short mes;
            short ano;
        } Data;

typedef  short  Cores;

typedef  struct
         {
             float abscissa;
             float ordenada;
         } Ponto;

typedef  short  Niveis;

typedef  short  Sentidos;

typedef  short  Tipos_sin;


/*************************************************/
/*       object and relationship   definitions      */
/*************************************************/

typedef  struct
         {
             char projetista [ 31 ];
             Data data_criacao;
         } Agencia;

typedef  struct
         {
             char ag_nome [ 11 ];
         } Ag_alt;

typedef  struct
         {
             char projetista [ 31 ];
             Data data_criacao;
             Cores desc_graf_cor;
             Ponto desc_graf_pos_txt;
         } Alternativa;

typedef  struct
         {
             char ag_nome [ 11 ];
         } Alt_ver;

typedef  struct
         {
             char nome [ 11 ];
             short dimensao;
             Sentidos sentido;
             Tipos_sin tipo_sin;
```

```
            Ponto desc_graf_pos;
            Ponto desc_graf_pos_txt;
        } Sinais_i;

typedef  struct
         {
            char projetista [ 31 ];
            Data data_criacao;
            Niveis nivel;
            char ag_nome [ 11 ];
         } Ver_gen;

typedef  struct
         {
            char nome [ 11 ];
            short dimensao;
            Sentidos sentido;
            Tipos_sin tipo_sin;
            Ponto desc_graf_pos;
            Ponto desc_graf_pos_txt;
         } Sinais_v;

typedef  struct
         {
            char projetista [ 31 ];
            Data data_criacao;
            Niveis nivel;
            char ag_nome [ 11 ];
          } Versao_primitiva;

typedef  struct
         {
            char projetista [ 31 ];
            Data data_criacao;
            Niveis nivel;
            char ag_nome [ 11 ];
          } Versao_composta;

typedef  struct
         {
            char nome [ 11 ];
            short dimensao;
            Sentidos sentido;
            Tipos_sin tipo_sin;
            Ponto desc_graf_pos;
            Ponto desc_graf_pos_txt;
         } Sin_vs_p;

typedef  struct
         {
            char nome [ 11 ];
            short dimensao;
            Sentidos sentido;
            Tipos_sin tipo_sin;
```

```
            Ponto desc_graf_pos;
            Ponto desc_graf_pos_txt;
        } Sin_vs_c;

typedef  struct
         {
            char nome [ 11 ];
            short dimensao;
         } Sin_oc;

typedef  struct
         {
            char nome [ 11 ];
            Ponto desc_graf_pos;
            float desc_graf_esc;
            Cores desc_graf_cor;
            Ponto desc_graf_pos_txt;
         } Ocorrencia_ag;

typedef  struct
         {
            char projetista [ 31 ];
            Data data;
         } Modelo_sim;


/*****************************************************/
/*    end    of   definition    file    ( DAMOKLES )   */
/*****************************************************/
```

ANEXO 4

ESQUEMAS REFERENTES À MODELAGEM DO CAPÍTULO 7

Esquema BD_PAC para os dados do AMPLO

```
SCHEMA AMPLO

/*
 ******************************************************
*    Definicao de dominios                             *
 ******************************************************
*/
    VALUE_SET
       DATA: STRUCT
                  dia: INT SUBR [1..31];
                  mes: INT SUBR [1..12];
                  ano: INT SUBR [1980..2010]
             END;
        CORES: ENUM { BRANCO, PRETO, AMARELO, AZUL,
                      VERMELHO, VERDE, CINZA, MAGENTA };
       PONTO: STRUCT
                  abscissa: FLOAT;
                  ordenada: FLOAT
              END;
        NIVEIS: ENUM { LACO, KAPA, NILO };
        SENTIDOS: ENUM { IN, OUT, INOUT };
        TIPOS_SIN: ENUM { BUS, CLOCK, CONTROL,
                          INTERGER, TERMINAL };
/*
 ******************************************************
*    Definicao dos objetos                             *
 ******************************************************
*/
    AGGREGATION TYPE AGENCIA
       ATTRIBUTES
           projetista: STRING[30];
           data_criacao: DATA
       COMPONENT
          AG_ALT
    END AGENCIA;

    SUPER TYPE AG_ALT
       ATTRIBUTES
           ag_nome: STRING[10]
       SUBTYPE
          ALT_VER
    END AG_ALT;
```

```
AGGREGATION TYPE ALTERNATIVA
   ATTRIBUTES
        projetista: STRING[30];
        data_criacao: DATA;
        def_int_arql: LONG_FIELD;
        def_int_arqk: LONG_FIELD;
        def_int_arqn: LONG_FIELD;
        desc_graf_pontos: LONG_FIELD;
        desc_graf_cor: CORES;
        desc_graf_pos_txt: PONTO
   COMPONENT
       ALT_VER, SINAIS_I
END ALTERNATIVA;

SUPER TYPE ALT_VER
   SUBTYPE
      VER_GEN
END ALT_VER;

SUPER TYPE VER_GEN
   ATTRIBUTES
        projetista: STRING[30];
        data_criacao: DATA;
        nivel: NIVEIS;
        arq_fonte: LONG_FIELD;
        arq_objeto: LONG_FIELD
   SUBTYPE
         VERSAO_PRIMITIVA, VERSAO_COMPOSTA
END VER_GEN;


SUPER TYPE SINAIS_I
   ATTRIBUTES
        nome: STRING[10];
        dimensao: INT;
        sentido: SENTIDOS;
        tipo_sin: TIPOS_SIN;
        desc_graf_pos: PONTO;
        desc_graf_pos_txt: PONTO
   SUBTYPE
      SINAIS_V
END SINAIS_I;

SUPER TYPE SINAIS_V
   SUBTYPE
       SIN_VS_P, SIN_VS_C
END SINAIS_V;

AGGREGATION TYPE VERSAO
   COMPONENT
       VER_GEN, SINAIS_V
END VERSAO;
```

```
OBJECT TYPE SIN_VS_P
END SIN_VS_P;

OBJECT TYPE SIN_VS_C
   AT MOST ONCE ( con_fac )
END SIN_VS_C;

OBJECT TYPE COMPORT
   ATTRIBUTES
        comportamento: LONG_FIELD
END COMPORT;

OBJECT TYPE SIN_OC
   ATTRIBUTES
       nome: STRING[10];
      dimensao: INT
   AT MOST ONCE (con_fac,
                 con_int.in,
                 con_int.out)
END SIN_OC;

AGGREGATION TYPE VERSAO_PRIMITIVA
   COMPONENT
       SIN_VS_P, COMPORT
END VERSAO_PRIMITIVA;

AGGREGATION TYPE OCORRENCIA_AG
   ATTRIBUTES
       nome: STRING[10];
       desc_graf_pos: PONTO;
       desc_graf_esc: FLOAT;
       desc_graf_cor: CORES;
       desc_graf_pos_txt: PONTO
   COMPONENT
      SIN_OC
END OCORRENCIA_AG;

AGGREGATION TYPE VERSAO_COMPOSTA
   COMPONENT
        OCORRENCIA_AG, SIN_VS_C, con_int,
       con_fac, tipo_v, tipo_i
END VERSAO_COMPOSTA;

SET TYPE MODELO_SIM
   ATTRIBUTES
       Projetista: STRING[30];
      Data: DATA
  MEMBER
       VERSAO_PRIMITIVA
END MODELO_SIM;
```

```
/*
 ***************************************************
*    Definicao dos relacionamentos                  *
 ***************************************************
*/
    RELSHIP TYPE con_int
       RELATES
           in:SIN_OC,
           out:SIN_OC
    END con_int;

    RELSHIP TYPE con_fac
       RELATES
           SIN_VS_C,
           SIN_OC
    END con_fac;

    RELSHIP TYPE tipo_i
       RELATES
           OCORRENCIA_AG,
           ALTERNATIVA
    END tipo_i;

    RELSHIP TYPE tipo_v
       RELATES
           OCORRENCIA_AG,
           VERSAO
    END tipo_v;

END amplo
/*
 ***************************************************
*    final do esquema AMPLO                         *
 ***************************************************
*/
```

Esquema DAMOKLES para os dados do AMPLO

```
SCHEMA  AMPLO

/*
 ******************************************************
*    Definicao de dominios                             *
 ******************************************************
*/
    VALUE_SET
        DATA:  STRUCT
                    dia:  INT SUBR [1..31];
                    mes:  INT SUBR [1..12];
                     ano:  INT SUBR [1980..2010];
               END;
         CORES:  ENUM  {  BRANCO,  PRETO,  AMARELO,
                          AZUL,  VERMELHO,
                           VERDE,  CINZA,  MAGENTA  };
        PONTO:  STRUCT
                    abscissa:  FLOAT;
                    ordenada:  FLOAT;
                END;
        NIVEIS:  ENUM  {  LACO,  KAPA,  NILO  };
        SENTIDOS:  ENUM  {  IN,  OUT,  INOUT  };
        TIPOS_SIN:  ENUM  {  BUS,  CLOCK,  CONTROL,
                                  INTERGER,  TERMINAL  };
```

```
/*
 ******************************************************
*    Definicao dos objetos                             *
 ******************************************************
*/
    OBJECT TYPE AGENCIA
       ATTRIBUTES
           nome: STRING[10];
           projetista: STRING[30];
           data_criacao: DATA;
           nro_alternativas: INT
        VERSIONS TREELIKE
       (
           ATTRIBUTES
                projetista: STRING[30];
                data_criacao: DATA;
                def_int_arql: LONG_FIELD;
                def_int_arqk: LONG_FIELD;
                def_int_arqn: LONG_FIELD;
                desc_graf_pontos: LONG_FIELD;
                desc_graf_cor: CORES;
                desc_graf_pos_txt: PONTO
            VERSIONS TREELIKE
           (
               ATTRIBUTES
                    projetista: STRING[30];
                    data_criacao: DATA;
                    nivel: NIVEIS;
                    arq_fonte: LONG_FIELD;
                    arq_objeto: LONG_FIELD
               STRUCTURE IS
                  SINAIS_VS
                AT MOST ONCE (esp_vs)
           )
            STRUCTURE IS
               SINAIS_I
       )
    END AGENCIA;

    OBJECT TYPE SINAIS_I
       ATTRIBUTES
           nome: STRING[10];
           dimensao: INT;
           sentido: SENTIDOS;
           tipo_lasso: TIPOS_LASSO;
           tipo_kapa: TIPOS_KAPA;
           tipo_nilo: TIPOS_NILO;
           desc_graf_pos: PONTO;
           desc_graf_pos_txt: PONTO
       AT MOST ONCE (sin_av)
    END SINAIS_I;
```

```
OBJECT TYPE SINAIS_VS
   ATTRIBUTES
       nome: STRING[10];
       dimensao: INT;
       sentido: SENTIDOS;
       tipo_lasso: TIPOS_LASSO;
       tipo_kapa: TIPOS_KAPA;
       tipo_nilo: TIPOS_NILO;
       desc_graf_pos: PONTO;
       desc_graf_pos_txt: PONTO
   AT MOST ONCE (sin_av,
                 sin_vp,
                 sin_vc)
END SINAIS_VS;

OBJECT TYPE SIN_VS_P
   ATTRIBUTES
       nome: STRING[10];
       dimensao: INT;
   AT MOST ONCE (sin_vp)
END SIN_VS_P;

OBJECT TYPE COMPORT
   ATTRIBUTES
       comportamento: LONG_FIELD
END COMPORT;

OBJECT TYPE SIN_OC
   ATTRIBUTES
       nome: STRING[10];
       dimensao: INT;
   AT MOST ONCE (con_fac,
                 con_int)
END SIN_OC;

OBJECT TYPE SIN_INT
   ATTRIBUTES
       nome: STRING[10];
       dimensao: INT;
   AT MOST ONCE (con_int)
END SIN_INT;

OBJECT TYPE SIN_VS_C
   ATTRIBUTES
       nome: STRING[10];
       dimensao: INT;
   AT MOST ONCE (sin_vc,
                 con_fac)
END SIN_VS_C;
```

```
OBJECT TYPE PRIMITIVA
   STRUCTURE IS
       SIN_VS_P, COMPORT, sin_vp
END PRIMITIVA;

OBJECT TYPE OCORR_AG
   ATTRIBUTES
       nome: STRING[10];
       desc_graf_pos: PONTO;
       desc_graf_esc: FLOAT;
       desc_graf_cor: CORES;
       desc_graf_pos_txt: PONTO
   STRUCTURE IS
      SIN_OC
   AT MOST ONCE (tipo_v,
                 tipo_i)
END OCORR_AG;

OBJECT TYPE COMPOSTA
   STRUCTURE IS
       OCORR_AG, con_int, SIN_INT, con_fac,
       SIN_VS_C, sin_vc, tipo_v, tipo_i
END COMPOSTA;

OBJECT TYPE ESPEC_VERSAO
   IS UNION OF
       PRIMITIVA, COMPOSTA
   AT MOST ONCE (esp_vs);
END ESPEC_VERSAO;

SET TYPE MODELO_SIM
   ATTRIBUTES
       Projetista: STRING[30];
      Data: DATA
   MEMBER
       VERSAO_PRIMITIVA
END MODELO_SIM;

/*
 ***************************************************
*    Definicao dos relacionamentos                 *
 ***************************************************
*/
RELSHIP TYPE sin_av
   RELATES
      SINAIS_I,
      SINAIS_VS
END sin_av;
```

```
    RELSHIP TYPE sin_vp
       RELATES
           SINAIS_VS,
           SIN_VS_P
    END sin_vp;

    RELSHIP TYPE con_int
       RELATES
           SIN_OC,
           SIN_INT
    END con_int;

    RELSHIP TYPE con_fac
       RELATES
           SIN_VS_C,
           SIN_OC
    END con_fac;

    RELSHIP TYPE sin_vc
       RELATES
           SIN_VS_C,
           SINAIS_VS
    END sin_vc;

    RELSHIP TYPE tipo_v
       RELATES
           OCORR_AG,
            AGENCIA.VERSION.VERSION
    END tipo_v;

    RELSHIP TYPE tipo_i
       RELATES
           OCORR_AG,
            AGENCIA.VERSION
    END tipo_i;

    RELSHIP TYPE esp_vs
       RELATES
           espec_ver,
            AGENCIA.VERSION.VERSION
    END esp_vs;

END AMPLO
/*
 ***************************************************
*    final do esquema AMPLO                         *
 ***************************************************
*/
```

Esquema BD_PAC para os dados do AMADEUS

```
SCHEMA AMADEUS

/*
 ******************************************************
*    Definicao de dominios                             *
 ******************************************************
*/

    VALUE_SET
       DATA: STRUCT
                  dia: INT SUBR [1..31];
                  mes: INT SUBR [1..12];
                  ano: INT SUBR [1980..2010];
             END;


/*
 ******************************************************
*    Definicao dos objetos                             *
 ******************************************************
*/
    OBJECT TYPE TAB_SIMB
       ATTRIBUTES
            nome_simb: STRING[15];
           tipo: STRING[15];
          valor: INT;
            escopo: LONG_FIELD;
    END TAB_SIMB;

    OBJECT TYPE NODO_P
       ATTRIBUTES
           simb: STRING[15]
       AT LEAST ONCE (p_nod)
    END NODO_P;

    AGGREGATION TYPE MODULO
       ATTRIBUTES
            particular: LONG_FIELD;
            cod_objeto: LONG_FIELD
       VERSION TREELIKE
       COMPONENTS
            TAB_SIMB, NODO_P, ref_at, mp_f, MODULO
       AT MOST ONCE (mp_f)
    END MODULO;
```

```
AGGREGATION TYPE PROGRAMAS
   ATTRIBUTOS
        linguagem: STRING[20];
        objetivo: LONG_FIELD;
        sistema: STRING[30]
   COMPONENTS
      MODULO, conex
END PROGRAMAS;

OBJECT TYPE SENTENCA
   ATTRIBUTES
       simb: STRING[20]
   AT LEAST ONCE (p_sen)
END SENTENCA;

AGGREGATION TYPE ESPECIFICACOES
   COMPONENTS
        SENTENCA, sp_f, ESPECIFICACOES
   AT MOST ONCE (sp_f)
END ESPECIFICACOES;

SUPER TYPE FORMAIS
   SUBTYPES
        PROGRAMAS, ESPECIFICACOES
END FORMAIS;

OBJECT TYPE INFORMAIS
   ATTRIBUTES
       texto: LONG_FIELD
END INFORMAIS;

SUPER TYPE TEXTUAIS
   SUBTYPES
       FORMAIS, INFORMAIS
END TEXTUAIS;

OBJECT TYPE NODO_G
   ATTRIBUTES
       simb: STRING[20]
   AT LEAST ONCE (p_nog)
END NODO_G;

AGGREGATION TYPE DIAGRAM
   COMPONENTS
       NODO_G, gp_f, DIAGRAM
   AT MOST ONCE (gp_f)
END DIAGRAM;
```

```
SUPER TYPE DOCUMENTOS
   ATTRIBUTES
       nome: STRING[30];
       autor: STRING[30];
       data_criacao: DATA;
      tipo: CHAR;
       descricao: LONG_FIELD
    VERSIONS TREELIKE
   SUBTYPES
       TEXTUAIS, DIAGRAM
END DOCUMENTOS;

SET TYPE CONFIGURAC
   ATTRIBUTES
       responsavel: STRING[30];
      data: DATA;
       observ: LONG_FIELD
   MEMBERS
       PROGRAMAS, ESPECIFICACOES,
       INFORMAIS, DIAGRAM
END CONFIGURAC;

OBJECT TYPE FATOR
   ATTRIBUTES
       f_simb: STRING[20]
END FATOR;

SET TYPE ALTERNATIVA
   MEMBERS
      FATOR
END ALTERNATIVA;

AGGREGATION TYPE CORPO
   ATTRIBUTES
       simb: STRING[30];
      tipo: CHAR
   COMPONENTS
       ALTERNATIVA
END CORPO;

OBJECT TYPE CABECALHO
   ATTRIBUTES
       cab_txt: STRING[30];
      tipo: CHAR
END CABECALHO;

AGGREGATION TYPE PRODUCOES
   COMPONENTS
       CORPO, CABECALHO, PRODUCOES, fat_p
END PRODUCOES;
```

```
OBJECT TYPE ATRIBUTOS
   ATTRIBUTES
       simb: STRING[15];
       tipo: STRING[15];
      valor: INT
   AT LEAST ONCE (p_atr)
END ATRIBUTOS;

OBJECT TYPE ACOES
   ATTRIBUTES
       simb: STRING[15];
       codigo: LONG_FIELD
   AT LEAST ONCE (p_ac)
END ACOES;

OBJECT TYPE L_TEXTUAIS
END L_TEXTUAIS;

OBJECT TYPE L_DIAGRAM
   ATTRIBUTOS
       dados_graf: LONG_FIELD
END L_DIAGRAM;

SUPER TYPE LEXICOS
   ATTRIBUTES
       simb: STRING[10]
   SUBTYPES
       L_TEXTUAIS, L_DIAGRAM
   AT LEAST ONCE (p_lex)
END LEXICOS;

AGGREGATION TYPE GRAMATICA
   ATTRIBUTES
       linguagem: STRING[20];
       especificador: STRING[30];
      data_especif: DATA;
   COMPONENTS
       PRODUCOES, ATRIBUTOS, ACOES, LEXICOS,
      p_atr, p_ac, p_lex
END GRAMATICA;
```

```
/*
 ******************************************************
*    Definicao dos relacionamentos                     *
 ******************************************************
*/
    RELSHIP TYPE ref_at
       RELATES
          NODO_P,
          TAB_SIMB
    END ref_at;

    RELSHIP TYPE mp_f
       RELATES
          NODO_P,
          MODULO
    END mp_f;

    RELSHIP TYPE conex
       RELATES
           chamador: MODULO,
           chamado: MODULO
    END conex;

    RELSHIP TYPE sp_f
       RELATES
           SENTENCA,
           ESPECIFICACOES
    END sp_f;

    RELSHIP TYPE gp_f
       RELATES
          NODO_G,
          DIAGRAM
    END gp_f;

    RELSHIP TYPE refer
       RELATES
           refte: DOCUMENTOS,
           refdo: DOCUMENTOS
    END refer;

    RELSHIP TYPE fat_p
       RELATES
          FATOR,
          PRODUCOES
    END fat_p;

    RELSHIP TYPE p_atr
       RELATES
          ATRIBUTOS,
          PRODUCOES
    END p_atr;
```

```
RELSHIP TYPE p_ac
   RELATES
      ACOES,
      PRODUCOES
END p_ac;

RELSHIP TYPE p_lex
   RELATES
      LEXICOS,
      PRODUCOES
END p_lex;

RELSHIP TYPE g_dia
   RELATES
      DIAGRAM,
      GRAMATICA
END g_dia;

RELSHIP TYPE g_esp
   RELATES
      ESPECIFICACOES,
      GRAMATICA
END g_esp;

RELSHIP TYPE g_prg
   RELATES
      PROGRAMAS,
      GRAMATICA
END g_prg;

RELSHIP TYPE p_nog
   RELATES
      NODO_G,
      PRODUCOES
END p_nog;

RELSHIP TYPE p_sen
   RELATES
      SENTENCA,
      PRODUCOES
END p_sen;

RELSHIP TYPE p_nod
   RELATES
      NODO_P,
      PRODUCOES
END p_nod;

END AMADEUS
```

Esquema DAMOKLES para os dados do AMADEUS

```
SCHEMA AMADEUS

/*
 ******************************************************
*    Definicao de dominios                             *
 ******************************************************
*/

    VALUE SET
       DATA: STRUCT
                  dia: INT SUBR [1..31];
                  mes: INT SUBR [1..12];
                  ano: INT SUBR [1980..2010];
             END;


/*
 ******************************************************
*    Definicao dos objetos                             *
 ******************************************************
*/
    OBJECT TYPE TAB_SIMB
       ATTRIBUTES
            nome_simb: STRING[15];
           tipo: STRING[15];
          valor: INT;
           escopo: LONG_FIELD;
    END TAB_SIMB;

    OBJECT TYPE NODO_P
       ATTRIBUTES
           simb: STRING[15]
       AT LEAST ONCE (p_nod)
    END NODO_P;
```

```
OBJECT TYPE MODULO
   ATTRIBUTES
       nome: STRING[20];
       programador: STRING[30];
       data: DATA;
       cod_objeto: LONG_FIELD
   VERSION TREELIKE
  (
       delta: LONG_FIELD
  )
   STRUCTURE IS
       TAB_SIMB, NODO_P, ref_at, mp_f, MODULO
   AT MOST ONCE (mp_f)
END MODULO;

OBJECT TYPE PROGRAMAS
   ATTRIBUTOS
       nome: STRING[20];
       programador: STRING[20];
       linguagem: STRING[20];
       data: DATA;
       objetivo: LONG_FIELD;
       sistema: STRING[30]
   VERSIONS TREELIKE
  (
       delta: LONG_FIELD
  )
   STRUCTURE IS
       MODULO, MODULO.VERSION, conex
END PROGRAMAS;

OBJECT TYPE SENTENCA
   ATTRIBUTES
       simb: STRING[20]
   AT LEAST ONCE (p_sen)
END SENTENCA;

OBJECT TYPE ESPECIFICACOES
   VERSIONS TREELIKE
  (
       delta: LONG_FIELD
  )
   STRUCTURE IS
       SENTENCA, sp_f, ESPECIFICACOES
   AT MOST ONCE (sp_f)
END ESPECIFICACOES;
```

```
OBJECT TYPE FORMAIS
    ATTRIBUTES
        autor: STRING[30];
        data: DATA;
        tipo: CHAR
    VERSIONS TREELIKE
   (
        delta: LONG_FIELD
   )
    STRUCTURE IS
         PROGRAMAS, PROGRAMAS.VERSION,
         ESPECIFICACOES, ESPECIFICACOES.VERSION
END FORMAIS;

OBJECT TYPE INFORMAIS
    ATTRIBUTES
        autor: STRING[30];
        data: DATA;
        tipo: CHAR;
        texto: LONG_FIELD
    VERSIONS TREELIKE
   (
        delta: LONG_FIELD
   )
END INFORMAIS;

OBJECT TYPE TEXTUAIS
    VERSIONS TREELIKE
   (
        delta: LONG_FIELD
   )
    STRUCTURE IS
         FORMAIS, FORMAIS.VERSION,
         INFORMAIS, INFORMAIS.VERSION
END TEXTUAIS;

OBJECT TYPE NODO_G
    ATTRIBUTES
        simb: STRING[20]
    AT LEAST ONCE (p_nog)
END NODO_G;
```

```
OBJECT TYPE DIAGRAM
   ATTRIBUTES
       nome: STRING[20];
       autor: STRING[30];
      notacao: CHAR
   VERSIONS TREELIKE
  (
       delta; LONG_FIELD
  )
   STRUCTURE IS
       NODO_G, gp_f, DIAGRAM
   AT MOST ONCE (gp_f)
END DIAGRAM;

OBJECT TYPE DOCUMENTOS
   ATTRIBUTES
       nome: STRING[30];
       autor: STRING[30];
       data_criacao: DATA;
       objetivo: LONG_FIELD
   VERSIONS TREELIKE
  (
       delta: LONG_FIELD
  )
   STRUCTURE IS
        TEXTUAIS, TEXTUAIS.VERSION,
        DIAGRAM, DIAGRAM.VERSION
END DOCUMENTOS;

OBJECT TYPE CONFIGURAC
   ATTRIBUTES
        responsavel: STRING[30];
      data: DATA;
       observ: LONG_FIELD
   STRUCTURE IS
        PROGRAMAS, PROGRAMAS.VERSION,
         ESPECIFICACOES, ESPECIFICACOES.VERSION,
        INFORMAIS, INFORMAIS.VERSION,
        DIAGRAM, DIAGRAM.VERSION
END CONFIGURAC;

OBJECT TYPE FATOR
   ATTRIBUTES
       f_simb: STRING[20]
END FATOR;

OBJECT TYPE ALTERNATIVA
   STRUCTURE IS
     FATOR
END ALTERNATIVA;
```

```
OBJECT TYPE CORPO
   ATTRIBUTES
       simb: STRING[30];
      tipo: CHAR
   STRUCTURE IS
      ALTERNATIVA
END CORPO;

OBJECT TYPE CABECALHO
   ATTRIBUTES
       cab_txt: STRING[30];
      tipo: CHAR
END CABECALHO;

OBJECT TYPE PRODUCOES
   STRUCTURE IS
        CORPO, CABECALHO, PRODUCOES, fat_p
END PRODUCOES;

OBJECT TYPE ATRIBUTOS
   ATTRIBUTES
       simb: STRING[15];
       tipo: STRING[15];
      valor: INT
   AT LEAST ONCE (p_atr)
END ATRIBUTOS;

OBJECT TYPE ACOES
   ATTRIBUTES
       simb: STRING[15];
       codigo: LONG_FIELD
   AT LEAST ONCE (p_ac)
END ACOES;

OBJECT TYPE L_TEXTUAIS
   ATTRIBUTES
       simb: STRING[10]
END L_TEXTUAIS;

OBJECT TYPE L_DIAGRAM
   ATTRIBUTOS
       dados_graf: LONG_FIELD
END L_DIAGRAM;

OBJECT TYPE LEXICOS
   ATTRIBUTES
       simb: STRING[10]
   STRUCTURE IS
       L_TEXTUAIS, L_DIAGRAM
   AT LEAST ONCE (p_lex)
END LEXICOS;
```

```
OBJECT TYPE GRAMATICA
   ATTRIBUTES
        linguagem: STRING[20];
        especificador: STRING[30];
        data_especif: DATA;
   STRUCTURE IS
        PRODUCOES, ATRIBUTOS, ACOES, LEXICOS,
        p_atr, p_ac, p_lex
END GRAMATICA;

/*
 ************************************************
*    Definicao dos relacionamentos               *
 ************************************************
*/
   RELSHIP TYPE ref_at
      RELATES
         NODO_P,
         TAB_SIMB
   END ref_at;

   RELSHIP TYPE mp_f
      RELATES
         NODO_P,
         MODULO
   END mp_f;

   RELSHIP TYPE conex
      RELATES
         chamador: MODULO,
         chamado: MODULO
   END conex;

   RELSHIP TYPE sp_f
      RELATES
         SENTENCA,
         ESPECIFICACOES
   END sp_f;

   RELSHIP TYPE gp_f
      RELATES
         NODO_G,
         DIAGRAM
   END gp_f;

   RELSHIP TYPE refer
      RELATES
         refte: DOCUMENTOS,
         refdo: DOCUMENTOS
   END refer;
```

```
RELSHIP TYPE fat_p
   RELATES
      FATOR,
      PRODUCOES
END fat_p;

RELSHIP TYPE p_atr
   RELATES
      ATRIBUTOS,
      PRODUCOES
END p_atr;

RELSHIP TYPE p_ac
   RELATES
      ACOES,
      PRODUCOES
END p_ac;

RELSHIP TYPE p_lex
   RELATES
      LEXICOS,
      PRODUCOES
END p_lex;

RELSHIP TYPE g_dia
   RELATES
      DIAGRAM,
      GRAMATICA
END g_dia;

RELSHIP TYPE g_esp
   RELATES
       ESPECIFICACOES,
      GRAMATICA
END g_esp;

RELSHIP TYPE g_prg
   RELATES
      PROGRAMAS,
      GRAMATICA
END g_prg;

RELSHIP TYPE p_nog
   RELATES
      NODO_G,
      PRODUCOES
END p_nog;
```

```
    RELSHIP TYPE p_sen
       RELATES
           SENTENCA,
           PRODUCOES
    END p_sen;

    RELSHIP TYPE p_nod
       RELATES
           NODO_P,
           PRODUCOES
    END p_nod;

END AMADEUS
```

BIBLIOGRAFIA

/ABR 88/ ABRAMOWICZ, K. et al. **DAMOKLES - database management system for design applications**; reference manual. Karlsruhe, Universität Karlsruhe, 1988.

/AFS 85/ AFSARMANESH, H. et al. An extensible object-oriented approach to database for VLSI/CAD. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 11., Stockholm, Aug. 21-23, 1985. **Proceedings**. Stockholm, SAS DATA/IBM, 1985. p.13-24.

/AFS 86/ AFSARMANESH, H. et al. Information management for VLSI/CAD. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTER, Port Chester, Oct. 6-9, 1986. **Proceedings**. Washington, IEEE, 1986. p.476-81.

/ATK 89/ ATKINSON, M.; BANCILHON, F.; DeWITT, D.; DITTRICH, K.; MAIER, D.; ZDONIK, S. **The object-oriented database system manifesto**. Le Chesnay, INRIA, 1989. (Rapport Technique Altair 30-89).

/ATW 85/ ATWOOD, T. M. An object-oriented DBMS for design support applications. In: COMPINT 85: COMPUTER AIDED TECHNOLOGIES, Montreal, Sept. 8-12, 1985. **Proceedings**. Washington, IEEE, 1985. p.299-307.

/BAN 88a/ BANCILHON, F. Object-oriented database systems. In: ACM SIGART-SIGMOD-SIGACT SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS, 7., Austin, Mar. 1988. **Proceedings.** New York, ACM, 1988.

/Ban 88b/ BANCILHON, F. et al. The design and implementation of O_2, an object-oriented database system. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED DATABASE SYSTEMS, 2., Bad Munster, Sept. 27-30, 1988. **Proceedings**. 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, **5**(1):3-26, Jan. 1987.

/BAT 84/ BATORY, D. S.; BUCHMANN, A. P. Molecular objects, abstract data type, and data models: a framework. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 20., Singapore, Aug. 1984. **Proceedings**. New York, IEEE, 1984. p. 172-82.

/BAT 85/ BATORY, D. S.; KIM, W. Modeling concepts for VLSI CAD objects. **ACM Transactions on Database Systems**, New York, **10**(3):322-46, Sept. 1985.

/BEC 88/ BECKER, K. Database support for a CAD environment for digital systems design. In: CONFERENCIA INTERNACIONAL DE CIÊNCIA DE LA COMPUTACION, 8., Santiago, Julio 4-8, 1988. **Actas**. Santiago, SCCC, 1988. p. 231-44.

/BEL 87/ BELKATIR, N.; ESTUBLIER, J. Experience with a data base of programs. **Sigplan Notices**, New York, **22**(1):84-91, Jan. 1987.

/BEN 82/ BENNETT, J. A database management system for design engineers. In: DESIGN AUTOMATION CONFERENCE, 19., Las Vegas, June 14-16, 1982. **Proceedings**. New York, IEEE, 1982. p. 268-73.

/BER 88/ BERKEL, T. et al. Modelling CAD-objects by abstraction. In: INTERNATIONAL CONFERENCE ON DATA AND KNOWLEDGE BASES, 3., Jerusalem, June 28-30, 1988. **Proceedings**. Jerusalem, 1988. p. 227-40

/BIL 89/ BILIRIS, A. Database support for evolving design objects. In: DESIGN AUTOMATION CONFERENCE, 26., Las Vegas, June 25-29, 1989. **Proceedings**. New York, IEEE, 1989. p. 258-63.

/BOK 88/ BOKLIS, V. Modelagem de dados em um ambiente de projeto de sistemas digitais. In: CONGRESSO NACIONAL DE INFORMÁTICA, 21., Rio de Janeiro, ago. 1988. **Anais**. Rio de Janeiro, SUCESU, 1988. p. 235-41.

/BRO 81/ BRODIE, M. L. Data abstraction for designing database-intensive applications. **Sigplan Notices**, New York, **16**(1):101-3, Jan. 1981.

/BRO 84/ BRODIE, M. L. On the development of data models. In: **ON CONCEPTUAL Modelling,** New York, Springer Verlag, 1984. p. 19-48.

/BUC 85/ BUCHMANN, A. P.; CELIS, C. P. An architecture and data model for CAD databases. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 11., Stockholm, Aug. 21-23, 1985. **Proceedings**. Stockholm, SAS DATA/IBM, 1985. p. 105-14.

/CAM 86/ CAMPBELL, R. H.; KIRSLIS, P. A. The SAGA project: a system for software development environments. In: IFIP WORKSHOP ON ADVANCED PROGRAMMING ENVIRONMENTS, Trondheim, June 16-18, 1986. **Proceedings**. Berlin, Springer Verlag, 1986. p. 142-55.

/CEL 90/ CELLARY, W.; JOMIER, G. Consistency of versions in object-oriented databases. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 16., Brisbane, 1990. **Proceedings**. Palo Alto, Morgan Kaufmann. 1990. p. 432-41.

/CHE 76/ CHEN, P. P. S. The entity-relationship model - toward a unified view of data. **ACM Transactions on Database Systems**, New York, **1**(1):9-36, Jan. 1976.

/CHE 88/ CHEN, D-S.; PARNG, T-M. A database management system for a VLSI design system. In: DESIGN AUTOMATION CONFERENCE, 25., Anaheim, June 12-15, 1988. **Proceedings**. New York, IEEE, 1988. p. 257-62

/CHO 88/ CHOU, H. T.; KIM, W. Versions and change notification in a object-oriented database system. In: DESIGN AUTOMATION CONFERENCE, 25., Anaheim, June 12-15, 1988. **Proceedings**. New York, IEEE, 1988. p. 275-81.

/CHU 86/ CHU, K. C. et al. A database-driven VLSI design system. **IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems**, New York, **5**(1):180-7, Jan. 1986.

/COD 70/ CODD, E. F. A relational model of data for large shared data banks. **Communications of the ACM**, New York, **13**(6):377-97, June 1970.

/DAR 87/ DART, S. A. et al. Software development environments. **Computer**, Los Alamitos, **20**(11):18-28, Nov. 1987.

/DAT 83/ DATE, C. J. **Introduction to database systems**. Reading, Addison-Wesley, 1983. v. 2.

/DAT 86/ DATE, C. J. **Introduction to database systems**. Reading, Addison-Wesley, 1986. v. 1.

/DEB 89/ DEBLOCH, S.; HARDER, T.; MATTOS, N.; MITSCHANG, B. **KRISYS: KBMS Support for Better CAD Systems**. Kaiserslautern, University of Kaiserslautern, 1989.

/DER 86/ DERETT, N. P. et al. An object-oriented approach to data management. In: IEEE COMPUTER SOCIETY INTERNATIONAL CONFERENCE - SPRING 86, San Francisco, Mar. 3-6, 1986. **Proceedings**. New York, IEEE, 1986. p.330-5.

/DIT 86a/ DITTRICH, K. R. Object-oriented database systems. In: ENTITY-RELATIONSHIP CONFERENCE, 5., Dijon, 1986. **Proceedings**. Amsterdam, North Holland, 1986. p. 5-13.

/DIT 86b/ DITTRICH, K. R. et al. DAMOKLES - a database system for software engineering environments. In: IFIP WORKSHOP ON ADVANCED PROGRAMMING ENVIRONMENTS, Trondheim, June 16-18, 1986. **Proceedings**. Berlin, Springer Verlag, 1986. p. 353-71.

/DIT 87/ DITTRICH, K. R.; KOTZ, A. M.; MULLE, J. A. Database support for VLSI design: the DAMASCUS system. In: UNGERER, M. H. **CAD-Shinittstellen und datentransferformate im elektronikbereich**. Berlin, Springer Verlag, 1987. p. 62-81.

/FAV 87/ FAVERO, E. L. **A implementação de um núcleo de um gerador de editores dirigidos por sintaxe em um microcomputador**. Porto Alegre, CPGCC da UFRGS, 1987.

/FOR 90/ FORNARI, Miguel. **Implementação de conceitos de abstração no sistema DAMOKLES**. Porto Alegre, Instituto de Informática da UFRGS, 1990.

/GAL 87/ GALLO, F. The object management system of the PCTE as a software engineering database management system. **Sigplan Notices**, New York, **22**(1):12-5, Jan. 1987.

/GOL 83/ GOLDBERG, A.; ROBSON, D. **Smalltalk-80: the language and its implementation**. Reading, Addison-Wesley, 1983.

/GOL 88/ GOLENDZINER, L. G.; BECKER, K. Interface orientada a objetos para um ambiente de CAD de sistemas digitais. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 3., Recife, mar. 23-25, 1988. **Anais**. Recife, SBC, 1988. p. 213-26.

/GOL 89a/ GOLENDZINER, L. G.; WAGNER, F. R.; FREITAS, C. M. D. S. Modeling digital systems in an integrated design environment. In: IFIP WG10.2 INTERNATIONAL SYMPOSIUM ON HARDWARE DESCRIPTION LANGUAGE AND THEIR APPLICATIONS, 9., Washington, June 19-21, 1989. **Proceedings**. Amsterdam, Elsevier Science, 1989. p. 147-56.

/GOL 89b/ GOLENDZINER, L. G.; WAGNER, F. R.; FREITAS, C. M. D. S. Representing digital systems as complex objects. In: DATENBANKSYSTEME IN BURO, TECHNIK UND WISSENSCHAFT, Zürich, Mar. 1-3, 1989. **Proceedings**. Berlin, Springer Verlag, 1989. p. 295-99.

/GUT 77/ GUTTAG, J. Abstract data types and the development of data structures. **Communications of the ACM**, New York, **20**(6):396-404, June 1977.

/HAB 86/ HABERMANN, A. N.; NOTKIN, D. Gandalf: software development environments. **IEEE Transactions on Software Engineering**, New York, **12**(12):1117-27, Dec. 1986.

/HAM 81/ HAMMER, M.; McLEOD, D. Database description with SDM: a semantic data model. **ACM Transactions on Database Systems**, New York, **6**(3):351-86, Mar. 1981.

/HAS 82/ HASKIN, R.; LORIE, R. On extending the function of a relational database system. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, Orlando, June 2-4, 1982. **Proceedings**. New York, ACM, 1982. p. 207-12.

/HAY 88/ HAYNIE, M. N. A database management system for large design automation databases. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, Chicago, June 1-3, 1988. **Proceedings**. New York, ACM, 1988. p. 269-76.

/HOR 86/ HORWITZ, S.; TEITELBAUM, T. Generating editing environments bases on relations and attributes. **ACM Transactions on Programming Languages and Systems**, New York, **8**(4):577-608, Oct. 1986.

/HUD 86/ HUDSON, S. E.; KING, R. CACTIS: a database system for specifying functionally-defined data. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED DATABASE SYSTEMS, 1., Pacific Grove, Sept. 23-26, 1986. **Proceedings**. Washington, IEEE, 1986. p. 26-37.

/HUD 87/ HUDSON, S. E.; KING, R. Object-oriented database support for software engineering environments. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, San Francisco, May 27-29, 1987. **Proceedings**. New York, ACM, 1987. p. 491-503.

/HUD 88/ HUDSON, S. E.; KING, R. The CACTIS project: database support for software environments. **IEEE Transactions on Software Engineering**, New York, **14**(6):709-19, June 1988.

/HUL 87/ HULL, R.; KING, R. Semantic database modeling: survey, applications, and research issues. **Computing Surveys**, New York, **19**(3):201-60, Sept. 1987.

/IOC 89/ IOCHPE, C. **Database recovery in the design environments: requirements analysis and performance evaluation**. Karlsruhe, Universität de Karlsruhe, 1989. Tese de doutorado.

/JOH 75/ JOHNSON, S. C. **YACC - yet another compiler-compiler**. Murray Hill, AT&T Bell Laboratories, 1975. Technical Report 32.

/KAT 82/ KATZ, R. H. A database approach for managing VLSI design data. In: DESIGN AUTOMATION CONFERENCE, 19., Las Vegas, June 14-16, 1982. **Proceedings**. New York, IEEE, 1982. p. 274-82.

/KAT 83/ KATZ, R. H. Managing the chip design database. **Computer**, Los Alamitos, **16**(12):26-36, Dec. 1983.

/KAT 84/ KATZ, R. H.; LEHMAN, T.J. Database support for versions and alternatives of large design files. **IEEE Transactions on Software Engineering**, New York, **10**(2):191-9, Mar. 1984.

/KAT 86a/ KATZ, R. H. Design database. In: **DESIGN Methodologies**. Amsterdam, North Holland, 1986. p. 501-25.

/KAT 86b/ KATZ, R. H.; CHANG, E.; BHATEJA, R. Version modeling concepts for computer-aided design databases. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, Washington, May 28-30, 1986. **Proceedings**. New York, ACM, 1986. p. 379-86.

/KEM 87/ KEMPER, A.; LOCKEMANN, P.C.; WALLRATH, M. An object-oriented database system for engineering applications. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, San Francisco, May 27-29, 1987. **Proceedings**. New York, ACM, 1987. p. 299-310.

/KER 78/ KERNIGHAN, B. W.; RITCHIE, D. W. **The C programming language**. Englewood Cliffs, Prentice-Hall, 1978.

/KET 87/ KETABCHI, A. K.; BERZINS, V. Modeling and managing CAD databases. **Computer**, Los Alamitos, **20**(2):93-102, Feb. 1987.

/KOT 88/ KOTZ, A. M.; DITTRICH, K. R.; MULLE, J. A. Supporting semantic rules by a generalized event/trigger mechanism. In: INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY, Venice, Mar. 14-18, 1988. **Advances in Database Technology**. Berlin, Springer Verlag, 1988. p.76-91.

/KUO 86/ KUO, J. H. et al. Information structuring for software environments. In: WORKSHOP ON ADVANCED PROGRAMMING ENVIRONMENTS, Trondheim, June 16-18, 1986. **Proceedings**. Berlin, Springer Verlag, 1986. p. 97-111.

/LEC 88/ LECLUSE, C.; RICHARD, P.; VELEZ, F. O_2, an object-oriented data model. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, Chicago, June 1-3, 1988. **Proceedings**. New York, ACM, 1988. p. 424-33.

/LES 75/ LESK, M. E. **Lex - a lexical analyzer generator**. Murray Hill, AT&T Bell Laboratories, 1975. Technical Report 39.

/LOR 82/ LORIE, R. A. Issues in databases for design applications. In: IFIP WG5.2 WORKING CONFERENCE ON FILE STRUCTURES AND DATA BASES FOR CAD, Seeheim, Sept. 14-16, 1981. **Proceedings**. Amsterdam, North Holland, 1982. p. 213-22.

/LUZ 90/ LUZZARDI, P. R. G. **LAGO - Uma linguagem de acesso global ao sistema AMPLO**. Porto Alegre, CPGCC da UFRGS, 1990. Trabalho em andamento.

/MAC 89a/ MACHADO, J. C.; PRICE, R. T. Modelagem dos dados de um ambiente de desenvolvimento de software. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 3., Recife, out. 25-27, 1989. **Anais**. Recife, DI-UFPE, 1989. p. 296-310.

/MAC 89b/ MACHADO, J. C. **Conceitos de abstração em bancos de dados para ambientes de projeto**. Porto Alegre, CPGCC da UFRGS, 1989. (TI-137).

/MAT 88/ MATTOS, N. Abstraction concepts: the basis for data and knowledge modeling. In: INTERNATIONAL CONFERENCE ON ENTITY-RELATIONSHIP APPROACH, 7., Roma, Nov. 1988. **Proceedings**. Amsterdam, North Holland, 1982. p. 331-50.

/MAT 89/ MATTOS, N. **An approach to knowledge base management**. Kaiserslautern, Universität Kaiserslautern, 1989. Tese de doutorado.

/MCL 81/ McLEOD, D. Abstraction in databases. **Sigplan Notices**, New York, **16**(1):19-25, Jan. 1981.

/MEL 88/ MELO, W. L. M.; PRICE, R. T. Implementação de um editor de diagramas generalizado. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 2., Canela, out. 27-28, 1988. **Anais**. Porto Alegre, DI-UFRGS, 1988. p. 123-8.

/MEL 89/ MELO, W. L. M. **Uma proposta de um editor diagramático generalizado**. Porto Alegre, CPGCC da UFRGS, 1989. Dissertação de mestrado.

/MEY 81/ MEYER, B. A three-level approach to the description of data structures, and notational framework. **Sigplan Notices**, New York, **16**(1):164-6, Jan. 1981.

/MUL 87/ MULLE, J. A.; DITTRICH, K. R.; KOTZ, A. M. Design management support by advanced database facilities. In: IFIP WG 10.2 WORKSHOP ON TOOL INTEGRATION AND DESIGN ENVIRONMENT, Paderborn, Nov. 26-27, 1987. **Proceedings**. Amsterdam, North Holland, 1987. p. 23-50.

/NES 86/ NESTOR, J. R. Toward a persistent object base. In: WORKSHOP ON ADVANCED PROGRAMMING ENVIRONMENTS, Trondheim, June 16-18, 1986. **Proceedings**. Berlin, Springer Verlag, 1986. p. 372-94.

/PEC 88/ PECKHAN, J.; MARYANSKI, F. Semantic data models. **Computing Surveys**, New York, **20**(3):153-90, Sept. 1988.

/PEN 88/ PENEDO, M. H.; RIDLLE, W. E. Sotfware engineering environment architectures. **IEEE Transactions on Software Engineering**, New York, **14**(6):689-96, June 1988.

/PEN 89/ PENEDO, M. H.; PLOEDEREDER, E.; THOMAS, I. Object management issues for software engineering environments - workshop report. **Sigplan Notices**, New York, **24**(2):226-34, Feb. 1989.

/PRI 85/ PRICE, R. T. De um editor dirigido por sintaxe a um ambiente para desenvolvimento de software. In: CONGRESSO NACIONAL DE INFORMÁTICA, 18., São Paulo, set. 23-29, 1985. **Anais**. São Paulo, SUCESU, 1985. p. 645-9.

/PRI 88/ PRICE, R. T.; FAVERO, E. L. Editores diagramáticos baseados em formalismos gramaticais. In: JORNADA ARGENTINA DE INFORMÁTICA, 17., Buenos Aires, Set. 26-30, 1988. **Anais**. Buenos Aires, SADIO, 1988. p. 509-27.

/PRI 89/ PRICE, R. T.; FAVERO, E. L. Um editor híbrido (texto e diagramas) orientado por estruturas (do tipo grafo e árvore). In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 3., Recife, out. 25-27, 1989. **Anais**. Recife, DI-UFPE, 1989. p. 137-51.

/REG 90/ REGO, A. M. **Uma linguagem gráfica de definção de dados para uma modelo ER estendido**. Porto Alegre, CPGCC da UFRGS, 1990. Dissertação de mestrado.

/REH 88/ REHM, S. et al. Support for design process in a structuraly object-oriented database system. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED DATABASE SYSTEMS, 2., Bad Munster, Sept. 27-30, 1988. **Proceedings**. Berlin, Springer Verlag, 1988. p. 80-97.

/ROS 89/ ROSENBLATT, W. R.; WILEDEN, J. C.; WOLF, A. L. OROS: toward a type model for software development environments. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGE AND APPLICATIONS, 4., Oct. 1-6, 1989. **Proceedings**. New York, ACM, 1989. p. 297-304.

/RUD 86/ RUDMIK, A. Choosing a environment data model. In: WORKSHOP ON ADVANCED PROGRAMMING ENVIRONMENTS, Trondheim, June 16-18, 1986. **Proceedings**. Berlin, Springer Verlag, 1986. p. 395-404.

/SID 80/ SIDLE, T. W. Weakness of commercial data base management systems in engineering applications. In: DESIGN AUTOMATION CONFERENCE, Minneapolis, June 23-26, 1980. **Proceedings**. New York, IEEE, 1980. p. 57-61.

/SMI 77/ SMITH, J. M.; SMITH, D. P. C. Database abstractions: aggregation and generalization. **ACM Transactions on Database Systems**, New York, **2**(2):105-33, June 1987.

/YAN 88/ YANG, Y. K. An enhanced data model for CAD/CAM database system. In: DESIGN AUTOMATION CONFERENCE, 25., Anaheim, June 12-15, 1988. **Proceedings**. New York, IEEE, 1988. p. 263-9.

/WAG 87a/ WAGNER, F. R.; FREITAS, C. M. D. S.; GOLENDZINER, L. G. A digital system design methodology based on nets of agencies. In: IFIP WG 10.2 INTERNATIONAL CONFERENCE ON COMPUTER HARDWARE DESCRIPTION LANGUAGES AND THEIR APPLICATIONS, 8., Amsterdam, Apr. 27-29, 1987. **Proceedings**. Amsterdam, North Holland, 1987. p. 213-24.

/WAG 87b/ WAGNER, F. R.; FREITAS, C. M. D. S.; GOLENDZINER, L. G. The AMPLO system - an integrated environment for digital systems design. In: IFIP WG 10.2 WORKSHOP ON TOOL INTEGRATION AND DESIGN ENVIRONMENT, Paderborn, Nov. 26-27, 1987. **Proceedings**. Amsterdam, North Holland, 1987. p.221-32.

/WAG 89/ WAGNER, F. R. Ambiente integrado para a simulação de sistemas digitais. In: SIMPÓSIO BRASILEIRO DE CONCEPÇÃO DE CIRCUITOS INTEGRADOS, 4., Rio de Janeiro, abr. 12-14, 1989. **Anais**. Rio de Janeiro, SBC, 1989. p. 74-82.

/WOL 86/ WOLF, W. An object-oriented, procedural, database for VLSI chip planning. In: DESIGN AUTOMATION CONFERENCE, 23., Las Vegas, June 29-July 2, 1986. **Proceedings**. New York, IEEE, 1986. p. 744-51.

/WOL 88/ WOLF, W.; LEUKEN, T. G. R. Object type oriented data modeling for VLSI data management. In: DESIGN AUTOMATION CONFERENCE, 25., Anaheim, June 12-15, 1988. **Proceedings**. New York, IEEE, 1988. p. 351-56.

/ZDO 84/ ZDONIK, Stanley B. Object management system concepts. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, Boston, June 13-21, 1984. **Proceedings**. New York, ACM, 1984. p. 13-19.

/ZDO 86/ ZDONIK, Stanley B. Version management in an object-oriented database. In: WORKSHOP ON ADVANCED PROGRAMMING ENVIRONMENTS, Trondheim, June 16-18, 1986. **Proceedings**. Berlin, Springer Verlag, 1986. p. 405-22.

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

Um modelo de dados para ambientes de projeto

Dissertação apresentada aos Srs.

Profa. Lia Goldstein Golendziner

Prof. Dr. Clesio Saraiva dos Santos

Prof. Dr. José Palazzo Moreira de Oliveira

Prof. Dr. Geovane Magalhães

Visto e permitida a impressão

Porto Alegre, 14/12/90

Profa. Lia Goldstein Golendziner
Orientador

Prof. Dr. Ricardo Augusto da L. Reis
Coordenador do Curso de Pós-Graduação
em Ciência da Computação