Projeto Conceitual e Lógico de Sistemas de Gerência de Banco de Dados Orientado a Objetos

Cláudio N. F. Trotta, Cláudia M. L. Werner

UFRJ/COPPE Sistemas
Cidade Universitária - CT - sala H-319
Caixa Postal 68511 - CEP 21945 - Rio de Janeiro - RJ
Tel. (021) 590-2552/2652

Sumário

Neste trabalho fazemos uma análise do emprego do Modelo de Entidades e Relacionamentos como ferramenta conceitual para a fase anterior ao projeto lógico de um banco de dados orientado a objetos. Abordamos a mudança de perspectiva necessária para que possamos utilizá-lo com sucesso neste novo contexto, inclusive introduzindo a abstração conceitual de especialização de relacionamento. Apresentamos, ainda, considerações para o projeto lógico, específicas ao SGBDOO utilizado.

1. Introdução.

O Modelo de Entidades e Relacionamentos (MER) [CHEN 76] foi concebido, e tem sido utilizado, como ferramenta conceitual para o projeto de banco de dados relacional. Na tentativa de utilizar esta ferramenta para reprojetar uma aplicação, anteriormente implementada no modelo relacional, seguindo o paradigma da orientação a objetos, notamos que o esquema lógico final se distanciou muito do Diagrama de Entidades e Relacionamentos (DER) obtido anteriormente. Este artigo tem como objetivo apontar as razões pelas quais isto ocorre, oferecendo recomendações para o uso do MER neste contexto.

Outra questão considerada é o processo do projeto lógico, isto é, a passagem do DER para o esquema de classes de um Sistema de Gerência de Banco de Dados Orientado a Objetos (SGBDOO). Neste processo, como ainda não existe uma definição consensual sobre os requisitos de um SGBDOO, soluções específicas foram tomadas para o sistema usado, no caso, o O2.

O O2 é um SGBDOO em desenvolvimento num projeto de cinco anos do consórcio Altair, França, iniciado em 1986. O sistema provê um ambiente multi-língue de programação (ex. CO2, BasicO2), uma linguagem de consulta, uma interface alfanumérica, um ambiente de programação (OOPE) e um conjunto de ferramentas geradoras de interface (LOOKS) [DEUX 89]. Ao interagir com o O2, o usuário pode escolher entre utilizar a interface alfanumérica ou o ambiente OOPE, sendo esta última a interface mais usada, pois oferece ferramentas gráficas para atualização, consulta e navegação na base de dados. O sistema utilizado é o protótipo, versão 1.1, distribuido para avaliação, que roda em equipamentos Sun 3/60, 3/80 ou 3/280S, com sistema operacional SunOS versão 4.0.

Na próxima seção descrevemos o projeto conceitual da aplicação. Na seção três apresentamos o modelo de dados do O2, com o objetivo de fornecer informações necessárias para o projeto lógico, descrito na seção quatro.

2. A Modelagem da Aplicação.

A aplicação reprojetada é um protótipo de um dicionário de dados automatizado, uma das ferramentas encontradas num ambiente do tipo **FEGRES** [TRAVASSOS 90]. Apesar desta aplicação já ter sido, previamente, modelada em [TROTTA 90a, 90b] e, efetivamente, construída usando-se um banco de dados relacional, o enfoque da orientação a objetos, agora abordado, nos levou a um esquema conceitual diferente do anterior.

Esta aparente contradição é explicada, se abandonarmos a visão ingênua de que o poder de expressão do método não determina, de forma direta, o esquema conceitual a ser produzido. Anteriormente, utilizou-se o MER como ferramenta conceitual e o Modelo Relacional como ferramenta para a implementação. Agora, embora utilizemos o MER como uma ferramenta gráfica de comunicação e modelagem, o paradigma da orientação a objetos dita as abstrações conceituais que serão reconhecidas no processo de modelagem e o modelo de dados específico do O2 especializa mais ainda estas abstrações. Desta forma, embora os construtores básicos das abstrações conceituais do DER, construído anteriormente [TROTTA 90b], sejam os mesmos do DER agora apresentado (figura 1), o esquema conceitual produzido é diferente, graças a esta alteração da perspectiva. O diagrama apresentados na figura 1 segue a convenção do MER extendido apresentada em [ELMASRI 89], com exceção da abstração de **especialização de relacionamento** que será apresentada na seção quatro. O esquema completo do O2 correspondente à figura 1 encontra-se no apêndice.

Podemos dizer que as diferenças entre os dois esquemas são causadas pelos seguintes fatores:

1) O relativamente forte poder de expressão do MER na parte estrutural em comparação com a capacidade de representar operações (nenhuma) e restrições de integridade (limitada);

2) A eliminação do conceito de chave primária.

A seguir discutimos cada um destes fatores.

2.1.Poder de expressão do MER.

No MER (quando utilizado como ferramenta conceitual para o modelo relacional) diversas restrições de integridade são capturadas, criando-se entidades e relacionamentos adicionais. Isto é, como o modelo é "forte" na parte estrutural, as restrições de integridade são, algumas vezes, "transformadas" em estruturas de dados não

naturais. É o exemplo de algumas categorias criadas no diagrama anterior (Rep_Nó, Rep_Construtor e Construtor) [TROTTA 90b]. As restrições de integridade que estas estruturas representam são, detalhadamente, explicadas no mesmo trabalho.

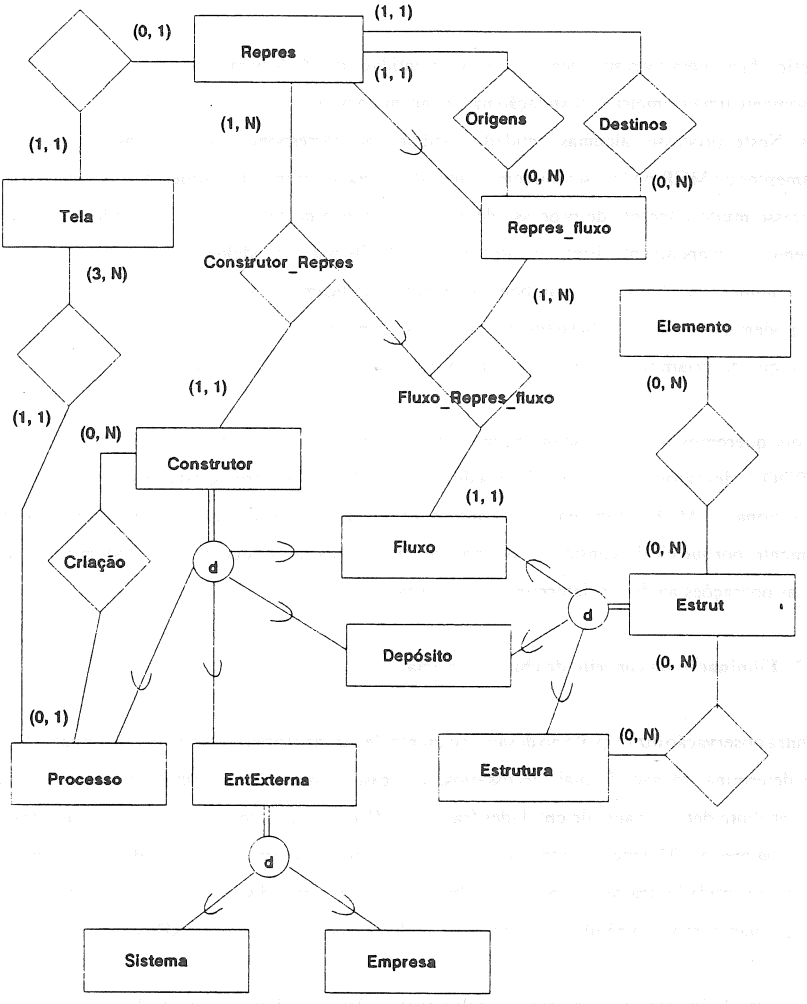

Figura 1 - O Esquema para o O2.

Ao passar para uma representação orientada a objetos, é possível modelar também o comportamento. Este aspecto, normalmente, desprezado totalmente no MER, pode levar ao aparecimento de hierarquias de generalização/especialização criadas somente com o intuito de fatorar operações. Além disto, as restrições de

integridade podem, também, ser incorporadas às especificações das operações. A combinação destes dois fatores, isto é, a criação de hierarquias de generalização/especialização para fatorar operações que mantêm a integridade, foi um ponto muito importante nesta aplicação específica (vide 4.2). Estes dois fatores influiram bastante no processo de modelagem.

Assim, fazer uma correspondência 1:1 de uma entidade do MER para uma classe na orientação a objetos, deve ser somente uma primeira aproximação na passagem do projeto conceitual para o projeto lógico orientado a objetos. Neste processo, algumas entidades podem não corresponder a uma classe e vice-versa, e os relacionamentos no MER podem ser implementados de formas diferentes no esquema orientado a objetos. Esse é um processo muito diferente do processo de projeto de banco de dados relacional, onde o projeto lógico é, praticamente, um mapeamento direto das abstrações do MER para o modelo relacional. No caso da orientação a objetos, o número de alterações e, também, sua natureza indicam novas abstrações (ex. duas entidades, antes distintas, podem passar a ser subclasses de uma classe comum, devido ao interesse em, também, fatorar as operações) que deveriam ter sido detectadas no projeto conceitual e não no lógico.

O que queremos dizer é que ao utilizarmos o MER para representar uma aplicação que será implementada num SGBDOO, devemos abordar o problema diferentemente do que seria se esta fosse implementada um em SGBD relacional. O MER, assim, não é de um nível conceitual tão elevado quanto gostaríamos. A razão disto é, basicamente, porque ele desconsidera, sistematicamente, as operações que os dados podem sofrer. Em outras palavras: as operações ajudam a determinar a estrutura dos dados.

2.2. Eliminação do conceito de chave primária.

Outra observação diz respeito ao desaparecimento de alguns atributos. No nível conceitual (MER), alguns atributos determinantes não são mais necessários. É o caso, também, de atributos não naturais criados para compor o atributo determinante de entidades fracas do MER. Isto acontece pois estas entidades fracas podem passar a ser *valores* no O2 (vide seção três) da entidade forte que as determinam, ou ainda, mesmo que continuem existindo como entidades fracas, o conceito de objeto é suficiente para distinguir suas instâncias. Isto vem sendo discutido amplamente sob o rótulo de "problema de identidade do objeto" [ATKINSON 89].

No nível da implementação (modelo relacional), alguns atributos de controle (como, por exemplo, a identificação da última instância de representação gráfica criada, necessária para compor a chave da próxima instância a ser criada) também se tornam desnecessários, pois objetos podem ser identificados independente destes atributos.

3. O Modelo de Dados do O2

No modelo de dados do O2 existe uma distinção entre os chamados *valores* e *objetos*. Objetos em O2, como em linguagens de programação orientadas a objetos (OOPL) (ex. Smalltalk, C++, etc), possuem uma identidade e encapsulam dados e comportamento (i.e. métodos). Valores, entretanto, podem ser vistos como dados, simplesmente, não obedecendo aos princípios da orientação a objetos da seguinte forma [LÉCLUSE 89a]:

- não possuem identidade;
- sua estrutura, embora similar a de objetos, é visível e manipulável;
- a manipulação é feita através de operadores primitivos, não por métodos.

Valores foram introduzidos em O2 para permitir a manipulação de estruturas complexas de dados, não existentes em OOPLs. Eles podem ser do tipo *atômico* (ex. inteiro, real, booleano, etc) ou *estruturado* (ex. tuple, set, list).

Objetos pertencem a *classes*. Classes descrevem a estrutura e o comportamento de uma coleção de objetos. A parte estrutural é um *tipo,* e a parte comportamental é um conjunto de *métodos* . A definição de uma classe consiste em prover : seu nome, seu tipo e seus métodos. É possível definir, neste instante, somente os nomes dos métodos para, posteriormente, definirmos seu código.

Todas as classes em O2 são subclasses diretas ou indiretas de uma classe pré-definida, denominada OBJECT. Um esquema em O2 é um conjunto de classes ligadas pela relação de herança, onde herança múltipla é permitida desde que não haja conflitos de nomes.

Um objeto é uma *instância* de uma classe e pode ser visto como um par *(identificador,valor)*. Como uma classe em O2, por "default", não é persistente, um objeto só será persistente se for referenciado por objetos persistentes ou nomeados. Para que os objetos de uma classe sejam persistentes é preciso definí-la com a cláusula **with extension**, explicitamente. Na versão do protótipo usado, entretanto, esta cláusula não está implementada, tendo sido necessário criarmos valores nomeados para os "sets" de objetos das classes que deveriam ser persistentes.

Métodos são escritos utilizando extensões de linguagens de programação tradicionais (ex. CO2, BasicO2), sendo que o protótipo usado oferece apenas a linguagem CO2. A definição de um método consiste em fornecer sua assinatura (i.e., seu nome, parâmetros, tipos ou classes dos parâmetros e tipo ou classe de seu resultado) e seu corpo (i.e., código em co2). Métodos podem, ainda, ser *públicos* (podem ser chamados a partir de quaisquer classes) ou *privados* (só podem ser chamados por métodos na mesma classe).

Uma estrutura de dados de um classe pode ser *privada* ou *pública*. A estrutura pública oferece um

relaxamento no encapsulamento dos dados, permitindo que a estrutura seja acessada por métodos de outras classes.

4. O Projeto da Aplicação.

No projeto lógico da aplicação tivemos que considerar o paradigma da orientação a objetos com as limitações específicas do SGBD usado, no caso o O2. O maior problema encontrado foi como representar relacionamentos pois não há uma representação direta para esta abstração. Além disto, existe uma certa assimetria na parte estrutural e na linguagem de consultas que causaram algumas dificuldades adicionais. A seguir descrevemos, detalhadamente, este problema e duas formas encontradas para contorná-lo.

4.1. Assimetria do modelo de dado do O2.

A assimetria do O2 em representar relacionamentos, combinada com a forma de expressar consultas, levam a formas alternativas de modelagem. Exemplificando, um relacionamento de cardinalidade 1:N, entre objetos de duas classes A e B, pode ser representado:

i) "à moda relacional", isto é, colocando-se um ponteiro para a classe "do lado do 1", na classe "do lado do N":

```
add class A
type tuple (.....);

add class B
type tuple (.....,
            a: A,
            ....);
```

ii) usando-se o construtor de set:

```
add class A
type tuple (....,
            b: set(B),
            ...);

add class B
type tuple (.....);
```

iii) ou ainda, usando-se uma combinação de i e ii, obtendo-se uma representação simétrica, mas com controle de integridade sob responsabilidade do usuário/programador.

```
add class A
type tuple(.....,
           b: set(B),
           ...);

add class B
type tuple (.....,
            a: A,
            ....);
```

A decisão sobre a alternativa preferível pode ser tomada em função da facilidade em programar os métodos em cada uma delas. É interessante notar que esta percepção só acontece num nível de detalhamento mais profundo, isto é, numa etapa muito posterior a que se deseja, ao projetar um esquema conceitual.

Para se verificar as implicações negativas de uma modelagem assimétrica, agravadas pelas construções permitidas na linguagem de consulta do O2, considere o seguinte exemplo:

```
add class Aluno
type tuple (nome:string,
            endereço:string,
            matrícula:set(Matrícula));

add class Matrícula
type tuple(nota:integer);

add class Curso
type tuple (código:integer,
            título:string,
            matrícula:set(Matrícula);
```

As classes acima representam um relacionamento N:M, de nome "Matrícula", entre as entidades *Aluno* e *Curso*. Como no projeto relacional, escolhemos representar este relacionamento como uma classe de nome *Matrícula*. Outras opções de modelagem seriam possíveis, por exemplo, colocando a nota diretamente nas classes *Aluno* e/ou *Curso*.

A simples consulta "listar os nomes dos alunos que cursam a disciplina cujo título é Cálculo", que em SQL seria, simplesmente:

```
Aluno   (nome,endereço)
Matrícula  (nome,código,nota)
Curso (código,título)

select  a.nome
from    a in Aluno, m in Matrícula, c in Curso
where  a.nome = m.nome and c.código = m.código
         and c.título = "Cálculo";
```

No O2 ficaria:

```
select   a.nome
from     a in Aluno,
         m in a.matrícula
where    m is in   (  select m1
                      from   c in Curso,
                             m1 in c.matrícula
                      where c.título = "Cálculo");
```

A dificuldade é causada pela inexistência de uma operação equivalente a operação relacional de junção Se a classe *Matrícula* for definida de forma a tornar o modelo simétrico, teríamos:

```
add class Matrícula
type tuple(nota:integer,
           aluno:set(Aluno),
           curso:set(Curso));

select   a.nome
from     a in Aluno,
         m in a.matrícula,
         c in m.curso
where  c.título = "Cálculo";
```

ou

```
select   a.nome
from     c in curso,
         m in c.matrícula,
          a in m.aluno
where    c.título = "Cálculo";
```

Embora a estrutura de dados tenha se tornado simétrica, as consultas ainda lembram uma navegação hierárquica no sentido *aluno-matrícula-curso* ou *curso-matrícula-aluno*, respectivamente.

4.2. Resolvendo a assimetria na aplicação.

Na aplicação do Dicionário de Dados escolhemos testar duas formas de resolver o problema da assimetria na modelagem de relacionamentos entre classes:

i) Mantendo simétrica a estrutura que representa os relacionamentos e controlando as restrições de integridade decorrentes da redundância introduzida; ou

ii) Representando a estrutura assimetricamente e enfrentando o problema da assimetria da linguagem de consulta.

No projeto lógico, mantivemos a representação simétrica em todos os relacionamentos, exceto em dois deles, escolhidos aleatoriamente.

4.2.1. Projeto Simétrico.

No primeiro caso, as restrições de integridade foram testadas valendo-se de operações de alto nível disponíveis na linguagem CO2. Vejamos o caso do relacionamento 1:N *Criação* (vide figura 1). Este relacionamento é representado pelos atributos *criador* da classe *Construtor* e *filhos* da classe *Processo*.

```
add class  Construtor
        inherits object
        type tuple (criador:Processo,
                    rep: set (Repres));

add class Processo
        inherits Construtor
        type tuple (ident_p:string;
                    nome:string,....
                    filhos:set(Construtor));

add name Processos: set (Processo);
```

A última linha cria um valor nomeado *Processos*. Todo objeto da classe *Processo* que for inserido neste "set" será persistente.

A' interface LOOKS [O2 89c], fornecida com o sistema, permite manipular os objetos na tela sem necessidade de programação. Por exemplo, para colocarmos um *Processo* como o criador de um *Construtor*, basta arrastarmos pela tela o ícone que representa o *Processo*, parando em cima da caixa de diálogo que representa o atributo *criador* (do *Construtor*).

Se o usuário realizar tal movimento, torna-se necessário manter a integridade da base de dados, que foi ferida, pois os dados devem ser simétricos. Em outras palavras, se o processo P1 foi arrastado para ser o criador do fluxo F1, é necessário incluir o fluxo F1 no conjunto de filhos de P1.

Um método de nome *integr_criador* da classe *Construtor* é responsável por esta restrição:

```
add method integr_criador in class Construtor
body integr_criador (paiant:Processo):object in co2
{
if (self->criador != paiant)
        {
        if (paiant != nil)
           {paiant->filhos -= set (self);};
        if (self->criador != nil)
           {self->criador->filhos += set (self);};
        };
}
```

A variável *paiant* é passada como parâmetro e representa o valor anterior do atributo *criador* da classe *Construtor*. Se houver mudança na paternidade, o objeto em questão é incluído (ou retirado) do conjunto de filhos do seu novo pai (antigo pai).

Por outro lado, ao alterar-se um processo, é necessário, também, testar a variação no seu conjunto de filhos. Isto ocorre no caso em que o fluxo F1 teria sido arrastado para dentro do conjunto de filhos de P1. O seguinte trecho de programa, no método que modifica um processo, é responsável por isto:

```
O2 set(Construtor) filhosant, filhosnovos,filhosdeletados;
O2 Construtor p;
O2 Processo paiant;

filhosant = self->filhos;
paiant    = self->criador;

/* -- modificação dos dados do processo */

filhosnovos = self->filhos - filhosant;
filhosdeletados = filhosant - self->filhos;

for (p in filhosnovos) {p->criador = self};
for (p in filhosdeletados) {p->criador = nil};

[self integr_criador (paiant)];
```

Repare na operação de diferença de conjuntos e ao envio da mensagem *integr_criador* na última linha. A razão disto é porque *Processo* também **é-um** *Construtor* (veja no esquema).

4.2.2. Projeto Assimétrico.

Para a segunda solução, consideramos três relacionamentos representados assimetricamente no esquema lógico (refira-se à figura 1 e ao esquema no apêndice): *Fluxo_Repres_fluxo*; *Origens* e *Destinos*.

A entidade *Repres* significa uma representação gráfica no DFD de um *Construtor* (i.e., processo, depósito ou entidade externa), enquanto *Repres_fluxo* (especialização de *Repres*) representa uma representação gráfica no DFD de um fluxo de dados. Assim, o relacionamento *Fluxo_Repres_fluxo* é uma ***especialização do relacionamento*** *Construtor_Repres*. A figura 1 mostra uma representação gráfica para esta abstração. O significado disto é que uma especialização da entidade *Construtor* (a entidade *Fluxo*) não participa do relacionamento *Construtor_Repres*, mas sim do relacionamento *Fluxo_Repres_fluxo*.

Para a abstração de especialização de relacionamento fazer sentido, é necessário que aconteça entre relacionamentos cujas entidades participantes são especializações entre si, isto é, se o relacionamento

Fluxo_Repres_fluxo acontece entre as entidades *Fluxo* e *Repres_fluxo* e, ainda, ele é uma especialização do relacionamento *Construtor_Repres*, que acontece entre *Construtor* e *Repres*, então *Fluxo* é uma especialização de *Construtor* e *Repres_fluxo* é uma especialização de *Repres*.

Os relacionamentos *Origens* e *Destinos* do DER representam as ligações gráficas, isto é, contêm a informação de quais fluxos ligam quais construtores.

A definição do esquema é:

```
add class Fluxo
        inherits Construtor, Estrut
        type tuple (redefines rep:set (Repres_fluxo),
                    nome_fluxo:string,
                    descrição:string,..... );

add name Fluxos: set(Fluxo);

add class Repres
        inherits object
        type tuple (x:integer,
                    y:integer,
                    tela:Tela,
                   construtor:Construtor);

add class Repres_fluxo
        inherits Repres
        type tuple (tipo_traçado:integer,
                    origem: tuple (o:Repres,
                                   xo:integer,
                                   yo:integer),
                    destino: tuple (d:Repres,
                                    xd:integer,
                                    yd:integer));
```

Repare que na classe *Repres_fluxo* não existe um atributo para representar o relacionamento com a classe *Fluxo*, nem na classe *Repres* sequer existe um atributo para representar o relacionamento com *Repres_fluxo*.

É muito fácil navegar na direção dos relacionamentos. Por exemplo, com uma ficha de fluxo de dados na tela, podemos selecionar uma de suas representações (instância de *Repres_fluxo)* e, acionando o "mouse", teremos a opção (que aparece num cardápio) de consultar o construtor (i.e., processo, depósito ou entidade externa) donde se origina aquela representação. Como a navegação é no sentido *Repres_fluxo - Repres - Construtor*, e existem atributos representando os relacionamentos nesta mesma direção, o código em CO2 fica:

```
add method origens in class Repres_fluxo
body origens():object in co2
{...
self->origem.o->construtor}
```

Graças ao "desencapsulamento" permitido no modelo de dados, é possível atingir diretamente um *Construtor* a partir de um objeto da classe *Repres_fluxo* (self) navegando através dos atributos.

Porém, a recíproca, isto é, a partir da classe *Repres* atingir a classe *Fluxo* (por exemplo, dado um processo, queremos recuperar os fluxos que têm origem nele), é mais difícil de implementar em CO2. Seu código é:

```
add method origens in class Repres
body origens():object in co2

O2 Fluxo f;
O2 set(Fluxo) fs;
O2 Repres_fluxo w;

for (f in Fluxos)
      {aux = select w->origem.o
             from w in f->rep
             where w-origem.o == self end;

       if (aux != (O2 set (Repres)) set () )
            {fs += set(f);};
      };
/* fs é o conjunto de todos os fluxos que tem origem no construtor representado por self */
```

Na verdade, a linguagem de consulta permite expressar esta consulta num nível muito mais alto, porém, na atual versão do protótipo, somente um subconjunto desta linguagem pode ser embutida em CO2. Na linguagem de consulta , teríamos:

```
fs = select f
     from f in Fluxos
      where exists x in
                    (select w->origem.o
                    from w in f->rep): (x == self) end;
```

Dentro da concepção do modelo de dados do O2, onde valores devem ser usados para representar informação que não é compartilhada [O2 89a], as classes *Repres* e *Repres_fluxo* poderiam ter sido assim representadas. Porém, é importante que sejam, realmente, representadas como classes pois suas instâncias são manipuladas como objetos. Algumas das operações destes objetos são: movimentação na tela do DFD, consulta às origens e destinos, alteração de tamanho, etc. Isto reforça a idéia de que as operações ajudam a determinar a estrutura dos dados, conforme discutido em 2.1.

5. Conclusões.

Neste trabalho, indicamos que o maior problema na aplicação do MER, para o projeto lógico orientado a objetos, diz respeito a sua incapacidade de representar operações. Como vimos, as operações ajudam a determinar a estrutura dos dados. O problema por trás da questão é um velho costume de encarar os dados independente das operações e restrições inerentes a ele. Modelar um conceito do mundo real como um atributo ou como uma entidade pode ser determinado, simplemente, verificando-se a existência de operações específicas sobre este conceito. Em caso afirmativo, ele merece um "status" de entidade. Outro exemplo no qual observamos que as operações podem determinar estruturas de dados são as hierarquias de generalização/especialização que surgem com o objetivo único de fatorar operações.

Em termos de projeto lógico orientado a objetos, ainda existe uma grande dependência do SGBDOO utilizado. No caso do O2, a falta de representação direta de relacionamentos causou dificuldades que tiveram que ser contornadas programando-se métodos para implementar restrições de integridade ou convivendo-se com o problema da assimetria da linguagem de consultas. Outros modelos de objetos [MONTE 91] permitem uma representação mais natural.

O sistema O2, principalmente as facilidades do LOOKS, mostrou-se bastante satisfatório para o tipo de aplicação implementada. A maioria do código escrito correspondeu às restrições de integridade necessárias para manter os relacionamentos simétricos, o que demonstra a necessidade da incorporação de algumas abstrações dos modelos semânticos de dados em Sistemas de Gerência de Banco de Dados Orientados a Objetos.

Referências Bibliográficas.

Manuais do O2:

[O2 90] "The O2 System User's Guide"
Version 1.1, Released 15 February 1990. Printing revision 1.3, 26 April 1990.

[O2 89a] "The O2 Programmer's Manual"
Version 1.0, Released 15 December 1989. Printing revision 1.2, 9 January 1990.

[O2 89b] "The CO2 Programmer's Manual"
Version 1.0, Released 15 December 1989. Printing revision 1.2, 9 January 1990.

[O2 89c] "The LOOKS User's Manual"
Version 1.0, Released 15 December 1989. Printing revision 1.1, 9 January 1990.

[O2 89d] "The LOOKS Programmer's Manual"
Version 1.0, Released 15 December 1989. Printing revision 1.1, 9 January 1990.

[O2 89e] "OOPE: The Object-Oriented Programming Environment"
Version 1.0, Released 15 December 1989. Printing revision 1.1, 9 January 1990.

[O2 89f] "The O2 Query Language User's Manual"
Version 1.0, Released 15 December 1989. Printing revision 1.1, 9 January 1990.

Relatórios Técnicos:

[ATKINSON 89] M. Atkinson et al., "The Object-Oriented Database System Manifesto", Rapport Technique Altair 30-89.

[BANCILHON 88] François Bancilhon, "Object-Oriented Database Systems", Rapport Technique Altair 16-88.
[DEUX 89] O. Deaux at al., "The Story of O2", Rapport Technique Altair 37-89.
[LÉCLUSE 89a] C. Lécluse, P. Richard, "The O2 Data Model", Rapport Technique Altair 39-89.
[LÉCLUSE 89b] C. Lécluse, P. Richard, "The O2 DataBase Programming Language", Rapport Technique Altair 26-89.
[MONTE 91] L.C.M. Monte, "Uma Representação Gráfica para o Modelo de Objetos do GEOTABA", Relatório Técnico do Programa de Eng. de Sistemas e Computação ES-236/91, COPPE/UFRJ, Março 1991.

Artigos:

[CHEN 76] P.P. Chen, "The Entity-Relationship Model - Towards a Unified View of Data", ACM Transactions Data Base Systems, vol.1, no.1, March 1976.
[ELMASRI 89] R.Elmasri, S.B.Navathe, "Fundamentals of Database Systems", The Benjamin/Cummings Publishing Company, Inc. 1989.
[TRAVASSOS 90] G. H. Travassos, "FEGRES: FErramentas GRáficas para Engenharia de Software", Tese de Mestrado do Programa de Engenharia de Sistemas COPPE/UFRJ, 1990.
[TROTTA 90a] C..N.F. Trotta, G.H.Travassos, "Modelagem de Dados no Projeto de um Ambiente de Desenvolvimento de Software", anais da SUCESU 1990.
[TROTTA 90b] C.N.F. Trotta et al., "Modelagem de Dados de Aplicações não Convencionais: Um Estudo de Caso", anais do 4 Simpósio Brasileiro de Engenharia de Software, outubro 1990.

Apêndice

Esquema do Banco de Objetos O2

```
add class Construtor
/*      with extension*/
        inherits object
        type tuple(criador:Processo,
                   rep: set(Repres));

public * in class Construtor;

add name Construtores : set(Construtor);

add class Repres
        inherits object
        type tuple (x:integer,
                    y:integer,
                    tela: Tela,
                    construtor:Construtor);

public * in class Repres;

add class Repres_fluxo
        inherits Repres
        type tuple (tipo_tracado:integer,
                    origem:  tuple(o:Repres,
                                   xo:integer,
                                   yo:integer),
                    destino: tuple(d:Repres,
                                   xd:integer,
                                   yd:integer));

public * in class Repres_fluxo;

add class Processo
/*      with extension*/
        inherits Construtor
        type tuple (ident_p:string,
                    nome:string,
                    descricao:string,
                    resumo_logico:string,
                    refs_fisicas:string,
                    det_logica:string,
                    filhos:set(Construtor),
                    tela: Tela);

public * in class Processo;

add name Processos: set (Processo);

add class Estrut
        inherits object
        type tuple (elementos: set(Elemento),
                    estruturas: set(Estrutura));

public * in class Estrut;

add class Deposito
/*      with extension*/
        inherits  Construtor,Estrut
        type tuple (ident_d:string,
                    nome:string,
                    descricao:string,
                    acesso_imediato:string,
                    org_fisica:string);
```

```
public * in class Deposito;

add name Depositos: set (Deposito);

add class EntExterna
/*      with extension*/
        inherits Construtor
        type tuple (ident_e:string,
                    nome:string,
                    descricao: string);

public * in class EntExterna;

add name EntExternas: set (EntExterna);

add class Sistema
/*      with extension*/
        inherits EntExterna
        type tuple (linguagem:string,
                    hardware:string,
                    pessoapd:string,
                    fonepd:string);

public * in class Sistema;

add class Empresa
/*      with extension*/
        inherits EntExterna
        type tuple (setor:string,
                    pessoaorg:string,
                    cargo:string,
                    foneor:string);

public * in class Empresa;

add class Tela
/*      with extension*/
        inherits object
        type tuple (x:integer,
                    y:integer,
                    fundo:integer,
                    linhas:integer,
                    processo: Processo,
                    reps: set (Repres));

public * in class Tela;

add name Telas: set (Tela);

add class Elemento
        inherits object
        type tuple (nome_ele:string,
                    descricao:string,
                    tipoelemento:string,
                    dominio:string,
                    vertambem:string,
                    estruts: set(Estrut));

public * in class Elemento;

add name Elementos: set (Elemento);

add class Estrutura
      inherits Estrut
        type tuple(nome_est:string,
                   descricao:string,
                   volume:string,
                   estruts_pai: set(Estrut));

public * in class Estrutura;

add name Estruturas: set (Estrutura);

add class Fluxo
/*      with extension*/
        inherits  Construtor, Estrut
        type tuple (redefines rep: set(Repres_fluxo),
                    nome_fluxo:string,
                    descricao:string,
                    desc_ampliada:string,
                    volume:string,
                    tipo:integer);

public * in class Fluxo;

add name Fluxos: set (Fluxo);
```