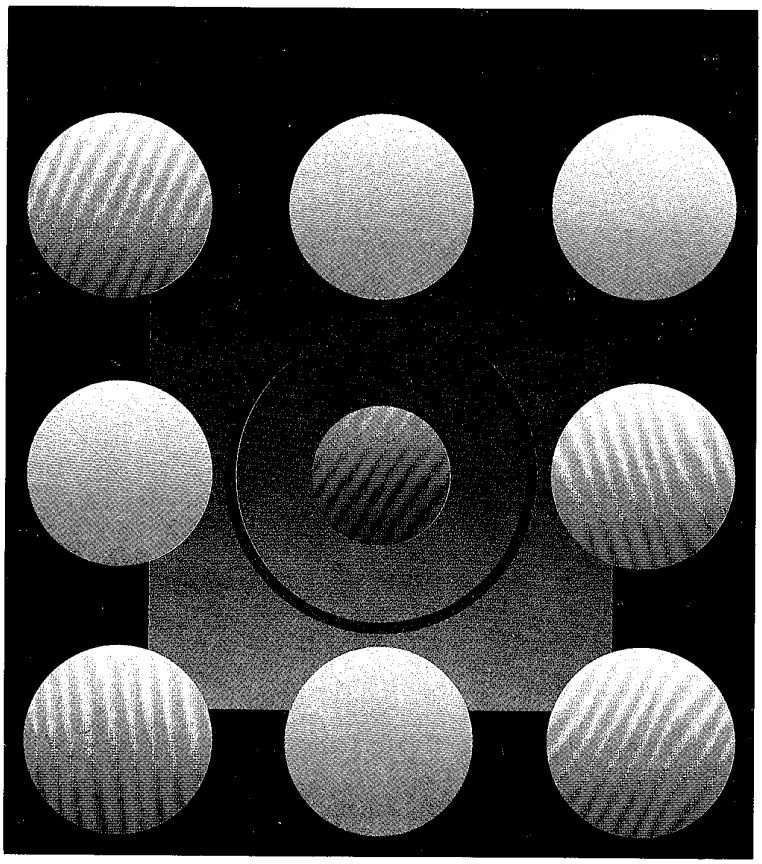
REVISTA
BRASILEIRA
DE COMPUTAÇÃO

RBC

ISSN 0101-0883

Uma publicação da SBC SOCIEDADE BRASILEIRA DE COMPUTAÇÃO

1981, Vol. 1, Nº 3



NESTE NÚMERO

MODELAGEM

LINGUAGENS

BANCO DE DADOS

EDITORA CAMPUS

RBC, Rio de Janeiro, V. 1, nº 3, 1981, pp. 179 a 193

ESPECIFICAÇÕES ABSTRATAS (DE BANCOS DE DADOS) VIA NÍVEIS DE TRAÇO

A. L. FURTADO
T. S. E. MAIBAUM*
P. A. S. VELOSO**
PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO / DEPARTAMENTO DE INFORMÁTICA
RUA MARQUÊS DE SÃO VICENTE, 225
CEP 22453 – RIO DE JANEIRO – RJ – BRASIL

SUMÁRIO

Este trabalho visa ser uma introdução informal a especificações formais para tipos abstratos de dados. Com este fim estende-se o conceito usual de traço a vários níveis, o que fornece uma ferramenta útil para a obtenção de especificações, além de apresentar analogias com idéias familiares simples. A apresentação é baseada em um exemplo simplificado de banco de dados, o qual é sucessivamente especificado em três formatos: não procedural, procedural e algébrico.

ABSTRACT

This paper amounts to an informal introduction to formal specifications for abstract data types. To this end the well-known concept of trace is extended to several levels, which gives a useful tool in obtaining formal specifications, besides having familiar simple analogues. The presentation is based on a simplified example of a data base. This example is successively specified in three formats: non-procedural, procedural and algebraic.

* Atualmente no Dept. of Computing, Imperial College of Science and Technology, Londres, Inglaterra.

** E Programa de Engenharia de Sistemas e Computação, COPPE-UFRJ.

1. INTRODUÇÃO

Quando uma firma decide adotar o enfoque de banco de dados para uma área de aplicação, sua preocupação inicial não deve ser como os dados serão estruturados, mas sim que gêneros de dados serão armazenados e quais os usos previstos para eles. Esta fase de especificação é crucial e, para ela, têm sido propostas certas técnicas formais (algumas discutidas e avaliadas recentemente.[21).Essa preocupação não é exclusiva da área de banco de dados, mas se prende à própria tarefa de programação de modo geral [11]. Aqui nos restringiremos ao enfoque de tipos abstratos de dados, o qual permite que a atenção se concentre inicialmente nas classes abstratas de dados e no seu uso [10]. Outros benefícios deste enfoque são especificações formais adequadas à verificação, teste e documentação [7].

Infelizmente, porém, o estilo da literatura sobre tipos abstratos de dados costuma ser mais dirigido a uma audiência de teóricos. Uma consequência disso é que se perdem oportunidades de aplicar - e sintonizar - essas idéias às necessidades concretas do dia-a-dia da computação [13]. Na área de processamento de dados comercial, por exemplo, até mesmo técnicas novas mais orientadas para a prática não têm encontrado aceitação tão ampla quanto seria de se desejar [6].

Neste trabalho pretendemos ilustrar como algumas noções de tipos abstratos de dados podem ser traduzidas por meio de conceitos bastante simples e familiares, como o de traço (ou rastreio) bastante usado para testar programas. Aqui pretendemos mostrar como traços podem desempenhar o papel de estrutura de dados "universal", servindo de base para especificações formais e executáveis.

A fim de ilustrar os vários conceitos examinados será utilizado um exemplo bem simples, da área de banco de dados, apresentado na seção 2. A seção 3 examina a idéia de traço neste contexto enquanto que a seção 4 introduz a idéia de níveis de traço, ilustrando seu uso em especificação. A seção 5 trata de especificações precisas e executáveis baseadas em traços e a seção 6 relaciona estas com a especificação algébrica. A seção 7 conclui com uma perspectiva e comentários gerais.

2. UM EXEMPLO SIMPLES

Como exemplo, bastante simplificado, para ilustrar a discussão, tomemos o caso de uma companhia que negocia uma máquina, disponível em um único modelo. A companhia opera de modo que um cliente pode

(a) alugar uma máquina ,ou

(b) comprar uma máquina

da companhia. Em ambos os casos o cliente usa a máquina, porém apenas no caso (b) ele a possui. No caso (a), o cliente pode resolver depois comprar a máquina alugada. Tendo alugado, mas não comprado, uma máquina, o cliente pode ainda de-

cidir devolvê-la à companhia. Para simplificar, vamos supor que um cliente nunca poderá ter mais do que uma máquina.

As palavras sublinhadas no parágrafo anterior correspondem às operações de consulta e de atualização, que ainda incluem uma operação início para inicializar o banco de dados a um estado "vazio".

Neste contexto é importante ter-se em mente a idéia de estado ou instância do banco de dados. Assim, podemos encarar um banco de dados como consistindo de vários estados. A transição de um estado a outro é efetuada pela aplicação de uma operação de atualização.

Usando o conceito de estado, podemos descrever melhor, se bem que ainda informalmente, as operações do nosso exemplo, como se segue.

Atualizações:

início - inicializa o banco de dados

aluga(c,s) - cliente c aluga uma máquina

compra(c,s) - cliente c compra uma máquina

devolve(c,s) - cliente c devolve uma máquina

Consultas:

usa(c,s) - cliente c usa uma máquina

possui(c,s) - cliente c possui uma máquina

Note-se que cada operação acima, exceto início, tem um parâmetro s que se refere ao estado corrente do banco de dados. Pois, atualizações e consultas têm sentido apenas no contexto de um estado.

O diagrama na fig. 1 mostra três particulares estados do banco de dados. As arestas representam transições e estão rotuladas com operações de atualização, onde o parâmetro estado está implícito por conveniência.

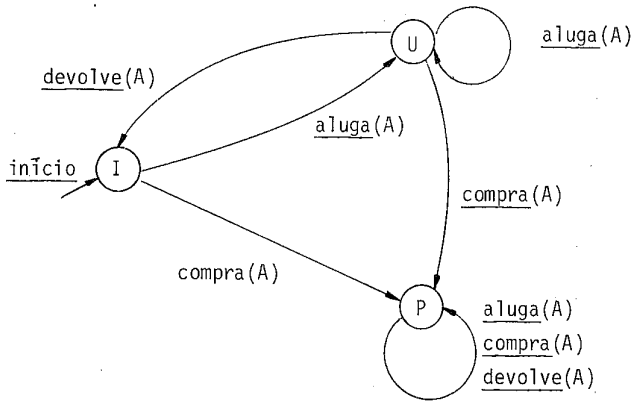


Fig.1 Parte do diagrama de transições do exemplo

3. TRAÇOS

A idéia comum de rastreio é algo como a "lista de instruções processadas, com os respectivos operandos e resultados" (cf. e.g. [14]). No nosso caso, uma instrução consiste da invocação de uma operação de atualização, ou seja, corresponde a uma transição de um estado a outro.

Para clarificar as idéias, voltemos ao nosso exemplo, concentrando-nos em um determinado cliente A. Suponhamos que A faz sucessivamente as seguintes transações, após início,

1. aluga uma máquina;
2. devolve uma máquina;
3. aluga uma máquina;
4. aluga uma máquina;
5. compra uma máquina;
6. devolve uma máquina.

A lista dos nomes das 7 operações acima, começando com início e terminando com o último devolve, seria o traço correspondente a essa série de transações. Será mais conveniente, porém, representar esta lista na forma "aplicativa" abaixo

devolve(A, compra(A, aluga(A, aluga(A, devolve(A, aluga(A, início)))))) (1)

Note-se que o traço acima corresponde a uma expressão, a qual, efetuada, dará como resultado um estado do banco de dados. Considerando o grafo da fig.1, se seguirmos as arestas na ordem indicada em (1), chegaremos ao estado P (passando, no caminho, sucessivamente, por I,U,I,U,U,P). Este estado P é o resultado de efetuarmos a expressão (1) e podemos dizer que esta expressão denota o estado P.

Assim, um traço representa uma sequência de operações, que denota o estado atingido ao fim da sua execução. Além disso, caso mais alguma atualização seja executada, este fato poderá ser levado em conta no traço por meio de uma manipulação com os nomes das atualizações, conforme veremos. Por outro lado, é possível responder a uma consulta através de inspeção e análise da sequência de atualizações constantes do traço.

Essas idéias sugerem o uso dos traços, já que denotam estados, como estrutura de dados.

4. NÍVEIS DE TRAÇO

Na seção anterior, o traço em (1) inclui de fato todas as atualizações invocadas. Por isso, a sequência contém mais informação do que o estritamente necessário para responder às consultas sobre o estado. Realmente, uma análise mais detalhada revela que algumas operações invocadas não chegam a causar mudança de estado (por exemplo a última atualização, devolve). Além disso, o traço indica

os estados intermediários atravessados, a ordem cronológica das operações, etc. Esta informação extra - boa parte de cunho "histórico" - pode ser útil, mas nem sempre é indispensável. Caso não seja necessária, ganhamos a flexibilidade de poder reestruturar a sequência de atualizações de maneira conveniente.

Essas considerações nos levam à idéia de níveis de traço. Por exemplo, o traço (1) denota o estado P no grafo da fig.1. Outros traços que denotam o mesmo estado, com menos informação "histórica", são, entre outros:

compra(A, aluga(A, devolve(A, aluga(A, início)))) (2)

compra(A, início) (3)

Tanto o traço (2) como o (3) foram obtidos de (1). No caso de (2), retiraram-se as atualizações que não causaram mudança de estado. Já no caso (3), foram eliminadas também as atualizações cujos efeitos foram cancelados ou absorvidos por outros.

Aqui vamos distinguir 4 níveis de traço, numerados 0.0, 1.0, 2.0 e 3.0. A própria notação, com ponto decimal, já sugere que outros níveis intermediários podem ser concebidos.

Nível 0.0 . Neste nível, o traço consiste da sequência dos nomes de todas as atualizações invocadas, na sua ordem cronológica. Por exemplo:

aluga(C, devolve(A, devolve(B, aluga(C, compra(B, aluga(A, aluga(B, início)))))))) (4)

Neste caso, a execução simbólica de uma atualização é trivial:basta acrescentar ao traço o nome da nova operação. Por outro lado, a resposta a uma consulta já pode envolver bastante mais trabalho. Por exemplo, para saber se B usa uma máquina no estado denotado por (4), não basta encontrar devolve(B - e responder não - nem tampouco responder sim, simplesmente por ter encontrado aluga(B. É preciso ver se essas operações realmente causaram mudanças de estado e não foram depois canceladas.

O nível 0.0 de traços lembra as chamadas trilhas de auditoria (*audit trails*), familiares na área de banco de dados. De fato para propósitos de auditoria, bem como para o levantamento de estatísticas de uso, tais traços completos podem ser úteis.

A tabela 1 apresenta parte de uma especificação semi-formal para o nosso exemplo ao nível 0.0.

Nível 1.0 . Neste nível o traço consiste apenas dos nomes das atualizações que efetivamente causaram uma mudança de estado. Por exemplo, um traço de nível 1.0 correspondente a (4) é

devolve(A, aluga(C, compra(B, aluga(A, aluga(B, início))))) (5)

Tab.1 Especificação nível 0.0

Atualizações:

.....

aluga(c,t) = aluga(c,t)

compra(c,t) = compra(c,t)

.....

Consultas:

.....

possui(c,t) = sim se o traço t contém compra(c,...)
não caso contrário

Aqui a execução de atualizações ainda é relativamente simples, pois os acréscimos serão feitos sempre no extremo mais recente da sequência. O único cuidado é o de não acrescentar ao traço o nome de uma atualização cujas precondições para alteração de estado não estejam satisfeitas. A execução de consultas se torna um pouco mais simples do que no nível 0.0. Por exemplo, se o traço contém devolve(A sem ocorrências posteriores de aluga(A ou de compra(A então podemos garantir que, neste estado, o cliente A não usa a máquina.

Como antes, o nível 1.0 de traços também lembra um conceito bem conhecido de banco de dados: os *logs* que são conservados com a finalidade de recuperação. Além disso, pode-se pensar também em um banco de dados onde a informação passada não é apagada, toda a informação sendo, por assim dizer, carimbada com a data (cf.[3]).

A tabela 2 mostra parte de uma especificação semi-formal para o exemplo corrente ao nível 1.0.

Tab.2 Especificação nível 1.0

Atualizações:

.....

$$\text{aluga}(c,t) = \begin{cases} \text{aluga}(c,t) & \text{se usa}(c,t) = \text{não} \\ t & \text{se usa}(c,t) = \text{sim} \end{cases}$$

.....

$$\text{devolve}(c,t) = \begin{cases} \text{devolve}(c,t) & \text{se usa}(c,t) = \text{sim e possui}(c,t) = \text{não} \\ t & \text{caso contrário} \end{cases}$$

Consultas:

$$\text{usa}(c,t) = \begin{cases} \text{sim} & \text{se t contém compra}(c,\ldots) \text{ ou se contém aluga}(c,\ldots) \text{ sem ocorrências mais recentes de devolve}(c,\ldots) \\ \text{não} & \text{caso contrário} \end{cases}$$

$$\text{possui}(c,t) = \begin{cases} \text{sim} & \text{se t contém compra}(c,\ldots) \\ \text{não} & \text{caso contrário} \end{cases}$$

Nível 2.0 . Nos níveis 0.0 e 1.0 o comprimento dos traços cresce com o tempo,

pois ambos contêm alguma informação histórica. Porém, se desejamos apenas que o traço identifique o estado atual do banco de dados, podemos cogitar de traços equivalentes que denotem o mesmo estado final de maneira mais compacta.

O traço de nível 2.0 contém somente as atualizações cujos efeitos não foram cancelados ou absorvidos por outras. Como exemplo, consideremos o traço (5) de nível 1.0. Aqui a atualização devolve(A cancela os efeitos de aluga(A enquanto o efeito de aluga(B - que é B usa uma máquina - é absorvido pelos de compra(B - que são B usa e possui a máquina. Assim, um traço de nível 2.0 correspondente a (5) é

$$\underline{\text{aluga}}(C, \underline{\text{compra}}(B, \underline{\text{início}})) \qquad (6)$$

Neste nível a execução das atualizações se torna bastante menos simples, já que certa manipulação interna da sequência se faz necessária. Por outro lado, a execução das consultas fica simplificada por não haver mais no traço operações "negativas" e por estar a informação mais "concentrada".

Os traços de nível 2.0 parecem estar relacionados com esforços para otimizar transações [15], i.e. interações de um usuário envolvendo várias operações em uma única sessão.

A tabela 3 mostra parte de uma especificação ao nível 2.0 do nosso exemplo.

Tab.3 Especificação nível 2.0

Atualizações:

.....

$$\text{compra}(c,t) = \begin{cases} \underline{\text{compra}}(c,t'), \text{ onde } t' \text{ é } t \text{ após remoção de } \underline{\text{aluga}}(c,\ldots) & \text{se possui}(c,t) = \underline{\text{não}} \\ t & \text{se possui}(c,t) = \underline{\text{sim}} \end{cases}$$

$$\text{devolve}(c,t) = \begin{cases} t & \text{se possui}(c,t)=\underline{\text{sim}} \text{ ou usa}(c,t)=\underline{\text{não}} \\ t', \text{ onde } t' \text{ é } t \text{ sem } \underline{\text{aluga}}(c,\ldots) & \text{caso contrário} \end{cases}$$

Consultas:

$$\text{usa}(c,t) = \begin{cases} \underline{\text{sim}} & \text{se } t \text{ contém } \underline{\text{aluga}}(c,\ldots) \text{ ou } \underline{\text{compra}}(c,\ldots) \\ \underline{\text{não}} & \text{caso contrário} \end{cases}$$

$$\text{possui}(c,t) = \begin{cases} \underline{\text{sim}} & \text{se } t \text{ contém } \underline{\text{compra}}(c,\ldots) \\ \underline{\text{não}} & \text{caso contrário} \end{cases}$$

Nível 3.0 . Mesmo ao nível 2.0 pode haver mais de um traço denotando o mesmo estado. Isto se deve ao fato de certas sequências de atualizações causarem os mesmos efeitos finais independentemente de sua ordem de execução. Por exemplo, tanto faz, primeiro B comprar uma máquina e depois C alugar uma máquina, ou vice-versa: os efeitos finais serão os mesmos.

No nível 3.0 queremos que cada estado seja denotado por um único traço. As

sim, a passagem do nível 2.0 ao nível 3.0 envolve um critério para escolher um representante dentre os vários traços equivalentes. Frequentemente, esta escolha pode ser efetuada por meio de uma simples reordenação dos traços de nível 2.0. No nosso exemplo, vamos tomar como critério para traços de nível 3.0 a ordem lexicográfica decrescente dos clientes. Assim, o (único) traço de nível 3.0 correspondente a (6) é

$$\underline{\text{compra}}(B, \underline{\text{aluga}}(C, \underline{\text{início}})) \qquad (7)$$

Neste nível, a execução de atualizações envolve manipulações internas do traço, em grau maior do que no nível 2.0, já que a adição do nome de uma atualização só será feita em posição apropriada na sequência. As consultas são tão simples quanto no nível 2.0, podendo ser mais eficientes se tirarmos partido da ordenação (cf. seção 5).

Cada traço de nível 3.0 representa, de maneira única, toda uma família de sequências de atualizações. Isso também tem um paralelo familiar em processamento de dados: comparação de conjuntos de ítens. Uma maneira usual de se comparar dois conjuntos de ítens de informação, armazenados, digamos, em duas fitas, começa ordenando-os segundo um mesmo critério.

A tabela 4 contém parte da especificação do nosso exemplo no nível 3.0.

Tab.4 Especificação nível 3.0

Atualizações:

.....

$$\text{compra}(c,t) = \begin{cases} t, \text{ com a remoção de eventual } \underline{\text{aluga}}(c,\ldots) \text{ e } \underline{\text{compra}}(c,\ldots) \text{ inserido na posição apropriada} & \text{se possui}(c,t) = \underline{\text{não}} \\ t & \text{se possui}(c,t) = \underline{\text{sim}} \end{cases}$$

.....

Consultas:

$$\text{usa}(c,t) = \begin{cases} \text{sim} & \text{se t contém } \underline{\text{aluga}}(c,\ldots) \text{ ou } \underline{\text{compra}}(c,\ldots) \\ \text{não} & \text{caso contrário} \end{cases}$$

.....

5. ESPECIFICAÇÃO PROCEDURAL

Conforme foi visto na seção anterior, a execução de atualizações ou de consultas em um traço envolve manipulações simbólicas na sequência correspondente. Uma especificação procedural trata o traço como uma sequência de símbolos e "implementa" as operações por procedimentos de manipulação de símbolos. Para escrever tais procedimentos é conveniente se lançar mão de uma linguagem de programação voltada para manipulação simbólica, como SNOBOL, LISP, ICON, REDUCE, etc. Aqui vamos usar uma notação procedural [4], baseada em PLANNER [8], a qual é bas-

tante simples de se entender e de traduzir para uma das linguagens acima. Cada procedimento (op) consiste de um cabeçalho e um corpo. Neste, os comandos são examinados em sequência e o valor devolvido pelo procedimento é o da primeira expressão à direita de um '⇒' tendo à esquerda uma expressão lógica com valor verdadeiro (a expressão lógica vazia sendo considerada verdadeira sempre). A instrução match t é do gênero case; dentro dela ocorre um processo de casamento de padrões, o valor da instrução sendo o do lado direito da expressão cujo lado esquerdo casa com o traço t.

A tabela 5 contém as especificações procedurais da atualização compra em cada um dos 4 níveis considerados. Note-se como os textos dos procedimentos vão aumentando de complexidade conforme o nível cresce. A tabela 6 mostra para cada nível uma especificação procedural da consulta usa. Aqui é interessante notar que o procedimento de um nível serviria para níveis subsequentes, porém são apresentados para esses programas mais simples e eficientes.

Tab.5 Especificação procedural da atualização compra nos 4 níveis

```
. Nível 0.0
  op  compra(c: cliente, t:bd) :bd
      ⇒ compra(c,t)
  po
. Nível 1.0
  op  compra(c: cliente, t:bd) :bd
      possui(c,t) ⇒ t;
      ⇒ compra(c,t)
  po
. Nível 2.0
  op  compra(c: cliente, t:bd) :bd
      var  x: cliente, s:bd
        possui(c,t) ⇒ t;
        ¬usa(c,t)  ⇒ compra(c,t);
        match  t
           aluga(x,s) ⇒ if x=c  then compra(c,s)
                        else aluga(x, compra(c,s))
           compra(x,s) ⇒ compra(x,compra(c,s))
        endmatch
  po
  Nível 3.0
  op  compra(c: cliente, t: bd) :bd
      var  x: cliente, s: bd
```

```
      possui(c,t) ⇒ t;
      match t
         início  ⇒ compra(c,t)
         aluga(x,s) ⇒ if x=c  then compra(x,s)
                        else if c<x then compra(x,t)
                                 else aluga(x, compra(c,s))
         compra(x,s) ⇒ if c<x then compra(c,t)
                        else  compra(x, compra(c,s))
      endmatch
   po
```

Tab.6 Especificação procedural da consulta usa nos 4 níveis

. Nível 0.0

```
   op   usa (x: cliente, t:bd) : lógico
        var  y: cliente, s:bd
        match  t
           início ⇒ não
           aluga(y,s) ⇒ if  x=y  then  sim
                        else  usa(x,s)
           compra(y,s)⇒ if  x=y  then  sim
                        else  usa(x,s)
           devolve(y,s)⇒ if  x=y  then possui(x,s)
                         else  usa(x,s)
        endmatch
   po
```

. Nível 1.0

```
   op   usa(x: cliente, t:bd) : lógico
        var y: cliente, s:bd
        match t
           início ⇒ não
           aluga(y,s) ⇒ if x=y then  sim
                        else  usa(x,s)
           compra(y,s) ⇒ if x=y then sim
                         else  usa(x,s)
           devolve(y,s) ⇒ if x=y then não
                         else usa(x,s)
        endmatch
   po
```

```
.Nível 2.0
op  usa(x: cliente, t:bd) : lógico
    var  y: cliente, s:bd
    match t
      início  ⇒  não
      aluga(x,s) ⇒  if x=y  then sim
                    else  usa(x,s)
      compra(x,s) ⇒  if x=y  then sim
                     else usa(x,s)
    endmatch
po

.Nível 3.0
op  usa(x: cliente, t:bd) : lógico
    var y: cliente, s:bd
    match  t
      início ⇒ não
      aluga(x,s) ⇒  if x=y  then sim
                    else  if x<y then  não
                          else  usa(x,s)
      compra(x,s) ⇒  if x=y  then sim
                     else  if x<y then  não
                           else  usa(x,s)
    endmatch
po
```

De maneira inteiramente análoga podem ser escritos procedimentos para as operações restantes, completando assim as especificações procedurais dos vários níveis. É interessante observar que o módulo da especificação procedural de um dado nível gerará apenas os traços desse nível.

Cabe observar que a especificação procedural é não apenas formal mas também executável. Isso permite experimentar com a especificação, para ver se foi realmente captado o que se tinha em mente, antes de se lançar à tarefa cara e demorada de implementação em máquina [4].

A especificação procedural, a um dado nível, é basicamente um processo para converter uma sequência de operações dada em um traço equivalente neste nível. Por exemplo, consideremos

$$\begin{array}{l} \text{aluga(C, devolve(A, devolve(B, aluga(C,} \\ \text{compra(B, aluga(A, aluga(B, início)))))))} \end{array} \qquad (8)$$

Se executarmos a expressão (8) com os procedimentos de nível 1.0, obteremos como resultado o traço (5). Por outro lado, o resultado de executar

devolve(A, aluga(C, compra(B, aluga(A, aluga(B, início)))))

ao nível 3.0 é

compra(B, aluga(C, início))

6. ESPECIFICAÇÃO ALGÉBRICA

Conforme mencionamos, a especificação procedural pode ser vista como um mecanismo para converter certas sequências de operações em outras equivalentes. Em vez disso, poder-se-ia pensar em apenas declarar que pares de sequências de operações são equivalentes. Essas considerações sugerem especificações algébricas, por meio de axiomas na forma de equações condicionais [5].

Consideremos de novo o traço (4), que denota um estado onde B usa uma máquina. Que equações condicionais precisaríamos ter para poder deduzir a equação usa(B,(4)) = sim ?

Podemos pensar em ir transformando o termo usa(B,(4)) em outros mais simples, "fazendo usa entrar", até transformá-lo em sim [12]. Para isto, vêm à mente axiomas como

$$x \neq y \rightarrow \text{usa}(x, \text{aluga}(y,s)) = \text{usa}(x,s) \qquad (9)$$

e

$$\text{usa}(x, \text{compra}(x,s)) = \text{sim} \qquad (10)$$

Esses dois axiomas correspondem a dois caminhos na execução do procedimento usa no nível 0.0. Por exemplo, (9) corresponde ao caminho em que t casa com aluga(y,s) e $x \neq y$. (cf. tabela 6)

A tabela 7 apresenta equações condicionais que correspondem à especificação nível 0.0. Elas permitem determinar o resultado de cada consulta.

Imaginemos agora que desejamos mostrar que os traços (4) e (5) denotam o mesmo estado. Como deduzir (4) = (5)? Podemos pensar em eliminar de (4) nomes de atualização que não causaram mudanças de estado. Isso pode ser feito pelos axiomas 1 a 4 da tabela 8.

Tentemos agora mostrar (5) = (6). As equações 5 e 6 da tabela 10 permitem fazer parte do trabalho. A equação 5 afirma que um devolve cancela um aluga imediatamente precedente. Para permitir tratar casos de operações não adjacentes são introduzidos axiomas de comutatividade - tanto incondicional como 7,8,9,10, quanto condicional como 11.

No caso do nosso exemplo, podemos com esses axiomas mostrar também (6)=(7). Em geral, seria necessário adicionar mais alguns axiomas, tipicamente de comuta

tividade.

Tab.7 Axiomas para consultas

1. usa(x, início) = não
2. usa(x, aluga(x,s)) = sim
3. x≠y → usa(x, aluga(y,s)) = usa(x,s)
4. usa(x, compra(x,s)) = sim
5. x≠y → usa(x, compra(y,s)) = usa(x,s)
6. possui(x, início) = não
7. possui(x, aluga(y,s)) = possui(x,s)
8. possui(x, compra(x,s)) = sim
9. x≠y → possui(x, compra(y,s)) = possui(x,s)

---

10. usa(x, devolve(x,s)) = possui(x,s)
11. x ≠ y → usa(x, devolve(y,x)) = usa(x,s)
12. x≠y → possui(x, devolve(y,s)) = possui(x,s)

Tab.8 Axiomas para atualização

1. usa(x,s) = sim → aluga(x,s) = s
2. possui(x,s) = sim → compra(x,s) = s
3. usa(x,s) = não → devolve(x,s) = s
4. possui(x,s) = sim → devolve(x,s) = s

---

5. devolve(x, aluga(x,s)) = devolve(x,s)
6. compra(x, aluga(x,s)) = compra(x,s)

---

7. aluga(x, aluga(y,s)) = aluga(y, aluga(x,s))
8. compra(x, compra(y,s)) = compra(y, compra(x,s))
9. aluga(x, compra(y,s)) = compra(y, aluga(x,s))
10. devolve(x, compra(y,s)) = compra(y, devolve(x,s))
11. x≠y → devolve(x, aluga(y,s)) = aluga(y, devolve(x,s))

Os axiomas das tabelas 7 e 8 constituem uma especificação algébrica para o nosso exemplo. (Na verdade, os axiomas 10, 11 e 12 da tabela 7 se tornam redundantes em presença dos da tabela 8 e poderiam ser eliminados). Note-se que, mesmo sem pretender ser exaustivo e rigoroso, se pode associar a cada nível de traço determinadas categorias de leis algébricas, como comutatividade, idempotência, etc.

7. CONCLUSÕES

Indicamos como traços podem ser utilizados para dar especificações de dados, abstraindo-se de detalhes de representação. Se bem que esta idéia tenha

pontos de contato com [1], exploramos aqui especialmente a existência de vários níveis de traço. Pode-se visualizar a passagem de um nível de traço a outro superior através da abstração de detalhes de informação histórica sobre a sequência de operações.

Foram apresentados três formatos de especificação. O primeiro (seção 4) consiste de uma especificação não procedural (do gênero entrada-saída) para cada operação. O formato procedural (seção 5) consiste em procedimentos de manipulação simbólica para cada operação. Finalmente, o formato algébrico (seção 6), consiste de equações condicionais relacionando as operações entre si. A passagem de um formato a outro pode ser feita de modo razoavelmente sistemático [12].

As especificações apresentadas se baseiam fortemente em (sequências de) atualizações. Outros formatos de especificação, como pré-condições e efeito [9], se voltam para consultas. Essas especificações, em vários formatos, se complementam umas às outras, havendo interesse no desenvolvimento de metodologias para efetuar transformações entre elas [16].

8. REFERÊNCIAS

[1] BARTUSSEK, W., PARNAS, D., "Using traces to write abstract specifications for software modules". *Technical Report* 77-012, University of North Carolina, Chapel Hill, NC, 1977.

[2] BRODIE, M.L., ZILLES, S.N. (eds.) *Proc. of the Workshop on Data Abstraction, Databases and Conceptual Modelling*, 1981.

[3] BUBENKO Jr., J.A., "On the role of 'understanding models' in conceptual scheme design", *Proc. 5th Very Large Data Bases Conf.*, pp. 129-139, 1979.

[4] FURTADO A.L., VELOSO, P.A.S., "Procedural specifications and implementations for abstract data types", *SIGPLAN Notices*, vol. 16 nº3, pp. 53-62, março de 1981.

[5] GOGUEN, J.A., THATCHER, J.W., WAGNER, E.G., "An initial algebra approach to the specification, correctness and implementation of abstract data types", In *Current Trends in Programming Methodology*, vol IV, R.T. Yeh (ed.), Prentice-Hall, Englewood Cliffs NJ, pp. 80-149, 1978.

[6] GOTTLIEB, C.C., "Some large questions about very large data bases," *Proc. 6th Very Large Data Bases Conf.*, pp. 3-7, 1980.

[7] GUTTAG, J., "Notes on type abstraction (version 2)", *IEEE Trans. on Software Engin.*, vol. 6, nº1, pp. 13-23, janeiro de 1980.

[8] HEWITT, C., "Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot",MIT, Dept. of Mathematics, Ph.D thesis, 1972.

[9] HOARE, C.A.R., "An axiomatic basis for computer programming", *Comm. of the ACM*, vol. 12, nº10, pp.576-580, 1969.

[10] LISKOV, B., ZILLES, S.N., "An introduction to formal specifications of data abstractions", In *Current Trends in Programming Methodology*, vol.I, R.T. Yeh (ed.) Prentice-Hall, Englewood Cliffs, NJ, pp. 1-32, 1977.

[11] PARNAS, D.L., "The use of precise specifications in the development of software", In *Information Processing 77*, B. Gilchrist (ed.), North-Holland, Amsterdam, pp. 861-867, 1977.

[12] PEQUENO, T.H.C., VELOSO, P.A.S., "Do not write more axioms than you have to", *Proc. Internat. Computing Symposium*, pp. 488-498, 1978.

[13] SIBLEY, E.H., "Database management systems: past and present", [2], p.192.

[14] SIPPL, C.J., SIPPL, C.P., *Computer Dictionary and Handbook*, Howard W. Sons & Co., Inc., Indianopolis, IN, 1978.

[15] SMITH, J.M., CHANG, P.Y.-T., "Optimizing the performance of a relational database interface", *Comm. of the ACM*, vol. 18, nº10, pp.568-579, 1975.

[16] VELOSO, P.A.S., CASTILHO, J.M.V., FURTADO, A.L., "Systematic derivation of complementary specifications", *Proc. 7th Very Large Data Bases Conf.*, pp. 409-421, 1981.