

XIV CONFERENCIA LATINOAMERICANA DE INFORMATICA / CLEI

17as JORNADAS ARGENTINAS DE INFORMATICA E INVESTIGACION OPERATIVA / SADIO

BUENOS AIRES, 26 AL 30 DE SEPTIEMBRE DE 1988

TRABAJOS SELECCIONADOS

SOCIEDAD ARGENTINA DE INFORMATICA
E INVESTIGACION OPERATIVA

MFW 829

005.74
CON6e
v.1.

PEDECIBA
INFORMATICA

COMITE EJECUTIVO

Rodolfo A. Boldt (Presidente)
Héctor Monteverde (Coordinador)
Clara Fuks (Tesorera)
Domingo Trassens (Prensa)
Juan Ignacio Hernández (Editor)

COMITE DE PROGRAMA

Jorge Vidart (Presidente) ESLAI, Argentina
Marcos Borges/ Núcleo de Computación, UFRJ, Brasil
Armando de Giusti / U.N. de La Plata, Argentina
Riccardo Ferrari / U.C.N. Sra. de Asunción, Paraguay
Alberto Flores Balanza / U.N. San Andrés, Bolivia
Pedro Hepp / U.C. de Chile, Chile
Ida Holz / U. de la República, Uruguay
Rodrigo López / U. de los Andes, Colombia
Diómedes Sarmiento / APCI, Perú
Amelia Soriano / U.C. de Venezuela, Venezuela
Alberto Teszkiewicz / UBA, Argentina

nº 2271

17/10/88

T O M O 1

I N D I C E

Página

CAPITULO 1: ALGORITMOS Y ESTRUCTURAS DE DATOS

Un algoritmo de Evaluación de Potencias utilizando cadenas de suma y resta
Francois Morain, Jorge Olivos, Francia - Chile 2

Reorganizaciones locales en el árbol K-D-B, su eficiencia en situaciones dinámicas
M. Diaz, O.Santana, G.Rodríguez, M.Martín- España 17

El árbol multidimensional equilibrado en altura: influencia de su comportamiento dinámico en las recuperaciones exactas, en rango y de los vecinos más próximos
O.Santana, M.Diaz, Z.Hernández, JC.Del Pino-España 33

O módulo de posicionamento relativo do projeto tranca
Marcelo Lubaszewski - Brasil 47

Una herramienta para la minimización de desperdicios
Leodegar Cabello - Paraguay 67

Una generalización al problema del viajante GTSP
Delia Kesner, Daniel Yankelevich - Argentina 77

Un algoritmo para enrutamiento no local
Diego Andrade Stacey - Ecuador 103

CAPITULO 2: PROGRAMACION FUNCIONAL Y LOGICA

Un sistema de tipos polimórficos para un lenguaje funcional con definiciones de constantes y tipos abstractos
Verónica Gaspes, Alvaro Tasistro - Argentina 126

Un sistema de tipos para un lenguaje funcional
Sylvia Daiqui, Verónica Gaspes, Delia Kesner, Carolina Lavatelli - Argentina 147

Implementación de una máquina de Multi-combinadores Categóricos
Martín A. Musicante y Rafael D. Lins - Argentina Brasil 165

Un esquema para la Programación Orientada a Objetos en PROLOG' 180
Javier Pinto - Chile

CAPITULO 3:ESPECIFICACIONES

Especificación de Operadores de Control y desarrollo de Herramientas de puesta a punto para un ambiente de Programación Lógica 192
Nora Sease - Argentina

Una implementación funcional a partir de una especificación algebraica 220
Fernando Carpani - Diana Cukierman - Uruguay

Definición de formas normales de términos CCS 229
A. Soriano - Venezuela

Una representación no tradicional de producto cartesiano 245
Laura Almada, Gustavo Betarte, Paula Severi- Uruguay

CAPITULO 4: BASE DE DATOS

Un modelo de Administración de Buffers para la predicción del desempeño de Sistemas de Bases de Datos 264
Ignacio R. Casas - Kenneth C. Sevcik- Chile-Canadá

Mecanismos de controle para um SGBDR Ativo 292
Decio Fonseca - Brasil

Artesano: Una arquitectura de administración de bases de datos basada en vistas con enfoque funcional 305
Paulina Frenkel, Nicolás Casariego, Jorge Zarauza
Argentina

Los SGBD Extensibles 316
H. Steffen, A. Gutiérrez, R. Ruggia- Francia-Uruguay

Gerenciamento de esquemas de dados dinamicamente modificaveis em modelo de base de dados orientados a objetos 338
Caetano Traina - Jan Frans Willem Slaets - Brasil

REVER: uma interface semantica para bancos de dados relacionais 356
Mónica de Lima Nogueira - Claudia Bauzer Medeiros
Brasil

Servidores de arquivos en redes de computadores 370
Liane Margarida Rckenbach Tarouco - M.Almeida - Brasil

Em busca de um sistema generalizado de gerencia de dados, modelos e interfaces com o usuario 388
María E. Braz -Rubens Nascimento Melo - Brasil

CAPITULO 5: INGENIERIA DE SOFTWARE

Um estudo sobre o tratamento da estrutura lógica de documentos 402
José Valdeni De Lima - Brasil

Hypermedia: nuevas herramientas para el conocimiento 417
Francesca Greselín - Riccardo Ferrari - Paraguay

Ferramentas para edicao de texto em um ambiente operacional amigavel 430
Ana Carla Bon - Luiz Montes Monte - Brasil

Organizacao do sistema operacional sombra 451
R. S. De Oliveira - C.M. Da Costa - Brasil

Modelando propriedades dinamicas de sistemas: comparacao entre duas abordagens 462
Carlos Heuser - Brasil

Un sistema de compilación, ejecución y análisis de redes de Petri Extendidas para computadoras IBM-PC compatibles 473
Verónica Ludueña - Oyarzabal Juan C. - Argentina

Convergence Properties of Stepwise Refined Sequences of Petri Nets 483
Ridardo Ferré - Suecia

TABA: Uma estacao de trablho para engenheiro de Software 493
Ana Calvacanti da Rocha - Jano Moreidra de Souza
Brasil

Editores diagramáticos baseados em formalismos gramaticais 509
Roberto Tomm Price - Eloi L. Favero - Brasil

Programación y metodologías orientadas a objetos y las metodologías avanzadas de diseño de Soft 528
Gustavo A. Pollitzer - Argentina

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Un Algoritmo de Evaluación de Potencias utilizando Cadenas de Suma y Resta

Francois Morain *
Jorge Olivos †

RESUMEN

Se propone un algoritmo simple (autómata finito) para calcular x^n que utiliza multiplicaciones y una sóla división. Este algoritmo mejora, en términos relativos, en un 33% el clásico método binario [11], y en términos absolutos la reducción en el número de operaciones es de un 11.11%.

1. Introducción

Los métodos más recientes empleados en los test de primalidad y algoritmos de factorización de enteros [3, 8, 9, 11, 12, 13] requieren efectuar productos de números de gran tamaño. Esta clase de cálculos corresponde al problema de evaluación de potencias (x^m) y ha sido formalizado en la literatura introduciendo el concepto de *cadena de suma* [11]. Una cadena de suma para el entero positivo m es una secuencia de $(r+1)$ términos $(a_0, a_1, \ldots, a_r)$ tal que:

$$1 = a_0, a_1, \ldots, a_r = m, \tag{1}$$

y con la propiedad,

$$a_i = a_j + a_k, \quad \text{para} \quad k \leq j < i, \tag{2}$$

* Institut National de Recherche en Informatique et en Automatique (INRIA), Domaine de Voluceau, B. P. 105 78153 Le Chesnay CEDEX (France).

† Departamento de Ciencias de la Computación, Universidad de Chile, Casilla 2777. Santiago. Chile. Proyecto Fondecyt 348/87.

para todo $i = 1, 2, \ldots, r$. El largo más corto, r, para el cual existe una cadena de suma para m se denota por $l(m)$. El estudio de esta función, así como de sus generalizaciones, ha sido abordada por numerosos autores [2, 7, 10, 19, 21]. Todos los algoritmos conocidos que evalúan la función l, para un m cualquiera, son de tipo exponencial y más aún se conjetura que el cálculo de $l(m)$ es un problema NP–completo. El método binario [9, 11], ver sección **2**, es un algoritmo, no optimal, que se utiliza frecuentemente en la práctica.

Una *cadena de suma y resta* tiene la regla:

$$a_i = a_j \pm a_k \tag{3}$$

en lugar de (2). En términos prácticos, estas cadenas surgen cuando en la evaluación de x^m se puede utilizar multiplicaciones y divisiones. 31 es el número más pequeño para el cual existe una cadena de suma y resta de largo menor que $l(31)$ y esta es: 1, 2, 4, 8, 16, 32, 31. En el caso de los test de primalidad que motivan este trabajo (F. Morain, J. Olivos, Speeding up the Computations on an Elliptic Curve using Addition-Substractions Chains, Preliminary Report. INRIA), el costo de una resta equivale al de una suma justificando así el interés en estas últimas cadenas. El objetivo de este artículo es proponer un algoritmo muy simple para el cálculo de una cadena de suma y resta cuya complejidad (número de operaciones de suma y resta) esperada supera al método binario.

La sección **2** recuerda el método binario. La sección **3** presenta el nuevo algoritmo en sus dos variantes y en la sección **4** se estudia la complejidad esperada de ambas alternativas.

2. El método binario

Una manera razonable de calcular x^n se basa en la representación binaria de $n = b_r \ldots b_0$. Se asocia a la secuencia de bits una palabra del lenguaje $\{S, M\}^*$, reemplazando cada "1" por SM (salvo b_r) y cada "0" por S, donde S se interpreta como un doble (*squaring*) y M como una suma (*multiply*). Por ejemplo, si $n = 23$ su representación es 10111; y la palabra asociada es $SSMSMSM$. Los cálculos correspondientes son: x^2, x^4, x^5, x^{10}, x^{11}, x^{22}, x^{23}; para más detalles ver [11]. Una implementación en Pascal del algoritmo es la siguiente (ver [9]):

function *power*(x: *number*; n: *integer*): number;

```
begin
  if n = 0 then power := 1
  else if n = 1 then power := x
  else if (n mod 2) = 0
    then
      power := sqr(power(x, m div 2))
    else
      power := sqr(power(x, m div 2)) * b
end;
```

Si denotamos por $Q_{bp}(n)$ el número de multiplicaciones requeridos por este algoritmo en el cálculo de la enésima potencia de un número, entonces:

$$Q_{bp}(n) = \lambda(n) + \nu(n) - 1 \tag{1}$$

donde,

$$\lambda(n) = \lfloor lg_2 n \rfloor,$$
$$\nu(n) = Card\{i \mid 0 \leq i \leq \lambda(n),\ b_i = 1\}.$$

Si se considera la clase de los enteros n cuyo $\lambda(n)$ es constante se verifica que el número máximo y esperado de $\nu(n)$ es $\lambda(n)$ y $\frac{1}{2}\lambda(n)$, respectivamente. El Teorema siguiente resume el comportamiento del método binario,

Teorema 1. *Para los enteros n que pertenecen al intervalo $[2^s, 2^{(s+1)} - 1]$, se tiene que la complejidad en el caso peor es igual a $2\lambda(n)$ y en el esperado a $\frac{3}{2}\lambda(n)$.*

Observemos que el caso peor se realiza cuando $n = 2^{(s+1)} - 1$, esto es, cuando todos sus dígitos binarios son iguales a uno. Por otra parte, es evidente que todo algoritmo de evaluación de potencias tendrá una complejidad de al menos $\lambda(n)$.

3. Nuevo Algoritmo

El algoritmo siguiente reemplaza el cálculo de n por otros números auxiliares (b y c) verificando: $n + b = c$; de aquí se deduce que $n = c - b$. Esta última operación es la única resta (o división) presente en el algoritmo. Apoyándonos en la representación binaria de los tres números, la propiedad

que tienen los bits de b , iguales a uno, es de "voltear" los bloques de bits correspondientes de n. Estos últimos bloques deben ser de largo mayor o igual a tres. Los dos ejemplos siguientes ilustran el algoritmo.

Ejemplo 4.1 Se utiliza la notación ya mencionada. En primer lugar ilustramos lo que ocurre con el caso peor del método binario.

```
n:   1 1 1 1
b:       1
c:  10 0 0 0
```

a continuación, un caso más general:

```
n:  10011110111000111001100 1
b:        1000100000100000000
c:  1010000100000100000011001
```

en el primer caso el número de operaciones se reduce de 6 a 5 y en el segundo caso de 36 a 31.

Sea $C(n)$ el número de operaciones del nuevo algoritmo, se deduce de inmediato que:

$$C(n) = T(c) + \nu(b) \tag{1}$$

y por construcción:

$$C(b) = Q_{bp}(b). \tag{2}$$

donde surgen dos alternativas. En primer lugar, si c se calcula con el método binario (i.e. $T(c) = Q_{bp}(c)$), entonces el número esperado de operaciones en este caso es de $\frac{11}{8}\lambda(n)$ que significa una reducción relativa de un 25%. Si c se calcula siempre con el nuevo algoritmo, entonces la complejidad esperada se reduce a $\frac{4}{3}\lambda(n)$ y el correspondiente algoritmo representa una mejora relativa, en relación al binario, de un 33%. En términos absolutos la reducciones son de 8.5% y 11.11%, respectivamente.

Ambos algoritmos se visualizan fácilmente en término de autómatas finitos. A continuación se entrega una descripción gráfica de ambos algoritmos. La primera figura corresponde a la primera versión del algoritmo, esto es, cuando se ingresa al estado T11 (bloque de al menos dos unos) se comienzan a voltear los unos correspondientes y se sale de él cuando se lee

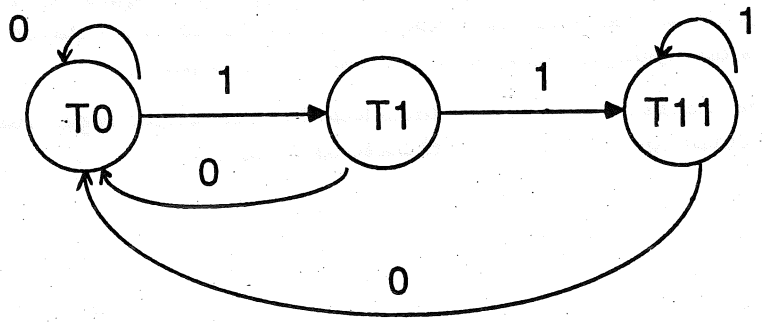


Fig. 1

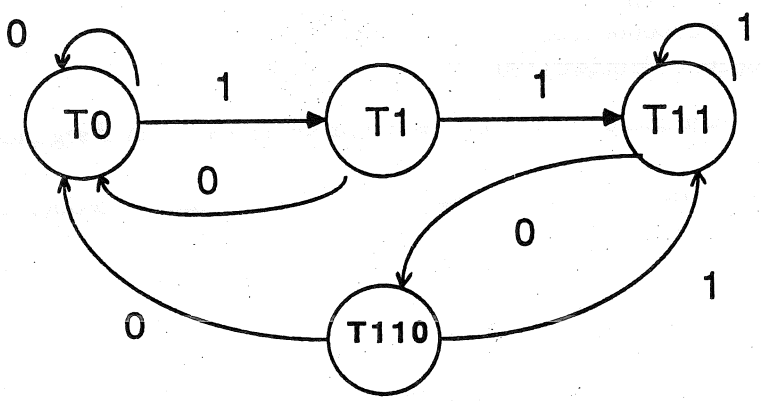


Fig. 2

un cero. En el segundo algoritmo, cuando se está en el estado T11 y se lee un cero se pasa al estado T110 que en caso de recibir un uno regresa al estado anterior (efectuando un puente, como se ilustró en el ejemplo anterior).

4. Análisis del Algoritmo.

En el anexo se entrega el análisis del algoritmo en su segunda versión utilizando el sistema $\Lambda\Upsilon^{\Omega}$ que se desarrolla en el INRIA (P. Flajolet, B. Salvy, P. Zimmermann, Lambda-Upsilon-Omega: An Assistant Algorithms Analyser, AECC-6, Rome, 1988). Este sistema permite efectuar el análisis automático de clases de algoritmos bien definidas operando sobre estructuras de datos que admitan descomposiciones en término de estructuras más simples. Consiste en un Analizador Algebraico que compila especificaciones de algoritmos (en un lenguaje de alto nivel, pág. 1 del anexo) en funciones generatrices de costos promedios (págs. 2–4) y de un Analizador Analítico que extrae información asintótica sobre los coeficientes de las funciones generatrices (págs. 4–7) obtenidas en la fase previa. La parte analítica se basa en el sistema de cálculo simbólico MAPLE, desarrollado en Waterloo.

Se observa que el algoritmo es una traducción inmediata del gráfico de la sección previa y en la página 4 del anexo se encuentra el resultado asintótico (función av_tau_treat_n) correspondiente.

5. Resultados experimentales.

El primer autor ha utilizado la primera versión del algoritmo en la implementación del test de Atkins (F. Morain, Implementation of the Goldwaser-Killian-Atkin test. Preliminary Report. INRIA). Trabajando con números de 100 dígitos decimales se ha obtenido una ganancia en tiempo en el cálculo de la ley sobre una curva elíptica que concuerda con lo anunciado por la teoría.

6. Agradecimientos.

El segundo autor quisiera expresar su reconocimiento al INRIA por una invitación como profesor visitante que le permitió abordar este trabajo. Este trabajo contó tambien con el apoyo del programa de cooperación Franco–Chileno.

Ambos autores agradecen a P. Zimmermann por su excelente disposición a explicar el uso del sistema $\Lambda\gamma^{\Omega}$ que se encuentra actualmente en su fase final de desarrollo.

7. Bibliografía

1. W. Bosma. *Primality testing using elliptic curves.* Report 85–12, Math. Institut, Universeit van Amsterd.

2. A. Brauer. *On addition chains.* Bull. Amer. Math. Soc., 45, p. 736–739.

3. R.P. Brent. *Some integer factorization algorithms using elliptic curves.* Research Report CMA-R32-85, The Australian national University, Canberra, 1985.

4. J. Brillhart, D.H. Lehmer, J.L. Selfridge, B. Tuckerman, S.S. Wagstaff Jr. *Factorizations of bn 1. Contemporary Mathematics*, 22, AMS, 1983.

5. J.W.O.S. Cassels. *Diophantine equations with special references to elliptic curves.* J. London Math. Soc., 41, 1966, p. 193–291.

6. D.V. Chudnovsky, G. V. Chudnovsky. *Sequences of numbers generated by addition in formal groups and new primality and factorization tests.* Research report RC 11262, IBM, Yorktown Heights, 1985.

7. P.Erdős. *Remarks on number theory III : on addition chains.* Acta Arithmetica, 6, 1960, p. 77–81.

8. S. Goldwasser, J. Killian. *Almost all primes can be quickly certified.* Proc. 18th STOC, Berkely, 1986, p. 316–329

9. G.H.Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley. 1984.

10. B.S. Kaliski, Jr. *A pseudo-random bit generator based on elliptic logarithms.* Proc. Crypto 86, p. 13–1, 13–21.

11. D.E. Knuth. *Seminumerical algorithms.* The art of computer programming, Vol. 2, Addison-Wesley. 2o. Ed. 1980.

12. N. Koblitz. *Elliptic curve cruptosystems.* Math. of Comp., v 48, 177, jan 1987, p. 203–209.

13. H.W. Lenstra, Jr. *Factoring with elliptic curves.* Report 86–18, Math. Inst., Univ. Amsterdam, 1986.

14. H.W. Lenstra, Jr. *Elliptic curves and number theoretic algorithms.* Report 86–19, Math. Inst., Univ. Amsterdam, 1986.

15. D.P. McCarthy. *The optimal algorithm to evaluate x^n using elementary multiplication methods.* Math. of Comp., v 31, 137, jan 1977, p. 251-256.

16. D.P. McCarthy. *Effect of improved multiplication efficiency on exponentiation algorithms derived from addition chains.* Math. of Comp., v 46, 174, apr 1986, p. 603–608.
17. P.L. Montgomery. *Modular multiplication without trial division.* Math. of Comp., v 44, 170, apr 1985, p. 519–521.
18. J. Olivos. *On Vectorial Addition Chains.* Journal of Algorithms, 2, 1981, p. 13–21.
19. J.M. Pollard. *Theorems on factorization and primality testing.* Proc. Cambridge Phil. Soc., 76, 1974, p. 521–528.
20. R.L. Rivest, A. Shamir, L. Adleman. *A method for obtaining digital signatures and public-key cryptosystems.* Comm. of the ACM, v 21, 2, feb 1978, p. 120–126.
21. A. Schönhage. *A lower bound for the length of addition chains.* Theor. Comput. Science, 1, 1, 1975, p. 1–12.
22. J.T.Tate. *The arithmetic of elliptic curves.* Inventiones Math., 23, 1974, p. 179–206.

```
*************************************************************************
type chain = sequence(bit);      % A + B = C then A = C - B %
     valid_chain = product(chain,one);
     bit = zero | one;
     zero,one = atom(1);

procedure treat(c:valid_chain);
case c of
  (c1,one) : begin treatStart(c1); divide end
end;

procedure treatStart(c:chain);
case c of
  () : nil;
  (zero,c1) : begin treat0(c1) end;
  (one,c1)  : begin treat1(c1) end
end;

procedure treat0(c:chain);
case c of
  () : begin squaring; multiplyC end;
  (zero,c1) : begin squaring; treat0(c1) end;
  (one,c1)  : begin squaring; treat1(c1) end
end;

procedure treat1(c:chain);       % one 1 has been recognized %
case c of
  () : begin multiplyC; multiplyC end;
  (zero,c1) : begin multiplyC; squaring; treat0(c1) end;
  (one,c1) : begin squaring; multiplyB; treat11(c1) end
end;

procedure treat11(c:chain);      % at least two 1 have been recognized %
case c of
  () : begin squaring; squaring; multiplyC end;
  (zero,c1) : begin squaring; treat110(c1) end;
  (one,c1)  : begin squaring; treat11(c1) end
end;

procedure treat110(c:chain); % at least two 1 and one 0 have been recognized %
case c of
  () : begin multiplyB; squaring; multiplyC end;
  (zero,c1) : begin multiplyC; squaring; treat0(c1) end;
  (one,c1)  : begin multiplyB; squaring; treat11(c1) end
end;

procedure naive(c:chain);
case c of
  () : nil;
  (zero,c1) : begin squaring; naive(c1) end;
  (one,c1)  : begin squaring; multiply; naive(c1) end
end;

measure multiplyB : 1;
        multiplyC : 1;
```

```
        multiply  : 1;
        divide    : 1;
        squaring  : 1;
*************************************************************************

Script started on Wed May 25 18:31:42 1988
yquem% maplecaml
   CAML (sun) (V 2-5) by INRIA Fri Jan 22

#load"luo";;

#order:=2;; <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
2 : num

#analyze"chains2";;
Generating functions are ordinary.

Counting generating functions:
  chain=Q(bit)
  valid_chain=chain*one
  bit=zero+one
  zero=z
  one=z

Complexity descriptors:
  tau_treat=tau_treatStart*one+1*chain*one
  tau_treatStart=zero*tau_treat0+one*tau_treat1
  tau_treat0=1*1+1*1+1*zero*Q(bit)+zero*tau_treat0+1*one*Q(bit)+one*tau_treat1
  tau_treat1=1*1+1*1+1*zero*Q(bit)+1*zero*Q(bit)+zero*tau_treat0+1*one*Q(bit)+1*
one*Q(bit)+one*tau_treat11
  tau_treat11=1*1+1*1+1*1+1*zero*Q(bit)+zero*tau_treat110+1*one*Q(bit)+one*tau_t
reat11
  tau_treat110=1*1+1*1+1*1+1*zero*Q(bit)+1*zero*Q(bit)+zero*tau_treat0+1*one*Q(b
it)+1*one*Q(bit)+one*tau_treat11
  tau_naive=1*zero*Q(bit)+zero*tau_naive+1*one*Q(bit)+1*one*Q(bit)+one*tau_naive

Solving generating functions ...
                                  one = z

                                 zero = z

                           valid_chain = Q(2 z) z

                                  bit = 2 z

                                chain = Q(2 z)

Solving complexity descriptors ...
tau_treat0

                                                                    2
    = - (- one Q(bit) + 2 one  zero Q(bit) - 2 - one zero Q(bit) - zero Q(bit)

          2                                .2           3                    2
      - one  Q(bit) + 2 one zero - one  zero + one  Q(bit) + one zero  Q(bit)
```

$$- \text{one}^2)$$

$$1 / (\text{one zero}^2 + \text{one}^2 \text{ zero} + 1 - \text{one zero} - \text{zero} - \text{one})$$

tau_treatStart

$$= - (- 3 \text{ one zero } Q(\text{bit}) + 2 \text{ one}^2 \text{ zero } Q(\text{bit}) - 2 \text{ zero} - \text{zero}^2 Q(\text{bit})$$

$$+ \text{one}^2 \text{ zero } Q(\text{bit}) + 2 \text{ one zero}^2 + \text{one}^3 \text{ zero } Q(\text{bit}) + \text{one zero}^3 Q(\text{bit})$$

$$+ \text{one}^2 \text{ zero} - 2 \text{ one} - 2 \text{ one}^2 Q(\text{bit}) + \text{one}^3 Q(\text{bit}) - \text{one}^2)$$

$$/ (\text{one zero}^2 + \text{one}^2 \text{ zero} + 1 - \text{one zero} - \text{zero} - \text{one})$$

tau_treat11

$$= - (- \text{one zero } Q(\text{bit}) + \text{one}^2 \text{ zero } Q(\text{bit}) + \text{zero}^3 Q(\text{bit})$$

$$+ 2 \text{ one zero}^2 Q(\text{bit}) - \text{zero } Q(\text{bit}) - \text{zero}^2 Q(\text{bit}) - \text{one } Q(\text{bit}) + \text{zero}^2$$

$$+ \text{one zero}^2 + 3 \text{ one zero} - 3)$$

$$/ (\text{one zero}^2 + \text{one}^2 \text{ zero} + 1 - \text{one zero} - \text{zero} - \text{one})$$

tau_treat110

$$= - (- 3 - 2 \text{ one } Q(\text{bit}) - 2 \text{ zero } Q(\text{bit}) + 2 \text{ one zero } Q(\text{bit}) + \text{zero}^2 Q(\text{bit})$$

$$+ \text{zero} + \text{one}^2 Q(\text{bit}) + 3 \text{ one zero} - \text{one}^2 \text{ zero})$$

$$/ (\text{one zero}^2 + \text{one}^2 \text{ zero} + 1 - \text{one zero} - \text{zero} - \text{one})$$

tau_treat1

$$= - (\text{zero}^2 Q(\text{bit}) + 2 \text{ one zero } Q(\text{bit}) - 2 \text{ zero } Q(\text{bit}) - 2 - 2 \text{ one } Q(\text{bit})$$

$$+ 2 \text{ one zero} + \text{one}^2 Q(\text{bit}) + \text{one zero}^2 - \text{one})$$

$$\text{zero}^2 \quad \text{one}^2$$

$$/ \ (\text{one zero} + \text{one zero} + 1 - \text{one zero} - \text{zero} - \text{one})$$

$$\text{tau_naive} = - \frac{Q(\text{bit}) \ (\text{zero} + 2 \ \text{one})}{-1 + \text{zero} + \text{one}}$$

tau_treat

$$= \text{one}$$

$$(3 \ \text{one zero} \ Q(\text{bit}) - 2 \ \text{one}^2 \ \text{zero} \ Q(\text{bit}) + 2 \ \text{zero} + \text{zero}^2 \ Q(\text{bit})$$

$$- \text{one}^2 \ \text{zero} \ Q(\text{bit}) - 2 \ \text{one} \ \text{zero}^2 - \text{one}^3 \ \text{zero} \ Q(\text{bit}) - \text{one zero}^3 \ Q(\text{bit})$$

$$- \text{one}^2 \ \text{zero} + 2 \ \text{one} + 2 \ \text{one}^2 \ Q(\text{bit}) - \text{one}^3 \ Q(\text{bit}) + \text{one}^2$$

$$+ \text{chain one zero}^2 + \text{chain one}^2 \ \text{zero} + \text{chain} - \text{chain one zero}$$

$$- \text{chain zero} - \text{chain one})$$

$$/ \ (\text{one zero}^2 + \text{one}^2 \ \text{zero} + 1 - \text{one zero} - \text{zero} - \text{one})$$

Average case analysis of function treat:

1) Number of arguments of treat of size n is:

$$1/2 \ 2^n + O((1/n)^{\text{infinity}} \ 2^n)$$

2) Total cost of treat on all arguments of size n is:

$$2/3 \ 2^n \ n + \frac{17}{36} \ n \ 2^n + O(\frac{2^n}{n})$$

3) Average cost of treat on arguments of size n is:

$$\text{av_tau_treat_n} := 4/3 \ n + \frac{17}{18} + O(1/n)$$

4) Floating point evaluation :

$$1.33333 \ n + .944444 + O(1/n)$$

Average case analysis of function treatStart:

1) Number of arguments of treatStart of size n is:

$$2^{n} + O((1/n)^{infinity - n})$$

2) Total cost of treatStart on all arguments of size n is:

$$4/3\ 2^{n}\ n + \frac{23}{18}\ 2^{n} + O(\frac{2^{n}}{n}).$$

3) Average cost of treatStart on arguments of size n is:

$$av_tau_treatStart_n := 4/3\ n + \frac{23}{18} + O(1/n)$$

4) Floating point evaluation :

$$1.33333\ n + 1.27778 + O(1/n)$$

Average case analysis of function treat0:
==========

1) Number of arguments of treat0 of size n is:

$$2^{n} + O((1/n)^{infinity - n})$$

2) Total cost of treat0 on all arguments of size n is:

$$4/3\ 2^{n}\ n + \frac{41}{18}\ 2^{n} + O(\frac{2^{n}}{n})$$

3) Average cost of treat0 on arguments of size n is:

$$av_tau_treat0_n := 4/3\ n + \frac{41}{18} + O(\frac{(1/2)^{(-n)}}{(1/2)^{n}\ n})$$

4) Floating point evaluation :

$$1.33333\ n + 2.27778 + O(\frac{(1/2)^{(-n)}}{(1/2)^{n}\ n})$$

Average case analysis of function treat1:
==========

1) Number of arguments of treat1 of size n is:

$$2^{n} + O((1/n)^{infinity - n})$$

2) Total cost of treat1 on all arguments of size n is:

$$4/3\ 2^n\ n + \frac{53}{18}\ 2^n + O\left(\frac{2^n}{n}\right)$$

3) Average cost of treat1 on arguments of size n is:

$$av_tau_treat1_n := 4/3\ n + \frac{53}{18} + O\left(\frac{1}{2^n}\right)$$

4) Floating point evaluation :

$$1.33333\ n + 2.94444 + O\left(\frac{1}{2^n}\right)$$

Average case analysis of function treat11:

1) Number of arguments of treat11 of size n is:

$$2^n + O((1/n)\ 2^{infinity\ n})$$

2) Total cost of treat11 on all arguments of size n is:

$$4/3\ 2^n\ n + \frac{41}{18}\ 2^n + O\left(\frac{2^n}{n}\right)$$

3) Average cost of treat11 on arguments of size n is:

$$av_tau_treat11_n := 4/3\ n + \frac{41}{18} + O(1/n)$$

4) Floating point evaluation :

$$1.33333\ n + 2.27778 + O(1/n)$$

Average case analysis of function treat110:

1) Number of arguments of treat110 of size n is:

$$2^n + O((1/n)\ 2^{infinity\ n})$$

2) Total cost of treat110 on all arguments of size n is:

$$4/3\ 2^n\ n + \frac{53}{18}\ 2^n + O\left(\frac{2^n}{n}\right)$$

3) Average cost of treat110 on arguments of size n is:

$$av_tau_treat110_n := 4/3\ n + \frac{53}{18} + O\left(\frac{1}{2^n}\right)$$

4) Floating point evaluation :

$$1.33333\ n + 2.94444 + O\left(\frac{1}{2^n}\right)$$

Average case analysis of function naive:
==========

1) Number of arguments of naive of size n is:

$$2^n + O((1/n)^{infinity}\ 2^n)$$

2) Total cost of naive on all arguments of size n is:

$$3/2\ 2^n\ n + O((1/n)^{infinity}\ 2^n)$$

3) Average cost of naive on arguments of size n is:

$$av_tau_naive_n := 3/2\ n$$

4) Floating point evaluation :

$$1.50000\ n$$

() : void

#quit();;
A bientot ...
yquem%
script done on Wed May 25 18:36:49 1988

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

REORGANIZACIONES LOCALES EN EL ARBOL-K-D-B. SU EFICIENCIA EN SITUACIONES DINAMICAS

M. DIAZ, O. SANTANA, G. RODRIGUEZ, M. MARTIN
Departamento de Informática y Sistemas
Universidad Politécnica de Canarias
Aptdo. 550, Las Palmas de Gran Canaria, España.

RESUMEN:

En este artículo se introducen diversas reorganizaciones locales para optimizar la ocupación en la actualización de la estructura del árbol K-D-B. Se realiza un estudio experimental en tres fases: creciente, decreciente y estacionaria. En cada fase se compara la evolución de la estructura con reorganizaciones locales y sin ellas, poniendose de manifiesto el aporte de las reorganizaciones locales en la optimización de la ocupación, sin olvidar el incremento en el número de accesos.

0) INTRODUCCION

La estructura de árbol K-D-B [RO81] es una combinación del árbol K-D [BE75], [BE79], [BF79] y del árbol-B [BM72]. Con ella se espera obtener la eficiencia de la búsqueda multidimensional de los árboles K-D y la eficiencia de accesos de los árboles-B. La partición del espacio en los árboles K-D-B, es similar a los árboles K-D: el espacio se parte en dos subespacios basándose en la comparación con algún elemento de un único dominio. Los árboles K-D-B, asi como los árboles-B, son árboles multirramas perfectamente equilibrados, con nodos de tamaño fijo. Cada nodo se almacena como una página por lo que se puede tener un uso eficiente de la memoria secundaria mediante la páginación.

En la sección 1 se define la estructura del árbol K-D-B. Las divisiones de páginas de puntos y páginas regiones se describen en la sección 2. Las reorganizaciones locales que se proponen en este trabajo se presentan en la sección 3. Las secciones 4 y 5 contienen los algoritmos de inserción y extracción que contemplan dichas reorganizaciones. Los resultados del estudio experimental realizado en tres fases: creciente, decreciente y estacionaria [SM87], en los que se pone de manifiesto el aporte de las reorganizaciones locales, se muestran en la sección 6. Por último, la sección 7 contiene las conclusiones y propuestas de investigaciones futuras.

1) ESTRUCTURA DE ARBOL-K-D-B

Existen dos tipos de nodos en el árbol K-D-B:

1. Pàginas regiones o nodos internos: que contienen las regiones y los punteros a otras páginas.

2. Páginas de puntos o celdas: que contienen los puntos.

Propiedades

El siguiente conjunto de propiedades definen la estructura del árbol K-D-B.

1. El árbol K-D-B es un árbol multirrama.

2. Ninguna página región contiene regiones con punteros nulos, ni está vacía.

3. La distancia, en páginas, desde la página raíz a cada una de las páginas de puntos es la misma.

4. En cada página región, las regiones de la página son disjuntas y su unión es una región.

5. Si la página raíz es una página región (puede no existir, o si hay sólo una página en el árbol será una página de puntos), la unión de sus regiones es el dominio total.

6. Si en una página región, la página hijo referida por el puntero de una 'región' es una página región, entonces la unión de las regiones en la página hijo es 'región'.

7. Refiriéndose al punto 6, si la página hijo es una página de puntos, entonces todos los puntos de la página estan contenidos en 'región'.

2) DIVISIONES

Las divisiones en los árboles-K-D-B parten el espacio en dos subespacios, basándose en la comparación con un elemento de un determinado dominio. Se pueden usar varias técnicas para seleccionar el dominio y el elemento en el domino.

Una de ellas es elegirlo cíclicamente. Existen excepciones para esta técnica debido a que a veces se tiene que dividir una página por un elemento determinado (divisiones descendentes).

En general, dado el dominio divisor, el punto de división se elige de tal forma que, en la medida que sea posible, el número de puntos o regiones en cada nueva página sea el mismo, con ello se minimiza el número de veces que se ha de dividir una página. Sin embargo, en algunos casos el dominio de división debe reelegirse.

2.1 DIVISION DE UNA REGION POR EL ELEMENTO xi

Sea la región I1xI2x...xInd. Si xi no pertenece a Ii, la región no cambia con la división. En otro caso, sea Ii=[mini(i),maxi(i)]; dividiendo la región resultan dos nuevas regiones:

1. I1 x...x [mini(i),xi) x...x Ind
2. I1 x...x [xi,maxi(i)) x...x Ind.

La región (1) se llama región izquierda y la (2) región derecha.

Si xi no pertenece a Ii, ya que xi<mini(i), se dice que la región permanece a la derecha de xi; si xi>=maxi(i), la región permanece a la izquierda de xi. Se dice que un punto (y1,y2,...,ynd) permanece a la izquierda de xi si yi<xi, y permanece a la derecha en otro caso.

2.2 DIVISION DE UNA PAGINA DE PUNTOS POR xi

La página de puntos se divide por xi creando dos nuevas páginas de puntos, la página derecha y la página izquierda; entonces, se transferirán todos los puntos a las páginas izquierda o derecha, dependiendo de si permanecen a la izquierda o a la derecha de xi; y luego se borrará la página vieja.

2.3 DIVISION DE UNA PAGINA REGION POR xi

Una página región se divide por xi creando dos nuevas páginas llamadas también página izquierda y página derecha; llenandolas con regiones derivadas de la página región vieja; y borrando la página vieja.

3) REORGANIZACIONES LOCALES

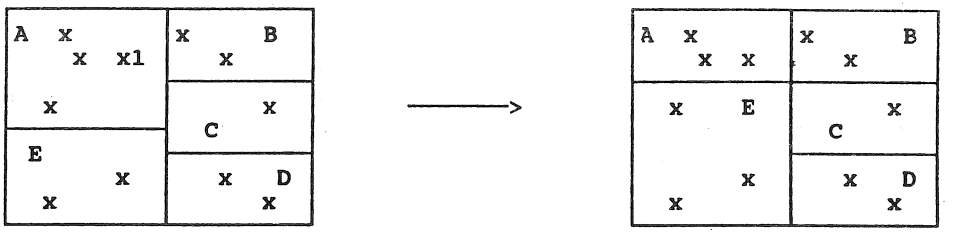
Como mejora de la ocupación de las páginas de puntos se introdujeron los procedimientos de rotación y recombinaciones.

Previamente se define el concepto de regiones unibles: si la unión de dos o más regiones es una región, se dice que las regiones se pueden unir.

3.1 ROTACION

Si al insertar un punto, la página de puntos se sobrecarga, antes de partirla se comprueba si la región que contiene a esos puntos se puede unir con otra región que no este llena, si es asi se redistribuyen los puntos de ambas. Este tipo de rotación, en el caso de poderse realizar, evita la creación de una página de puntos y por lo tanto, la posible sobrecarga de páginas regiones.

Por ejemplo, para tamaño máximo de página de puntos, tmpp = 3, al insertar el punto x1 resultará :

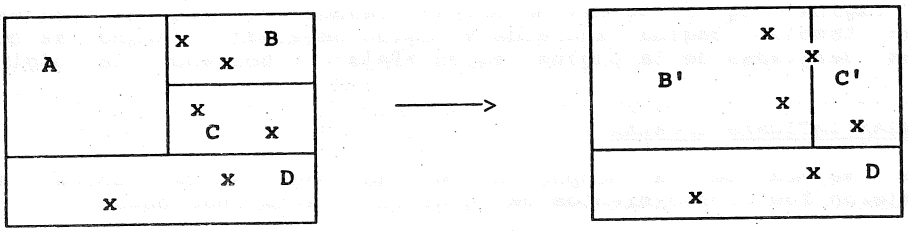


3.2 RECOMBINACIONES DE PAGINAS VACIAS

La division descendente provocada por la partición de una página región puede originar páginas vacías. Para mejorar la ocupación se borran dichas páginas, debiendo de reestructurar las regiones de la página región padre. Se proponen dos métodos:

3.2.1 RECOMBINACION POR UNION

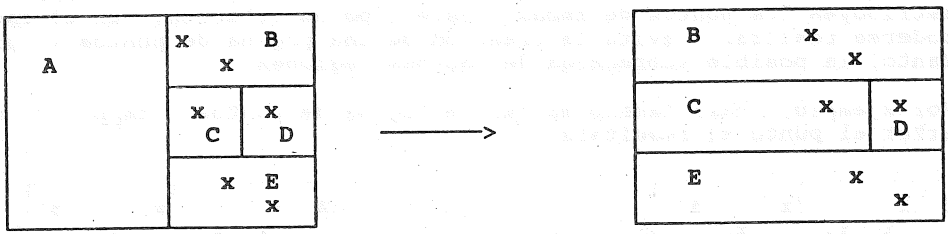
La región que señalaba a una página de puntos vacía se une con otra u otras adyacentes en una dimensión, formando una región mayor, que señalará a una página de puntos que contendrá todos los puntos correspondientes a las páginas implicadas en la unión. Si esta página se sobrecarga se divide como se muestra en la siguiente figura, para tmpp=3.



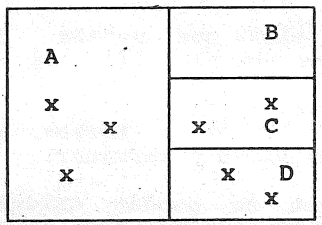
Este tipo de recombinación se utiliza sólo en las páginas regiones del último nivel.

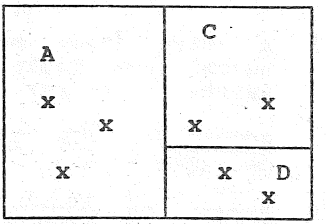
3.2.2 RECOMBINACION POR EXTENSION

Se reparte la región entre otra u otras adyacentes, aumentando el tamaño de dichas regiones a lo largo de una dimensión.



Tanto la recombinación por unión como por extensión dan el mismo resultado si se utiliza una única región adyacente.





3.2.3 RECOMBINACION DE UNA PAGINA DE PUNTOS VACIA

Sea PP la página de puntos vacía y PR la página región que contiene la región que señala a PP. El algoritmo para borrar PP es el siguiente:

1. Si el número de regiones de PR es 1, borrar PP, poner el número de regiones de PR a 0, ir a RECOMBINAR UNA PAGINA REGION VACIA y terminar.

2. En otro caso, buscar la región 'i' de PR que señale a PP. Buscar una región 'j' que se pueda unir con la 'i'. Si no existe, ir al paso 4.

3. RECOMBINAR POR EXTENSION las regiones 'i' y 'j'. Borrar PP, decrementar en 1 el número de regiones de PR y terminar.

4. Buscar un conjunto de regiones tales que se puedan extender sobre la región 'i'. Si no se encuentran, ir al paso 6.

5. RECOMBINAR POR EXTENSION las regiones seleccionadas, decrementar el número de regiones de PR en 1 y terminar.

6. Unir todas las páginas de puntos a las que señalan todas las regiones de PR en una única página de puntos, y todas las regiones de PR en una sóla región (esto siempre se puede hacer, ya que la unión de todas las regiones de una página región forman una región).

7. Dividir la página de puntos hasta que no se produzca una situación de sobrecarga.

8. Terminar.

3.2.4 RECOMBINACION DE UNA PAGINA REGION VACIA

Se PR la página región que está vacía, y PADRE la página región que señala a PR. El algoritmo para borrar PR es el siguiente:

1. Si el número de regiones de PADRE es 1, borrar PR, poner el número de regiones de PADRE a 0 e ir a RECOMBINAR UNA PAGINA REGION VACIA.

2. En otro caso, buscar la región 'i' de PADRE que señala PR. Buscar una región 'j' que se pueda unir con la 'i'. Si no existe, ir al paso 4.

3. RECOMBINAR POR EXTENSION las regiones 'i' y 'j'. Borrar PR, decrementar en 1 el número de regiones de PADRE y terminar.

4. Buscar un conjunto de regiones tales que se puedan extender sobre la región 'i'. Si no es posible ir al paso 6.

5. RECOMBINAR POR EXTENSION las regiones seleccionadas, decrementar el número de regiones de PADRE en 1 y terminar.

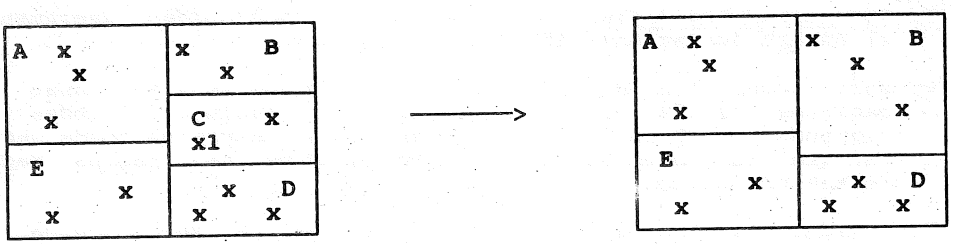
6. Unir todos los puntos que estén por debajo de PADRE en una única página, y dejar en cada nivel una página región con una única región. Resolver cualquier situación de sobrecarga.

7. Terminar.

3.3 RECOMBINACION DE PAGINAS DE PUNTOS NO VACIAS

Al extraer un punto, de una página de puntos no vacía, se comprueba si la región que contiene a ese punto se puede unir con otra región, si es asi se recombinan en una unica región si la suma de los puntos de las dos páginas de puntos no excede de un cierto umbral. En realidad, se trata de unir dos páginas de puntos sin que se produzca una sobrecarga (umbral <= tmpp).

Por ejemplo, para umbral = tmpp = 3, al borrar el punto x1 resultará :



4) INSERCION

Para insertar un punto en un árbol K-D-B, se parte del nodo raiz y se recorre el árbol hasta llegar a la página de puntos donde hay que insertar el punto. Una vez allí se comprueba si dicho punto ya existe, en cuyo caso, no se realiza la inserción.

Una vez realizada la inserción, si la página de puntos queda sobrecargada, se intenta hacer una rotación, si no se realiza la división de dicha página. Esta división origina la división de la región que le señala. Si por esta causa se sobrecarga la página

región, también habrá que dividirla, tal como se vio en 2.3.

IMPLEMENTACION DEL PROCESO DE INSERCION

El algoritmo para insertar un 'punto' es el siguiente:

I1: Si Raíz es nula, crear una página de puntos conteniendo el 'punto', poner su dirección en raíz y terminar.

I2: En otro caso, buscar la página de puntos en la que se debe añadir 'punto'. Si 'punto' ya está en la página no insertalo, dar un mensaje de error y terminar.

I3: Añadir 'punto' a la página de puntos. Si la página no se sobrecarga, terminar.

Si se sobrecarga:

I3.1: Si se trata de la página raíz, dividirla en dos nuevas páginas de puntos, crear una página región, con dos regiones que señalen a dichas páginas, que será la nueva raíz del árbol y terminar.

I3.2: Si no es la raíz, se comprueba si se puede ROTAR, y en tal caso se rota y termina. Si no es posible, se divide la página de puntos, y en la página región PADRE se divide la región que señala a la página de puntos en cuestión.

I4: Si PADRE se sobrecarga. Se elige un dominio(i) y un elemento xi de este dominio tal que PADRE se divide por xi resultando dos nuevas páginas regiones izquierda y derecha. Si alguna de las dos queda sobrecargada, volver al punto I4.

I5: Si la página PADRE era la página raíz, ir a (I6). En otro caso, obtener la página que sea la página padre de la página dividida. Reemplazar la región vieja por las dos nuevas regiones (región izquierda y región derecha), que se obtienen dividiendo la región vieja por xi. Si esto causa sobrecarga en una página, repetir desde I4, si no ir a I7.

I6: Crear una nueva página región conteniendo dos regiones y sus respectivos punteros hacia las nuevas páginas regiones izquierda y derecha, respectivamente. Colocar en esta nueva página región la raíz del árbol.

I7: Si en el proceso de división se ha quedado alguna página vacía, borrarla, RECOMBINAR las regiones y terminar.

5) EXTRACCION

La estructura del árbol K-D-B propuesta por J. T. Robinson no excluye las páginas de puntos vacías, y no exige nada sobre la utilización del almacenamiento. El algoritmo de extracción que se presenta en este trabajo elimina dichas páginas y mejora la ocupación utilizando el procedimiento de recombinación de páginas no vacías.

IMPLEMENTACION DEL PROCESO DE EXTRACCION

El algoritmo para extraer un 'punto' del árbol es el siguiente:

E1. Obtener la página de puntos donde debe de estar el punto a extraer. Si no existe, dar un mensaje de error y terminar.

E2. Extraer el punto de la página de puntos.

E3. Si la página de puntos queda VACIA, eliminarla, RECOMBINAR y terminar. Si es NO VACIA intentar una RECOMBINACION con otra página de puntos, si la suma de los puntos no supera un cierto umbral y terminar.

6) DESCRIPCION DEL ESTUDIO EXPERIMENTAL

El rendimiento de una estructura de datos esta determinado por dos criterios: el tiempo de procesamiento y la utilización de memoria.

Se han realizado varias simulaciones para estudiar el rendimiento del árbol K-D-B, midiendo estos criterios en base a los siguientes parámetros:

a) número de páginas regiones totales.
b) número de páginas de puntos totales.
c) número de páginas regiones accedidas.
d) número de páginas de puntos accedidas.
e) ocupación de las páginas regiones.
f) ocupación de las páginas de puntos.
g) altura del árbol.

De acuerdo con el caracter dinámico de la estructura el estudio se ha realizado en las siguientes fases:

1) Estado creciente. Inserciones repetidas.

2) Estado estacionario. Inserciones y extracciones manteniendo aproximadamente constante el número de puntos.

3) Estado decreciente. Extracciones repetidas.

Todos los números aleatorios utilizados en los experimentos se han obtenido a partir de un generador de números pseudoaleatorios de distribución continua uniforme en [0,1). Para cada punto se generaron nd valores aleatorios, siendo nd la dimensión del espacio.

Una cuestión previa es el estudio comparativo de la estructura para los dos tipos de recombinaciones de páginas de puntos vacías: por unión o por extensión. Se construyeron dos árboles de dimensión 2, tamaño máximo de las páginas de puntos, tmpp=20, y tamaño máximo de las páginas regiones, tmpr=10; obteniéndose resultados similares. Para realizar el estudio experimental se eligió la recombinación por extensión porque es más rápida, al no tener que comprobar la condición de que la unión de todas las regiones formen una región, y en ningún caso provoca sobrecarga.

ESTADO CRECIENTE

El estado creciente corresponde a la fase de crecimiento del árbol a través de sucesivas inserciones.

Se construyeron árboles de 10000 puntos. Los parámetros estadísticos anteriormente descritos, se midieron en intervalos regulares de 100 inserciones.

Inicialmente se comprueba el efecto que produce la introducción del procedimiento de rotación, para ello se construyen dos árboles con nd=4, tmpp=25 y tmpr=13.

- El número de páginas regiones y páginas de puntos en el árbol construído CON ROTACIONES fue de 65 y 486, respectivamente. Y en el que se construyó SIN ROTACIONES fue de 113 y 748, respectivamente.

- La ocupación de las páginas de puntos presenta un notable incremento, figura 1. El efecto que produce la rotación en las páginas regiones es que pospone la partición de dichas páginas, esto da lugar a que la figura 2 presente intervalos en los que la diferencia es notable.

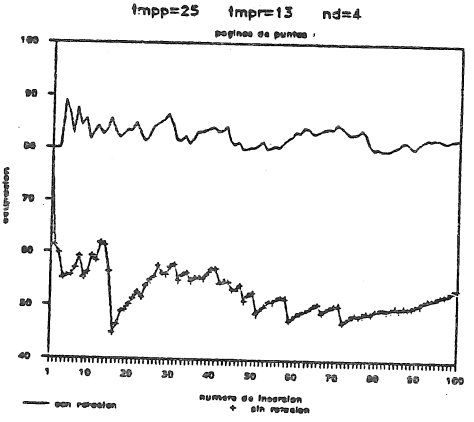


Figura 1

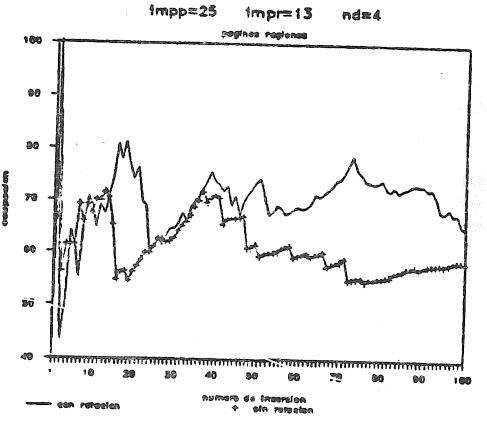


Figura 2

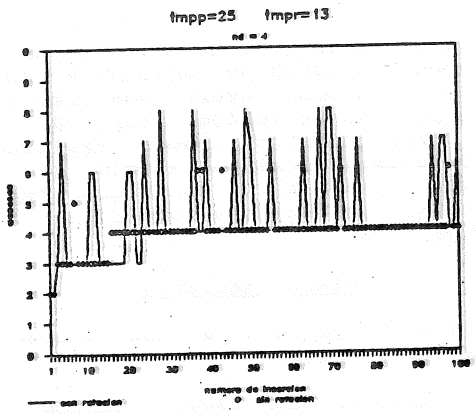


Figura 3

- El aumento del número de accesos es debido a que la ocupación es mucho mayor y las sobrecargas son por consiguiente más frecuentes. Estas sobrecargas se pueden resolver mediante rotaciones o particiones. Una rotación requiere tres accesos y una partición de una página de puntos que no provoca sobrecarga en una página región da lugar a dos accesos. En la práctica existen intentos fallidos de rotación, que conllevan uno o ningún acceso, figura 3.

Debido a estos resultados, todos los experimentos presentados a continuación se llevaron a cabo utilizando la rotación.

Se construyeron 12 árboles, teniendo en cuenta la variacion de la dimensionalidad, nd, el tamaño máximo de las páginas de puntos, y el tamaño máximo de las páginas regiones, para los siguientes valores:

nd = 2, 4, 6

tmpp-tmpr = 10-5, 15-8, 20-10, 25-13

obteniéndose que:

- La ocupación de las páginas de puntos aumentaba al aumentar tmpp y tmpr, estando en todos los casos entre un 65% y un 85 %, como se puede observar en las figuras 4 y 5, en las que cada curva representa una dimensión.

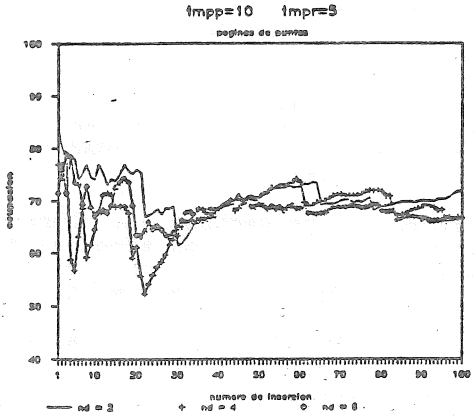


Figura 4

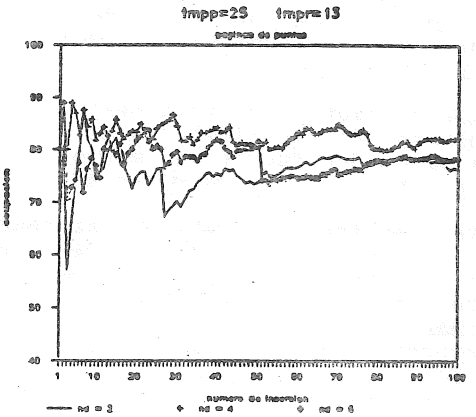


Figura 5

- La ocupación de las páginas regiones se mantiene alrededor de un 70%, el incremento de tmpp y tmpr provoca una mayor variabilidad, como se observa en las figuras 6 y 7.

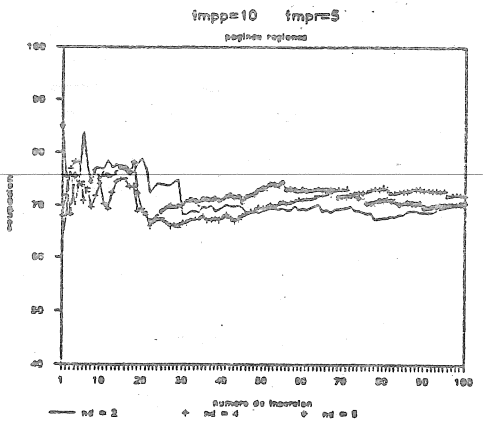


Figura 6

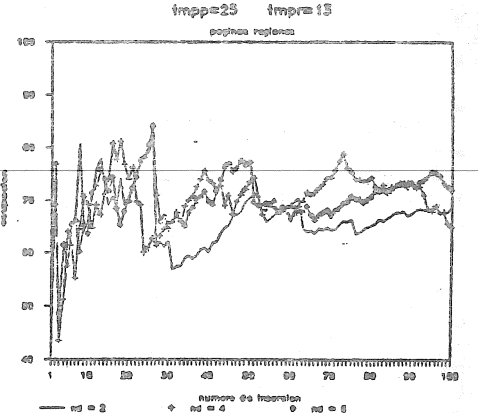


Figura 7

- La dimensonalidad apenas influye en la ocupación, manteniéndose en la banda antes mencionada, dependiendo de tmpp y tmpr.

ESTADO ESTACIONARIO

El estado estacionario corresponde a la fase de actualización del árbol, a través de la selección de inserciones y extracciones de forma aleatoria.

A partir de un árbol de 10000 puntos construido en el estado creciente, se insertaron 2000 nuevos puntos y se extrajeron 2000 de los ya existentes.

Los parámetros anteriormente descritos, se midieron en intervalos regulares de 100 inserciones + extracciones. Las reorganizaciones locales usadas fueron las mismas que en el estado creciente, añadiendo las recombinaciones de páginas de puntos no vacías.

Se realizaron pruebas manteniendo fijos los parámetros nd, tmpp y tmpr, y variando el umbral de recombinación para ver los efectos de este parámetro en la ocupación. Los valores dados a dicho umbral fueron los siguientes: 100%, 80%, 60% y 0% del tamaño máximo de las páginas de puntos. Como se observa en las figuras 8 y 9, la disminución del umbral de recombinación produce un efecto negativo, más acusado para umbrales altos, en el porcentaje de ocupación de las páginas de puntos y opuesto en las páginas regiones.

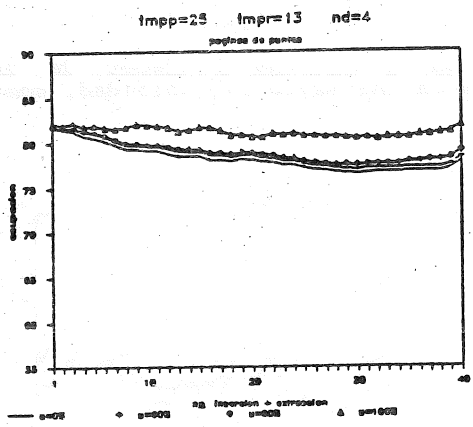


Figura 8

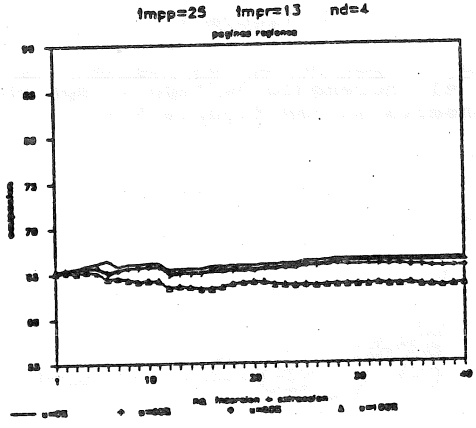


Figura 9

- La diferencia de accesos para umbrales del 100% y del 0% se debe al uso de las recombinaciones de páginas de puntos no vacías. Dichas recombinaciones pueden dar lugar a: 2 accesos en el caso de que se realice la recombinación, 1 acceso si la suma de puntos supera el umbral y ningún acceso en el caso de que no exista la región adyacente, figura 10.

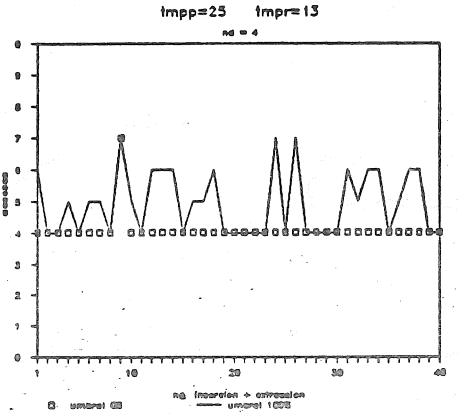


Figura 10

La variación de la dimensión no presenta diferencias significativas.

ESTADO DECRECIENTE

El estado decreciente corresponde a la fase de disminución del árbol a través de sucesivas extracciones.

A partir del árbol final del estado creciente se realizaron las extracciones de los 10000 puntos, midiendo los mismos parámetros y en iguales intervalos que en el estado creciente. Al igual que en el estado estacionario, se relizaron pruebas para distintos umbrales de recombinación de páginas de puntos no vacías, obteniéndose los resultados que se muestran en las figuras 11, 12 y 13.

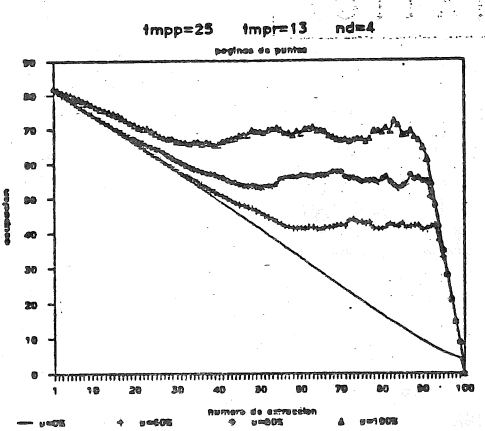


Figura 11

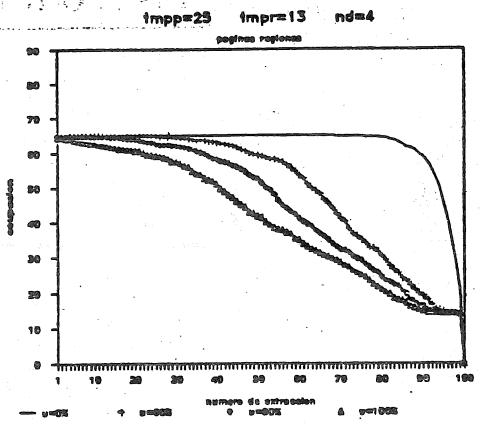


Figura 12

- A umbrales altos la ocupación de páginas de puntos presenta una suave caida inicial y un mantenimiento posterior a lo largo del proceso. Tan solo en la extracción de los últimos puntos aparece un descenso brusco. Esto hace pensar en lo adecuado del sistema que permite mantener un alto nivel de ocupación hasta alcanzar las últimas extracciones. La ocupación de las páginas regiones presenta un comportamiento contrario a la de las páginas de puntos, donde la caída brusca final se pone de manifiesto en la ocupación de las páginas regiones cuando el umbral de las páginas de puntos es del 0%.

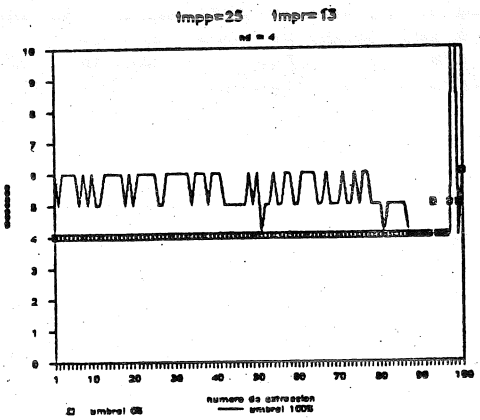


Figura 13

- Los accesos que se muestran en la figura 13, para umbral 100%, estan provocados por la recombinación de páginas de puntos no vacías, salvo en las últimas extracciones.

La variación de la dimensión no presenta diferencias

PEDECIBA
INFORMATICA

significativas.

CONCLUSIONES

El esquema de mantenimiento que se propone para el árbol K-D-B que describió J. T. Robinson, elimina páginas de puntos vacías y la resolución de sobrecargas se realiza con: particiones, tanto de páginas regiones como de páginas de puntos; rotaciones, sólo en estas últimas. Se incorpora la operación de recombinación para páginas de puntos no vacías.

El presente estudio experimental en la fase creciente introduce las rotaciones, como alternativa a las particiones, y recombinaciones de páginas vacías consiguiendo mejorar la ocupación en función del crecimiento del tamaño máximo de las páginas de puntos y de las páginas regiones, llegando a un 85% y un 70% respectivamente, para tmpp=25 y tmpr=13.

La extracción sucesiva de puntos conduce a un descenso en la ocupación de páginas de puntos, para evitarlo se propone la recombinación de páginas de puntos no vacías.

En la fase estacionaria la combinación de rotaciones y recombinaciones para umbral 100% consigue mantener la ocupación.

La fase decreciente muestra que la ocupación presenta un suave descenso inicial y un mantenimiento posterior, hasta casi las últimas extracciones, si se utilizan recombinaciones a umbrales altos.

Las operaciones de eliminación de páginas de puntos vacías, la rotación como alternativa a la partición y la recombinación de páginas de puntos no vacías, si bien aumentan la ocupación, se ve incrementado el número de accesos.

Nuestra línea de investigación continúa estudiando la respuesta del árbol K-D-B a los distintos tipos de interrogaciones: parcial, en rango y los m-vecinos más próximos.

REFERENCIAS

[BE75] BENTLEY, J.L.
Multidimensional binary search trees used for associative searching.
Comm. ACM 18, 9 (1975), 509-517.

[BE79] BENTLEY, J.L.
Multidimensional binary search trees in database applications.
IEEE Trans. Software Eng, SE-5, 4 (1979), 333-340.

[BF79] BENTLEY, J.L. AND FRIEDMAN, J.H.
Data structures for range searching.
Computing Surverys 11, 4 (1979), 397-409.

[BM72] BAYER, R. AND MCCREIGHT, E.
Organization and maintenance of large ordered indexes.
Acta informática 1, 3 (1972), 173-189.

[RO81] ROBINSON, J.T.
The K-D-B-Tree: A search structure for large multidimensional dynamic indexes.
Departament of Computer Science, Carnegie-Mellon University
Pittsburyh, Pennsylvania 15213, 1981.

[SM87] SANTANA O., MAYOR O., DIAZ M., LOPEZ G.
Comportamiento del árbol-BD en las fases creciente, decreciente y estacionaria.
E.U.Informática, Universidad Politécnica de Canarias.
España 1987.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

EL ARBOL MULTIDIMENSIONAL EQUILIBRADO EN ALTURA: INFLUENCIA DE SU COMPORTAMIENTO DINAMICO EN LAS RECUPERACIONES EXACTA, EN RANGO Y DE LOS VECINOS MAS PROXIMOS.

O. SANTANA, M. DIAZ, Z. HERNANDEZ, J.C. DEL PINO
Departamento de Informatica y Sistemas
Universidad Politecnica de Canarias
Aptdo. 550, Las Palmas de Gran Canaria, España.

Resumen:

En este trabajo se presenta un estudio empirico del comportamiento dinamico de la estructura de arbol multidimensional equilibrado en altura [VV84], analizando los parametros fundamentales que determinan su respuesta ante inserciones y extracciones continuadas y ante una fase estacionaria. Asimismo, se estudia su efectividad frente a los árboles quintarios [LW80,SM87] en atender a las interrogaciones exacta y en rango, y su comportamiento en la busqueda de los vecinos mas proximos.

0- INTRODUCCION

El árbol multidimensional equilibrado en altura, KDHBT [VV84], es una generalizacion del árbol AVL [AV62]. Las principales caracteristicas de esta estructura son su eficiente uso de menoria, y su capacidad dinamica de respuesta a las inserciones y extracciones mediante sencillos reequilibrados. La inserción de un dato en la estructura requiere en el peor de los casos una sola operación de reestructuración. Pero estas ventajas le hacen perder eficiencia en determinadas búsquedas con respecto a otras estructuras similares, aunque estáticas o con un alto coste en las operaciones de inserción y extracción o con una elevada demanda de memoria. Aunque han sido propuestas gran número de otras generalizaciones, los árboles KDHBT son una estructura atractiva.

En la seccion 1 se describe la estructura KDHBT como un estado mas avanzado del arbol de busqueda k-dimensional, KDT. De especificar sus condiciones de equilibrio y las acciones a tomar, reequilibrados, cuando se produzca una perturbacion del mismo, a fin de reestablecerlo, se encarga la seccion 2. En la seccion 3 se expone como se llevan a cabo las operaciones que producen modificaciones estructurales en el arbol, asi como las operaciones que deben realizarse para garantizar la estructura KDHBT. Para valorar el comportamiento de la estructura en inserción y extracción así como ante los distintos tipos de interrogaciones se realizaron pruebas estadísticas midiendo los parámetros indicativos del coste de estas operaciones. En la seccion 4 figuran las pruebas realizadas con inserciónes y extracciónes: Fase creciente (comportamiento durante una inserción continuada), Fase estacionaria (comportamiento con inserción y extración aleatoria), Fase decreciente (extracción continuada). En la seccion 5 se comparan las realizaciones con que se llevan a cabo las interrogaciones exacta y en rango en el arbol Quintario [LW80 y SM87] y en el KDBHT. La busqueda de los M-vecinos mas proximos se introduce en la seccion 6; para a continuacion estudiar su conducta en la seccion 7 mediante un proceso de simulacion. En la seccion 8 se presentan las conclusiones de este trabajo.

1- DESCRIPCION DE LA ESTRUCTURA

1.1 ARBOL KDT

Un fichero k-dimensional o fichero-k es un conjunto de N registros, en el que cada registro está identificado por una k-tupla ordenada (Xk,...,X1) de valores que se llaman claves o atributos.

Un árbol de búsqueda k-dimensional, T, con k ≥ 1, para representar un fichero-k, F, está definido como: T=(u,Ti,Tc,Td), donde u es la raíz del árbol que guarda una clave, de valor s en la dimensión k; Ti, Tc, y Td se denominan respectivamente, subárbol izquierdo, subárbol central, y subárbol derecho de u. Ti, es un árbol de búsqueda k-dimensional para todos los registros en F en los que sus claves en la dimesión k son menores que s; Tc es un árbol de búsqueda de dimensión (k-1) representando un subespacio de T cuyos registros están identificados por la (k-1)-tupla ordenada (Xk-1,...,X1), tomada de los registros en F para los que su clave Xk es igual a s; Td, el subárbol derecho, es un árbol de búsqueda de dimensión k para todos los registros en F en los que sus claves en la dimensión k son mayores que s.

Sea un nodo R de altura h1 que tiene un hijo izquierdo directo P, con altura h2 (si h2 = 0 implica que el subárbol está vacío). Entonces se dice que R es el padre derecho directo de P; además, los nodos no existentes en el camino entre R y P en las altura h2+1,...,h1-1 también se llaman hijos directos de R (se consideran nodos no existentes, a todos los nodos que debieran existir entre dos nodos enlazados, con la posibilidad de que el receptor del enlace sea nulo, para que entre ellos existan nodos a todas las alturas posibles). Dos hijos izquierdos o derechos no existentes en las alturas h y h+1 constituyen un par (de nodos) n.e. (no existentes) a altura h.

1.2 ARBOL KDHBT

Sea P, un nodo en un árbol de búsqueda k-dimensional, que tiene algun par n.e de hijos izquierdo o derecho directos a altura h1. Sea T1 de altura h2 el subárbol central de P, y T2 de altura h3 el subárbol central del padre izquierdo o derecho, directo o indirecto de P (si existe). T2 y T1 son, respectivamente, los subárboles adyacentes izquierdo y derecho del par n.e. El par n.e. se dice soportado directamente por T1 si h2 ≥ h1, y se dice soportado indirectamente por T2 si h2 < h1 y h3 ≥ h1. Se dice que el par n.e. está soportado si lo está directa o indirectamente.

Un árbol k-dimensional equilibrado en altura o kDHBT, con k >= 1, representa un fichero-k, F, mediante un árbol de búsqueda k-dimensional que almacena F, con la restricción de que todo par n.e. en el árbol está soportado.

2- OPERACIONES DE REEQUILIBRADO. ROTACIONES

La condición de equilibrio en altura de un nodo P, de altura h se dice que es "r" si la altura de su subárbol central es h-1, si no; la condición de equilibrio será:

"=" si la altura de su subárbol izquierdo es igual a la de su subárbol derecho y es h-1;
"-" si su subárbol izquierdo es más alto que su subárbol derecho;
"+" si su subárbol derecho es más alto que su subárbol izquierdo;

Si P tiene un par n.e., f, de hijos directos izquierdos o derechos de altura h, el cual es el unico posible par n.e. no soportado en el subárbol que tiene como raíz P, asi como en el camino entre P y la raíz del arbol. Entonces, se dice que P tiene una violación de estructura izquierda o derecha, respectivamente, o simplemente una violación de estructura, si f no está directamente soportado. Se dice que f es el par n.e. asociado a la violación de estructura. La violación de estructura es fuerte si f tampoco está soportado indirectamente. La violación de estructura se dice que es especial si la altura del subárbol izquierdo o derecho o el subárbol central de P es h-1. Un nodo P en T se dice que tiene una doble violación de estructura a altura h si P1, un hijo directo o indirecto izquierdo de P, tiene una fuerte violación de estructura izquierda a altura h, y P2, un hijo directo o indirecto derecho de P, tiene también una fuerte violación de estructura a altura h. Los pares n.e. asociados a la fuerte violación de estructura de P1 y P2, se dice que son los pares n.e. asociados de una doble violación de estructura.

Las operaciones de reequilibrado se usan para corregir una violación de estructura. Sea T un kDHBT con un nodo P de altura h que tiene una violación de estructura a altura h-2. Sea su par n.e. asociado, el único par n.e. en el árbol que puede no estar soportado. En tal situación, se aplica en el nodo la rotación simple cuando la condición de equilibrio de Q, el hijo de P opuesto al par n.e. es "r", "=", o igual a la de P. La doble rotación tipo A se aplica cuando no se aplica la simple y la violación de estructura es especial o la altura del subárbol de Q mas proximo al par n.e. es h-3. La doble rotación tipo B se aplica en los demás casos.

3- INSERCION Y EXTRACCION

La inserción se realiza buscando (de la misma forma que en un árbol binario) para cada dimensión el valor del atributo del registro a insertar para esa dimensión, si se encuentra se desciende por el hijo central, repitiendose el proceso anterior para la siguiente dimensión. Cuando se alcanza un nodo terminal (enlace nulo) en una dimensión dada, se creará el nodo para esta dimensión y las siguientes uniendolos por el subárbol central, si se encuentra la clave en la última dimensión entonces es que el registro ya existía. Después de creados los nodos del registro insertado, se comienza el ascenso en el que se comprueba si en el nodo actual tras la inserción ha variado la altura del hijo por el que se descendio, entonces se actualiza los campos altura y condición

y se comprueba si necesita un reequilibrado, en cuyo caso se realiza el tipo de rotación apropiado según las siguientes condiciones: Si la condición del subárbol mayor es "r", "=" o la misma que la del nodo desequilibrado, se realiza una rotación simple, si no, siendo h la altura del nodo desequilibrado, si la altura de alguno de los hijos del nodo desequilibrado, después de una hipotetica aplicación de la doble rotación A, tiene una altura de "h-3", se aplica una rotación doble A; si no, se aplica una rotación doble B.

En la extraccion, se desciende por el árbol en busca del registro a extraer. Cuando se encuentra el nodo de la última dimensión este se elimina y se asciende eliminando los nodos pertenecientes al registro que no forman parte de otro. El último nodo a eliminar será el que se encuentre en un nivel con varios nodos (excepto cuando el árbol no tenga sino un registro), para eliminar este nodo existen distintas posibilidades según tenga hijos o no. Si tiene los dos hijos (izquierdo y derecho) se busca el predecesor y se sustituye por él. Si no, simplemente se elimina y se sustituye por su hijo si lo hubiera. En este proceso se debe verificar si la disminución de altura provoca algún desequilibrio en cuyo caso se corregirá. Seguidamente se continúa el ascenso arreglando las posibles violaciones de estructura debidas a la disminución de altura. Si esta se ha producido en un hijo central se examina la posible doble violación, corrigiendola si la hubiera.

4- PRUEBAS DE COMPORTAMIENTO DE LA ESTRUCTURA

Las pruebas realizadas para estudiar el comportamiento normal de la estructura son las siguientes: Fase Creciente, estudio de la estructura durante la inserción; Fase Estacionaria, inserción y extracción aleatoria; Fase Decreciente, comportamiento en extracción. En estas pruebas se midió el porcentaje de rotaciones realizadas.

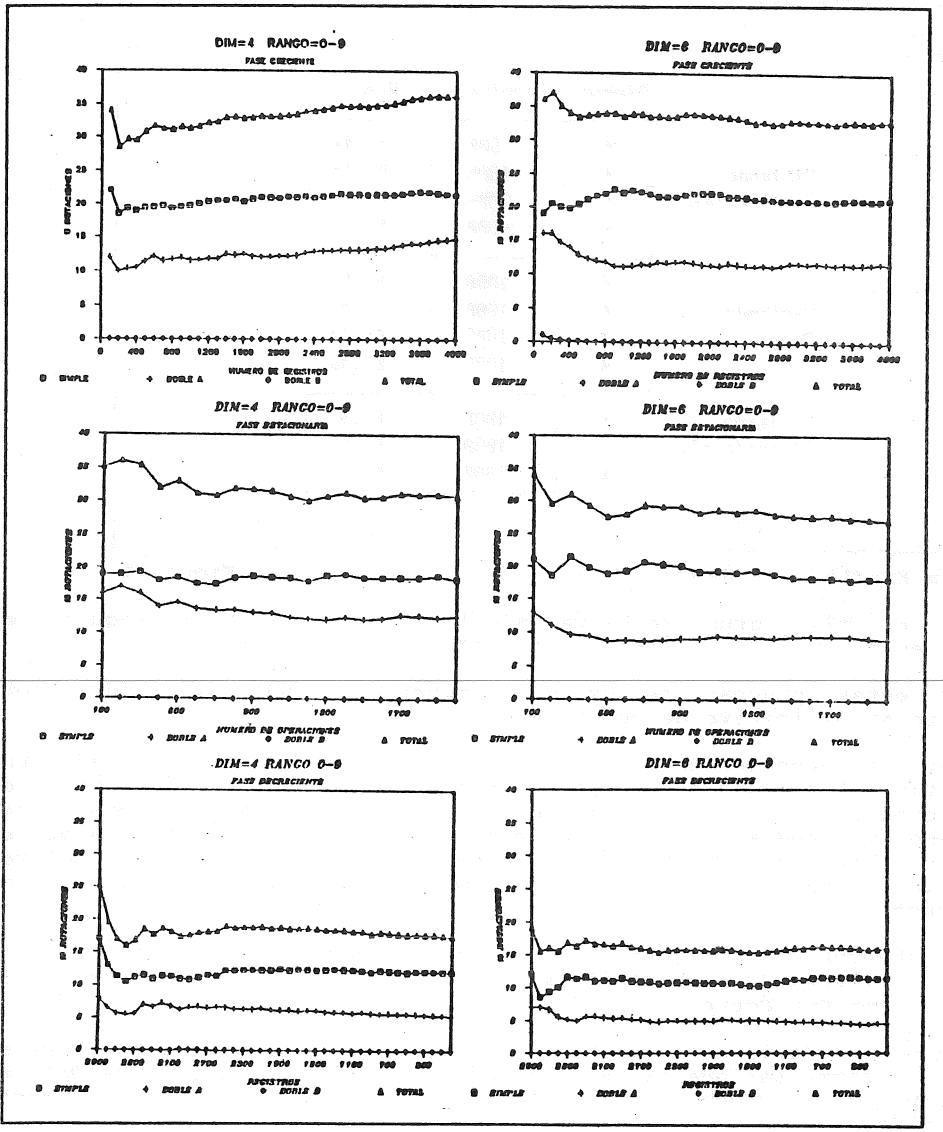
Se realizaron pruebas para dimensión 4 y 6, insertando 4000 registros en la Fase Creciente, despues insertando y extrayendo 1000 registros aleatorias en la Fase Estacionaria y finalmente extrayendo sucesivamente los 4000 registros de forma aleatoria en la Fase Decreciente. El rango de claves usados en todas las pruebas fue de 0 a 9.

La figura 1 muestra los resultados de rotaciones en cada fase. Los valores se calcularon como [rotaciones*100 / operaciones(I+E)].

Se puede apreciar que la diferencia de una dimensión a otra no es excesiva, aunque con tendencia a disminuir el número de rotaciones al aumentar la dimensión. En las tres fases el número de rotaciones casi permanece constante, aunque hay diferencias entre ellas. La fase decreciente es la que menos rotaciones necesita, al igual que ocurre en los AVL, esto es debido a que es más dificil desequilibrar un árbol al extraer un registro que al insertarlo. Esta es la razón por la que en la fase estacionaria se necesiten menos rotaciónes que en la creciente, ya que en ella se intercalan aleatoriamente inserciones y extracciones.

El comportamiento de los diferentes tipos de rotaciones, parece ser el previsto: la rotación simple es la que se produce más frecuentemente,

debido a las condiciones poco restrictivas para su aplicación, en contraste con los tipos dobles, en los que, sobre todo en el tipo B, su aparición viene muy condicionada a unas restricciones que en muy pocos casos se dan.



Rotaciones en las tres fases para DIM=4,6. Figura 1

5- BUSQUEDA EXACTA Y EN RANGO, COMPARACION CON LOS ARBOLES QUINTARIOS

Las pruebas realizadas para las búsquedas exactas y en rango se realizaron variando la Cardinalidad, Dimensionalidad y Rango de claves. En la figura 2 se muestran las pruebas realizadas variando los parámetros indicados.

	Dimenci.	Cardinali.	Rango
Variando Cardinalidad	4	500	0..14
	4	1000	0..14
	4	2000	0..14
	4	4000	0..14
Variando Rango	4	1000	0..9
	4	1000	0..11
	4	1000	0..14
	4	1000	0..16
Variando Dimensionali.	4	1000	0..9
	6	1000	0..9
	8	1000	0..9

Búsqueda Exacta y en Rango. Figura 2

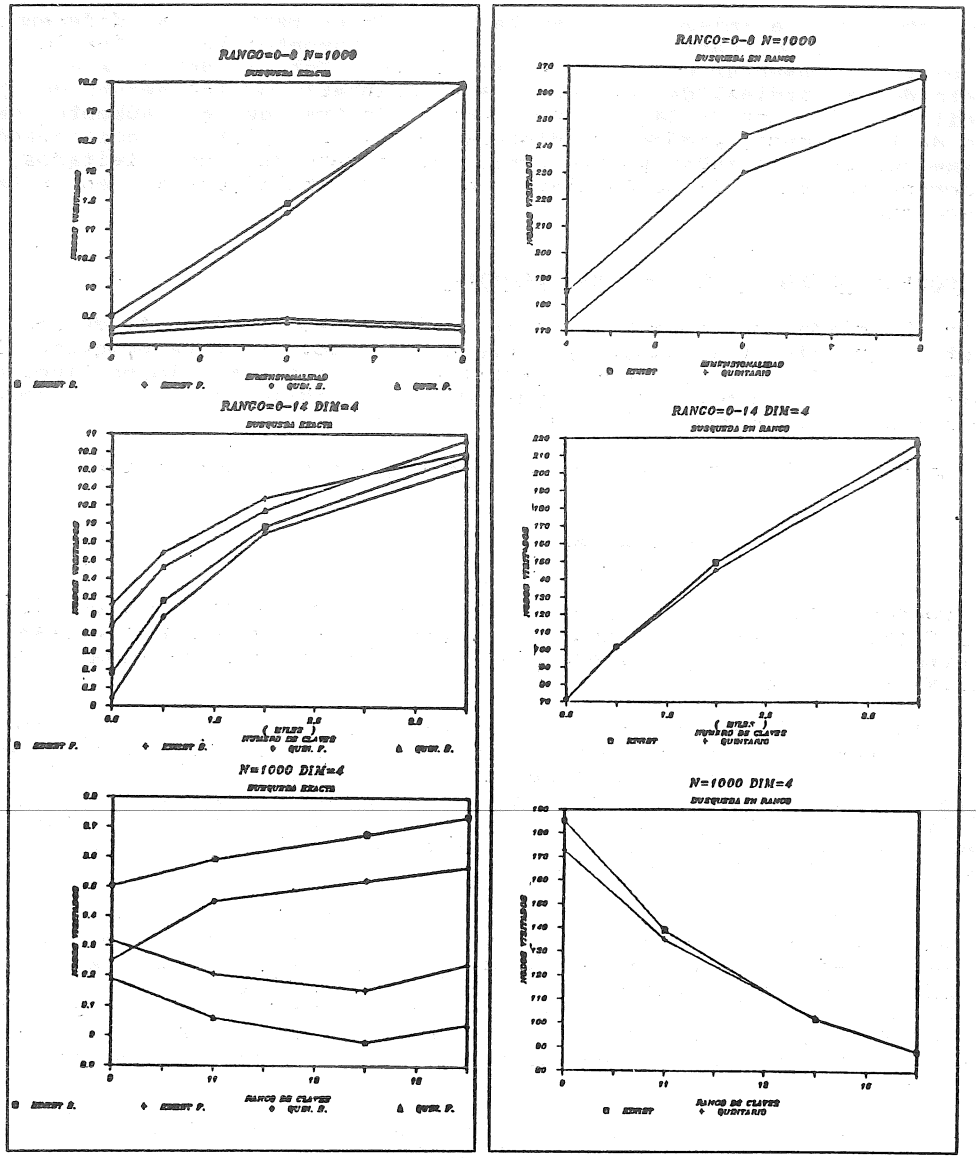
En el primer grupo se ha variado la cardinalidad en el segundo el rango y en el tercero la dimensionalidad.

En estas pruebas se evaluo el número de nodos visitados como parámetro significativo de comparación.

Para la obtención de los resultados se generó un árbol de cada uno de los tipos mostrados en el cuadro anterior y se realizaron sobre ellos 500 búsquedas aleatorias exactas con exito y 500 con fallo, tomando como resultado la media de las 500 pruebas. Para la búsqueda en rango se varió el hipercubo entre uno y cinco y se eligieron 500 claves aleatorias para cada hipercubo, tomando como resultado la media de las 2500 pruebas.

Se observa en la figura 3 que el comportamiento del quintario en la búsqueda exitosa y con fallo es mejor que el del KDHBT aunque la diferencia es muy pequeña.

Que el quintario sea mejor, es el resultado de que su estructura se genera de forma tal que si se eliminan las proyecciones (no usadas en este tipo de búsqueda) queda un árbol multidimensional completamente equilibrado.



Búsqueda exacta y en rango. Figura 3

En las pruebas de busqueda en rango los árboles quintarios son superiores a los KDHBT debido al uso que se hace en los primeros de las proyecciones, lo que agiliza parte de la búsqueda. De cualquier forma la diferencia entre uno y otro no es importante.

Se observa, asimismo, que en la búsqueda en rango, la diferencia entre el arbol Quintario y el kDHBT crece al aumentar la cardinalidad y disminuye al aumentar el rango de las claves. Esto es debido a que el aumento de la cardinalidad proporciona al Quintario un incremento de la posibilidad de uso de las proyecciones, mientras que el aumento del rango de las claves, produce un dispersión de los registros, ocasionando la disminución, en ambas estructuras, del número de nodos visitados, y un decremento, en el árbol Quintario, de las posibilidades de uso de las proyecciones.

6- BUSQUEDA DE LOS M-VECINOS MAS PROXIMOS

La búsqueda de los M vecinos más próximos es la búsqueda de los M registros que distan lo mínimo posible a uno dado. En esta búsqueda la distancia usada es un parámetro importante. Existen multiples tipos de distancia. En el campo numérico para datos multidimensionales se pueden generalizar de la siguiente forma:

La métrica Ln entre dos puntos de dimensión m se define como:

$$d = \eta\sqrt{\sum_{i=1}^{m} (Xi-Yi)\eta}$$

Cuando n toma el valor 2 se le llama distancia euclídea y es igual a $\sqrt[2]{\Sigma (Xi-Yi)^2}$ i=1..m. Cuando n es igual a ∞ se le denomina distancia infinita y es equivalente a la distancia definida como $d=\max|Xi-Yi|$ i=1..m.

Estas dos distancias serán las empleadas para realizar las pruebas. Al encontrar los M vecinos más próximos a un punto X, determinados con distancia euclidea, estos se encuentran en una hiperesfera de radio d y centro en X, siendo d la mayor distancia de los M vecinos a X. La infinita a su vez representa un hipercubo centrado en X de lado 2d (siendo d la mayor distancia de los M vecinos a X) con aristas en la union de dos hiperplanos coordenados.

Se recorre la estructura desde el primer nivel, localizando el nodo con valor de clave, en esa dimension, mas cercano al correspondiente de la interrogacion, acumulando la diferencia entre estos valores y descendiendo por su hijo central. El proceso se repite hasta alcanzar el ultimo nivel, en el que se inicia la vuelta atras con exploracion de alternativas por proximidad de los valores de clave en cada nivel, que no entrañen una distancia superior a la del vecino mas alejado hasta ese momento.

La localizacion ordenada por proximidad a un cierto valor, de los valores de clave de un árbol binario de búsqueda que no sobrepasen una diferencia determinada es un problema sencillo. Basta con dividir en dos la secuencia en orden simetrico del árbol e ir recorriendo ambas subsecuencias a partir del punto de corte de forma coherente con la ordenacion creciente de las diferencias al valor de clave dado. En el momento en que una de las subsecuencias se agote, o presente una

diferencia superior a la maxima especificada, se suspende su recorrido. El proceso termina al suspenderse el recorrido de ambas subsecuencias. Este método de búsqueda se basa en el uso de dos pilas para poder realizar la búsqueda de los predecesores y sucesores ordenadamente. El algoritmo realiza los siguientes pasos:

1 Se busca en el árbol el valor de clave dado. Esta búsqueda se realiza de la misma forma que en cualquier árbol binario, con la unica diferencia de que se van almacenando los valores de clave que se encuentren durante el descenso, en la pila de sucesores si su valor de clave es mayor que el buscado y en la de predecesores si es menor. Con esto se logra dividir el árbol en dos partes: la de los sucesores y la de los predecesores a la clave buscada, figura 4.

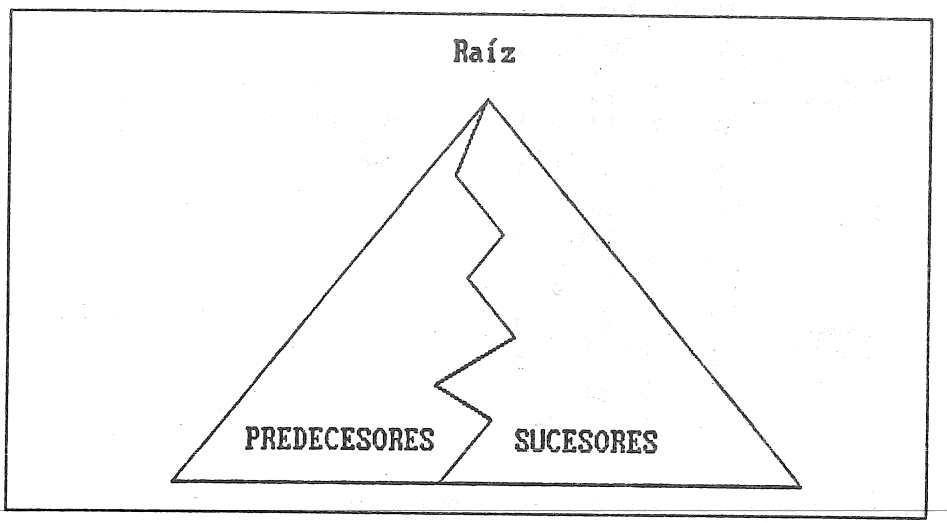


Figura 4

La búsqueda de la clave puede finalizar de dos formas: cuando se encuentre la clave buscada o cuando se encuentre un enlace nulo (el valor de clave buscada no existe).

2.a Si se encuentra el valor de clave buscada, se inserta en ambas pilas, se considera que se ha encontrado un vecino y se procesa. Despues se buscará el sucesor y el predecesor de la clave usando cada una de las pilas, si el sucesor o predecesor superase la diferencia permitida se eliminaria de la búsqueda la pila correspondiente.

2.b Si lo que ha detenido la búsqueda es un puntero nulo, entonces se tendrá en la cima de cada pila el sucesor y predecesor de el valor de clave, si alguno de las cimas supera la diferencia permitida se elimina su pila de la búsqueda.

3 Mientras exista predecesor y sucesor se elige el que tenga menor diferencia a la clave y se procesa. A continuación, se busca el sucesor

o predecesor según el caso. El proceso se repite hasta que se cumplan las condiciones de finalizacion especificadas anteriormente.

7- PRUEBAS REALIZADAS EN LA BUSQUEDA DE LOS M VECINOS MAS PROXIMOS

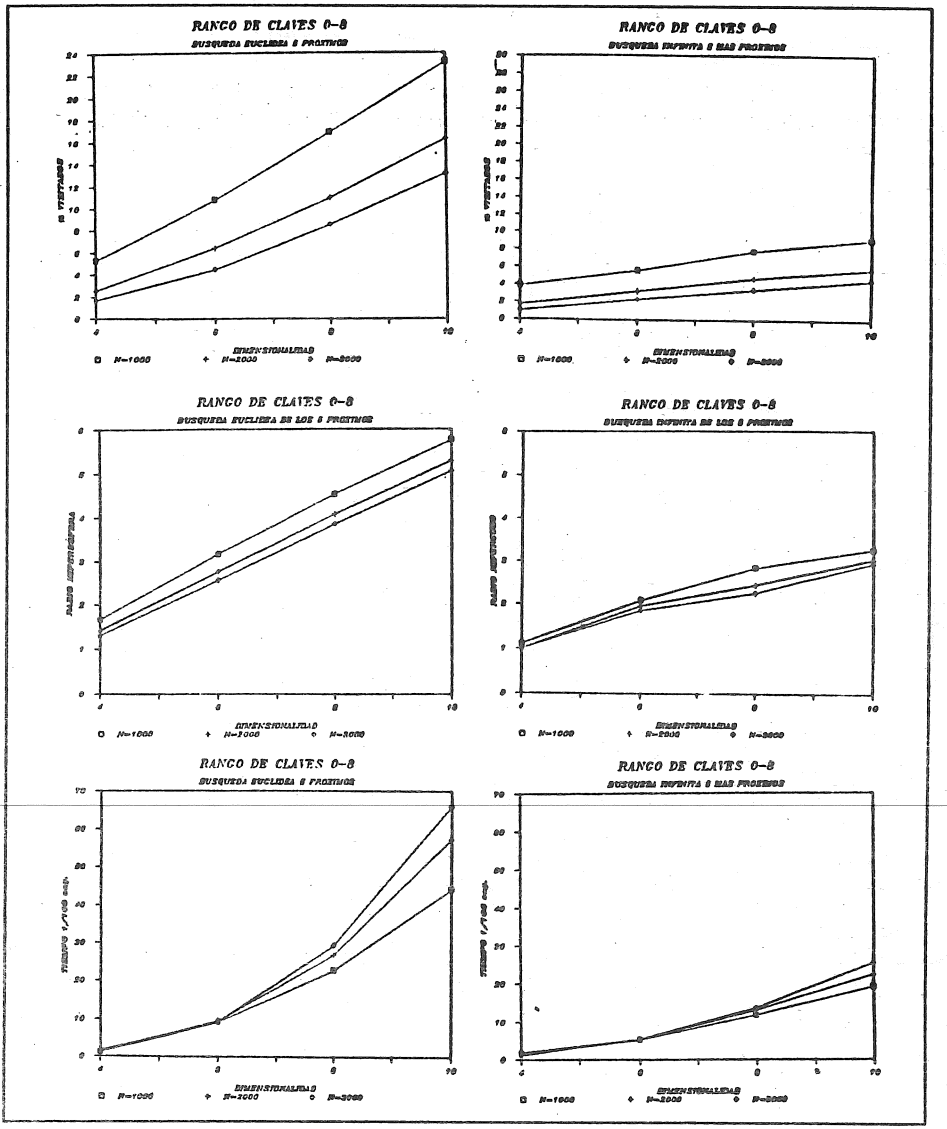
En las pruebas realizadas se variaron los parámetros significativos de la estructura, realizando las mismas pruebas para la distancia euclídea que para la infinita, formando el cuadro de pruebas mostrado en la figura 5.

	Dimensio.	Rango	Nº veci.	
Variando Dimensionalidad	4	0-8	5	Tomando en todas las pruebas valores para 1000, 2000 y 3000 registros.
	6	0-8	5	
	8	0-8	5	
	10	0-8	5	
Variando Rango	8	0-9	5	
	8	0-11	5	
	8	0-14	5	
	8	0-16	5	
Variando Nº vecinos	8	0-8	1	
	8	0-8	5	
	8	0-8	10	
	8	0-8	15	
	8	0-8	20	
	8	0-8	25	

Pruebas en la búsqueda de los M vecinos más próximos. Figura 5

Se realizaron 2000 búsquedas por prueba tomando como resultado el promedio. Se midieron el número de nodos visitados, la distancia máxima de los vecinos encontrados y el tiempo de búsqueda.

En los resultados de la figura 6, en los que se varía la dimensionaliadad de los registros, se aprecia que en la distancia euclídea el tanto por ciento de nodos visitados es mayor que en la búsqueda con distancia infinita. El aumentar el número de registros en el árbol el porcentaje de nodos visitados disminuye, aunque aumenta en valor absoluto. El número de nodos visitados esta directamente relacionado con los radios de una y otra distancia. El tiempo de búsqueda está relacionado con el número de nodos tratados.



Resultados variando la dimensionalidad. Figura 6

Al variar el rango de los valores de claves se observa en la figura 7, que como es lógico esperar, el aumento de distancia está directamente relacionado con el aumento del rango de las claves. El número de nodos visitados no parece sensible al aumento de dispersión.

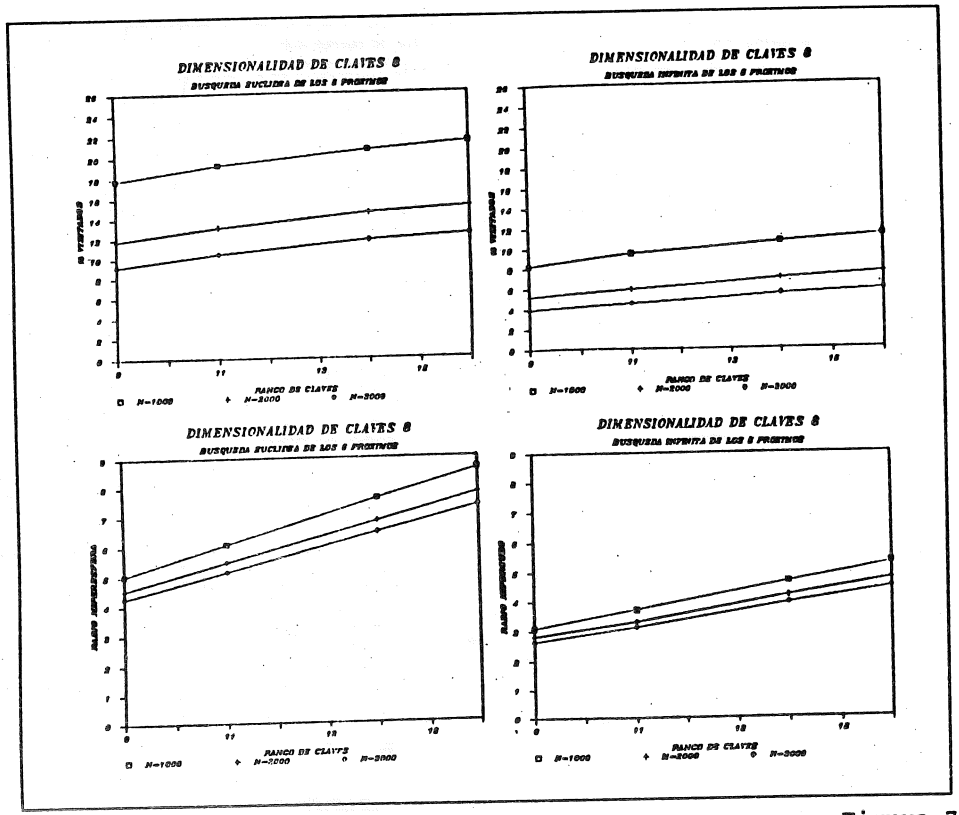
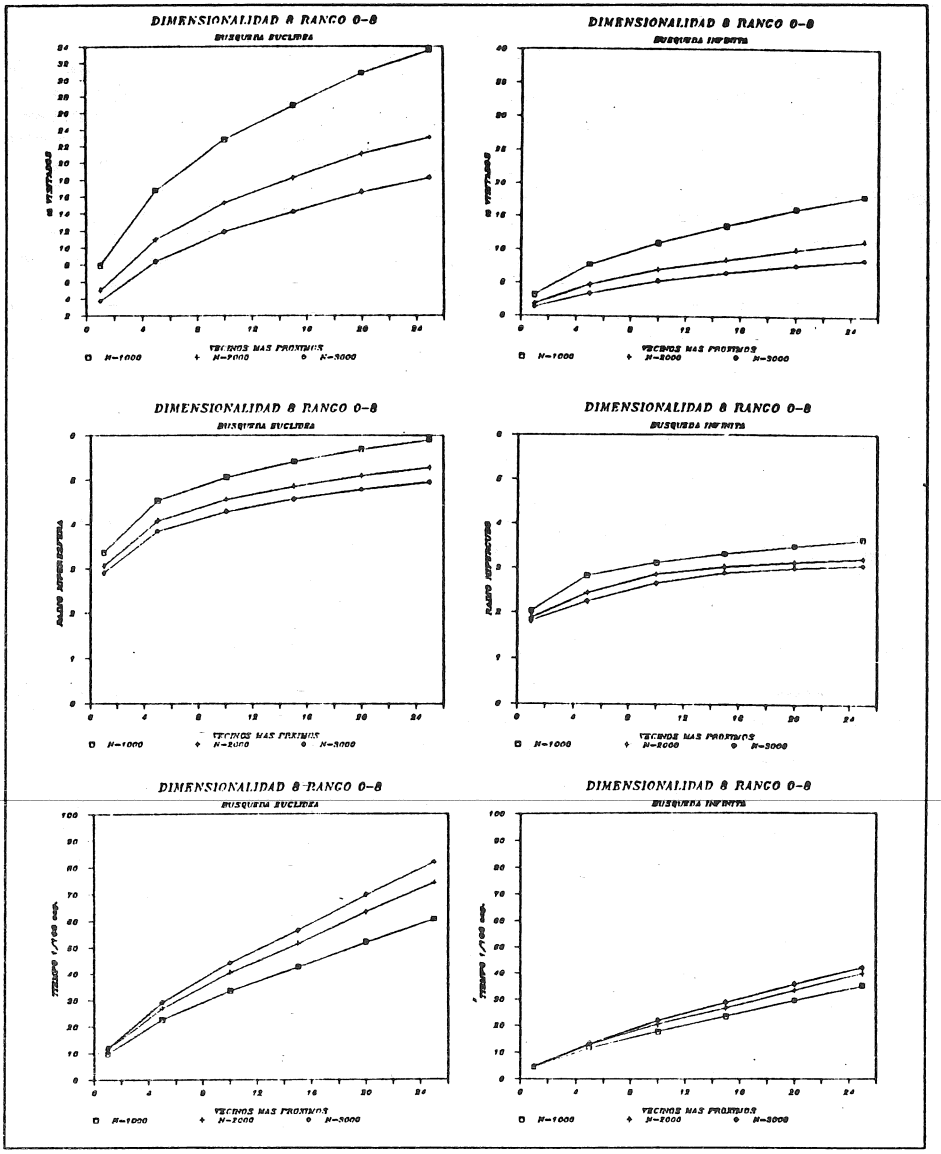


Figura 7

Resultados variando el Rango.

El tiempo de busqueda y el numero de nodos visitados, dependen del número de vecinos a buscar, figura 8, aunque con un crecimiento no tan rapido como en las pruebas en que se varia la dimensionalidad. Los radios que se obtienen con una y otra distancia vuelven a ser la explicación de las diferencias que se aprecian entre la búsqueda euclídea e infinita.



Resultados variando el número de vecinos. Figura 8

8- CONCLUSIONES

A lo largo de este trabajo se han visto distintos aspectos de la estructura KDHBT. El primer punto importante que surge es que sin ser una estructura complicada ofrece dos características que muchas de las otras estructuras en su ambito no ofrecen, como son: dinamismo sin perdida de eficiencia ni coste excesivo de memoria. De las pruebas de comportamiento en fase creciente, estacionaria y decreciente se deduce su capacidad de reacción ante las acciones de modificación de los datos, teniendo como respuesta pocas acciones de reestructuración y mantenimiento de la altura media. En las pruebas de búsqueda exacta y en rango al ser comparadas con el árbol quintario se comprueba que la perdida de eficiencia debida al dinamismo no es significativa, sobre todo en la búsqueda exacta. En la búsqueda de los vecinos más próximos el comportamiento de la estructura se ve influenciado por la distancia con que se determinan. Con distancia infinita en la realizacion de los experimentos parece conducirse mejor que con la euclidea.

REFERENCIAS

[AV62] G.M. ADEL'SON-VELSKIJ AND Y. M. LANDIS: An algorithm for the organization of information. Sov. Math. Doklady. Vol 3, pp. 1259-1263, 1962

[LW80] LEE, D. T. AND WONG, C. K.: Quintary trees: A file structure for multidimensional database systems. Vol 5, N° 3, pp 339-353. 1980

[SM87] SANTANA, O.; MAYOR, O.; DIAZ, M.; REINA, S.: Arboles Quintarios: Estudio esperimental para interrogaciones exactas, parciales, en rango y en rango parcial. XIII Conferencia Latinoamericana de Informatica. Vol II, pp 1148-1168. 1987

[VV84] VIJAY K. VAISHNAVI: Multidimensional height-balanced trees. IEEE Transactions on Computers. Vol C-33, N° 4, pp 334-343. 1984.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

O MÓDULO DE POSICIONAMENTO RELATIVO DO PROJETO TRANCA

MARCELO LUBASZEWSKI *

Pós-Graduação em Ciência da Computação (PGCC) - UFRGS
Av. Osvaldo Aranha, 99 - CP 1501
90.001 - Porto Alegre - RS - BRASIL
Telefone: (0055) 512-212161
Telex (051) 2680

RESUMO

Este artigo apresenta o desenvolvimento do módulo de posicionamento relativo do Projeto TRANCA ("TRANsparent Cell Approach").

TRANCA trata-se de um sistema de síntese automática de layout, baseado em uma metodologia de projeto que emprega técnicas de roteamento diferentes daquelas em que "standard cell" tradicional se apóia. As interconexões entre células são realizadas sobre as áreas ativas do circuito, com vistas à eliminação de canais de roteamento e à obtenção de rotas menos resistivas.

A partir da descrição estrutural do circuito, de informações topológicas da biblioteca de células e da área reservada em planta baixa, o módulo de posicionamento relativo gera partições de células com aproximadamente a mesma área e com um número mínimo de redes comuns. Cada partição constituirá uma fileira de células na estrutura física. Maximizando o número de conexões realizadas sobre as células e minimizando o número de células de interconexão responsáveis pela comunicação entre fileiras, obtém-se um ganho considerável em área de implementação e velocidade do circuito.

O procedimento para a partição de blocos "standard cell" com transparência fundamenta-se em partição de grafos aplicada ao posicionamento de circuitos. As principais características do módulo sob investigação residem em: tratamento flexível do critério de balanceamento, realização de otimizações locais (baseadas em transferências individuais de células e alterabilidade de pesos de redes) e tempo de processamento reduzido (pela utilização de estruturas de dados redundantes e ordenações diversas).

* Engenheiro Eletrônico (UFRGS -1986); Mestrando em Ciência da Computação (PGCC - UFRGS).

1. INTRODUÇÃO

A integração de circuitos VLSI tornou-se viável graças ao surgimento de *ferramentas computacionais para auxílio ao projeto*: à medida que aumenta a complexidade do sistema digital que se deseja implementar, tanto mais se faz necessária a *redução do tempo de projeto* e a garantia de *correção do circuito*.

Atendendo à demanda do mercado, sistemas de *síntese automática* passaram a ser desenvolvidos e têm evoluído no sentido de abranger transformações a partir de níveis de abstração cada vez maiores. O objetivo deste processo é chegar a ferramentas que possibilitem a realização física do projeto, partindo-se da especificação comportamental do circuito apresentada sob forma algorítmica.

1.1. O Sistema TRANCA

Dentro do contexto de projeto automático, o sistema TRANCA pretende contribuir com uma ferramenta de *síntese de layout de blocos "standard cell"* a partir de uma descrição a nível lógico do circuito (estrutural ou comportamental).

A figura 1 mostra os módulos que constituem o sistema TRANCA. Os módulos envoltos por retângulos duplos são aqueles já desenvolvidos [LUB 88a] [LUB 88b] ou em desenvolvimento. As setas em destaque simbolizam os pontos de entrada no sistema.

O *estilo de layout* gerado por este sistema difere de "standard cell" tradicional pelo fato do *roteamento* ser realizado *sobre a área ativa do circuito*, dispensando o uso de canais dedicados a interconexões. Esta metodologia introduz *compactação de área* e *minimização de características elétricas indesejáveis* [REI 87] [REI 88].

Não é intenção deste artigo fazer uma análise global dos módulos que constituem o sistema TRANCA. Pretende-se, contudo, descrever as estratégias envolvidas no módulo de posicionamento relativo.

1.2. O Módulo de Posicionamento Relativo

De uma maneira geral, o posicionamento das células que constituem um bloco é dito *relativo*, quando a um conjunto de células, cujas relações de adjacência (interconexões) são conhecidas, se atribui parte de uma área sem a definição de localização de cada célula individualmente [LIE 85].

Os geradores de posicionamento existentes são baseados em diferentes tipos de algoritmos. Os mais apropriados para o problema de *alocação relativa* fundamentam-se em *partição de grafos*.

No sistema TRANCA, o objetivo é dispor de um gerador de posicionamento relativo que forneça uma partição de um bloco em conjuntos de células com aproximadamente a mesma área e pouco relacionados entre si. Cada um destes conjuntos constituirá uma fileira (banda) na estrutura física.

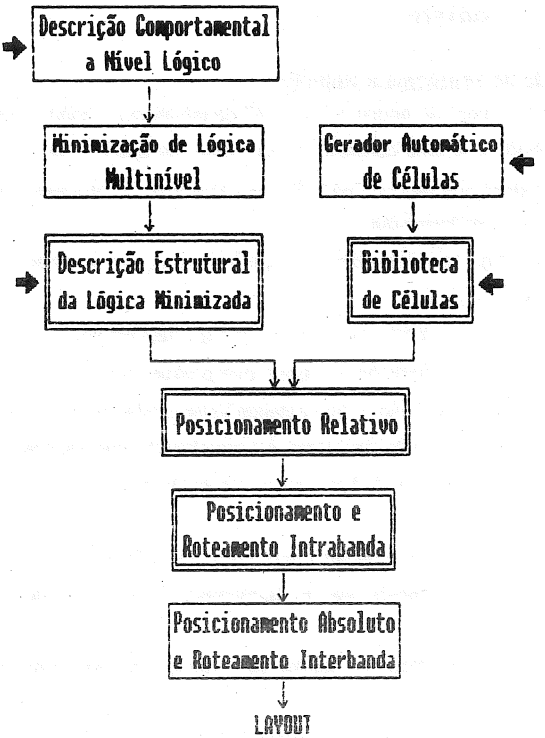


Figura 1 - O Sistema TRANCA.

Basicamente, este trabalho está dividido como se explica a seguir.

Na seção 2, o problema de partição de grafos é analisado em detalhe.

A seção 3 procura estabelecer o vínculo entre partição e posicionamento de circuitos. Analisa-se como realmente a heurística Kernighan-Lin (seção 2) tem sido empregada e como as limitações de dimensões são tratadas.

A seção 4 detém-se na solução para o problema de partição de blocos de "standard cells" com transparência [REI 87] aplicada ao seu posicionamento.

Na seção 5, descreve-se os procedimentos envolvidos na implementação do módulo de posicionamento relativo, as estruturas de dados adotadas e as características relevantes de alguns algoritmos já desenvolvidos.

Comentários relacionados à inclusão de procedimentos para a otimização global das partições aparecem na seção 6.

2. PARTIÇÃO DE GRAFOS

2.1. Definição do Problema e Soluções

Dado um grafo G, com N nodos n_i (i=1...N) de tamanho t_i, ligados por A arcos a_{ij} (j=1...N) com pesos p_{ij}, e supondo que uma condição de contorno, do tipo tamanho (T < somatório dos t_i) ou cardinalidade (C< N) desejada de cada partição, seja fornecida, o processo consiste na divisão de G em subgrafos g_k (k <= N) de forma que:

(a) para qualquer g_k, somatório dos t_i (n_i pertence a g_k) <= T, ou para qualquer g_k, somatório dos n_i (n_i pertence a g_k) <= C; e,

(b) somatório dos p_{ij} (a_{ij} pertence a mais de um g_k) seja mínimo.

O que se deseja é gerar partições do grafo que possuam um número mínimo de arcos comuns com pesos elevados (custo de partição) e que estejam balanceadas segundo algum critério.

Uma solução exata para este problema é normalmente impraticável, uma vez que ele é de natureza combinatória. Problemas deste tipo - sujeitos à explosão combinatória, são conhecidos como *NP-completos* [HYA 73].

Já que a obtenção de uma solução ótima neste caso implicaria em elevadíssimo tempo de processamento, ***heurísticas*** parecem ser bons recursos para a produção de bons resultados rapidamente.

Várias tentativas têm sido feitas no sentido de formular procedimentos para a determinação de partições.

A ***geração de soluções aleatórias***, mantendo a melhor de todas aquelas analisadas até um determinado momento e encerrando o processo ao atingir um tempo ou número de tentativas pré-especificado, apresenta uma probabilidade de sucesso, em cada tentativa, extremamente baixa [KER 70].

A ***procura de aglomerados de células fortemente relacionadas*** geralmente não inclui provisões para a satisfação das limitações de tamanho das partições e não sistematiza a alocação de nodos que não pertencem obviamente a nenhum aglomerado em particular.

A heurística *Kernighan-Lin*, descrita a seguir, além de prover procedimentos para os casos não tratados nas heurísticas referidas acima, apresenta uma probabilidade estatística de sucesso satisfatória. Kernighan & Lin são referenciados em toda a literatura que versa sobre partição aplicada.

2.2. A Heurística Kernighan-Lin

O problema de *partição em múltiplos subgrafos* não conta até o presente momento com nenhum método satisfatório para seu tratamento. O que se tem feito é a aplicação sucessiva de ***bipartições***, especificando-se diferentes critérios de balanceamento a cada passo (quando necessário), no sentido de gerar tantos subgrafos quantos se queira e que representem boas partições quando comparados dois a dois.

Neste sentido, a heurística Kernighan-Lin, definida para bipartições, passa a ter um caráter mais abrangente.

A idéia da heurística é, partindo-se de dois subgrafos quaisquer, determinar um conjunto de nodos de cada subgrafo, tal que, transferindo-se cada um destes conjuntos para o outro subgrafo, se obtenha uma redução na inter-relação entre as partições do grafo (figura 2.1). Este procedimento é repetido até que não seja mais possível a minimização do custo.

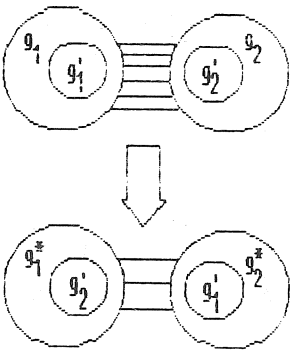


Figura 2.1-A idéia da heurística Kernighan-Lin.

Antes da apresentação detalhada da heurística, a introdução de alguns conceitos é primordial.

Custo Externo de um nodo (E) é entendido como sendo o somatório dos pesos atribuídos aos arcos do nodo que o relacionam com o subgrafo ao qual ele não pertence.

Custo Interno (I), por sua vez, é o somatório dos pesos atribuídos aos arcos que o ligam a nodos pertencentes ao mesmo subgrafo.

Suponha que g_1 e g_2 sejam uma partição do grafo e que n_1 pertencente a g_1 e n_2 pertencente a g_2 sejam dois nodos. Ganho (G) de intercâmbio de n_1 e n_2 é definido como a diferença entre o somatório dos pesos dos arcos comuns a g_1 e g_2 antes e depois da troca dos nodos. Prova-se [KER 70] que o ganho é dado por

$$G = E_{n1} - I_{n1} + E_{n2} - I_{n2} - 2p_{12},$$

onde p_{12} é o peso do arco que liga n_1 a n_2.

Exemplo:

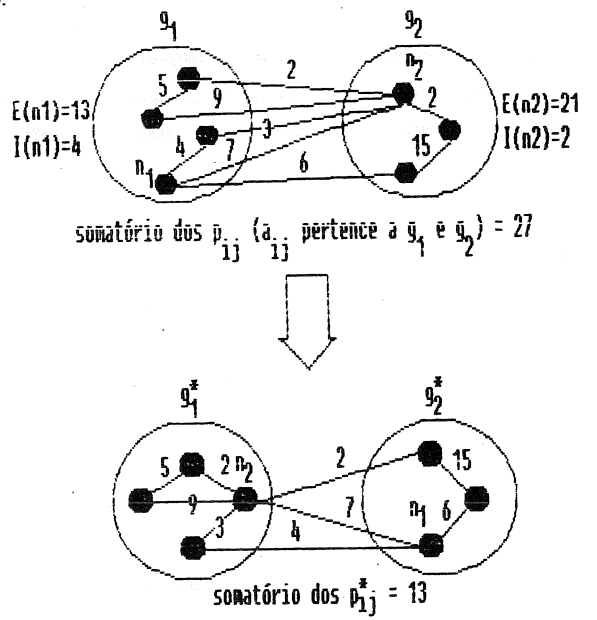


Figura 2.2 - Exemplo de troca de nodos.

Podemos definir ainda o que se entende por custo de um nodo (D):

$$D_{ni} = E_{ni} - I_{ni} \quad (i = 1...N).$$

A seguir se apresenta o procedimento global na forma algorítmica:

1. Calcular o custo de todos os nodos de g_1 e g_2.
2. Escolher n_i pertencente a g_1 e n_x pertencente a g_2 tal que

$$G = D_{ni} + D_{nx} - 2p_{ix}$$

seja máximo.

3. Esquecer n_i e n_x temporariamente e chamá-los de n_i' e n_x' respectivamente.
4. Recalcular os custos dos nodos pertencentes a $g_1 - \{n_i\}$ e $g_2 - \{n_x\}$ por

$$D_{ny}' = D_{ny} + 2p_{yi} - 2p_{yx} \qquad n_y \text{ pertence a } g_1 - \{n_i\}$$

$$D_{nz}' = D_{nz} + 2p_{zx} - 2p_{zi} \qquad n_z \text{ pertence a } g_2 - \{n_x\}$$

onde as parcelas somadas correspondem aos pesos dos arcos que passam a integrar também o outro subgrafo, e as subtraídas, aos pesos dos arcos que passam para o interior do subgrafo em questão.

5. Repetir os passos de 2 a 5, não levando em consideração os nodos anteriormente selecionados em 2, até que todos os nodos tenham sido analisados.
6. Escolher dentre os pares (n_i', n_x') a seqüência (a partir do primeiro, inclusive) daqueles que, tendo seus ganhos somados, produzem um ganho total máximo.
7. Se somatório dos $G > 0$, então realizar a troca dos nodos selecionados em 6 entre g_1 e g_2 e voltar a 1.
8. Fim do procedimento.

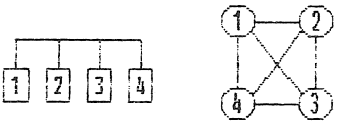
Uma medida razoável de efetividade de um procedimento é dada pela probabilidade de que ele chegue a uma solução ótima em uma única tentativa e pelo tempo de processamento envolvido. [KER 70] discute em detalhes ambos os aspectos com relação à referida heurística, chegando a uma probabilidade de $2^{-N/30}$ e tempo de máquina da ordem de $N^{2,4}$.

3. PARTIÇÃO DE CIRCUITOS

3.1. O Tratamento de Redes Múltipias

O mapeamento de um circuito integrado em uma estrutura do tipo grafo é imediato: a *nodos* correspondem *células*, a *arcos* correspondem *grupos de conexões*, a *peso de arco* entre dois nodos corresponde o *número de conexões* entre duas células, e a *subgrafos* correspondem *subcircuitos*. Devido a esta correspondência direta, toda a teoria apresentada até então é válida para o problema de partição de circuitos aplicado ao posicionamento destes.

No entanto, a modelagem de circuitos por grafos deixa a desejar no tocante à representação de *conexões multiponto (redes múltiplas)*. Arcos relacionam nodo a nodo de um grafo, de forma que conexões múltiplas seriam representadas por *subgrafos completos* (subgrafos onde cada nodo se liga a todos os outros) nesta estrutura (figura 3.1).



(a)rede multiponto (b)subgrafo completo

Figura 3.1 - Representação de conexões multiponto em grafos.

Ao realizar uma troca de nodos entre partições, se pelo menos um deles faz parte de uma rede múltipla, o ganho calculado nem sempre retrata o ganho real (figura 3.2). Isto decorre do fato de uma conexão do tipo multiponto, que é implementada com um único fio, ser representada por vários arcos de pesos comparáveis àquele que corresponde à ligação física de somente duas células.

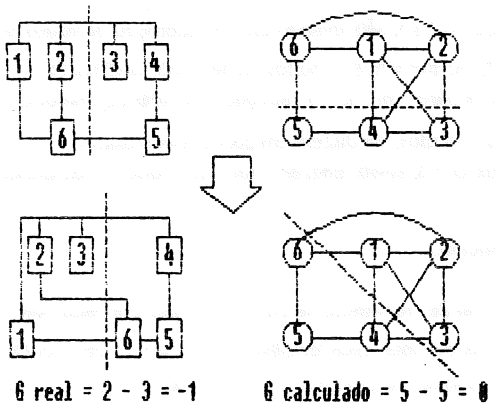


Figura 3.2 - Intercâmbio das células (nodos) 3 e 6.

A modelagem por grafos favorece, outrossim, a formação de aglomerados de nodos que compartilham a mesma rede múltipla, o que não é sinônimo de uma boa partição (figura 3.3) [SCH 72].

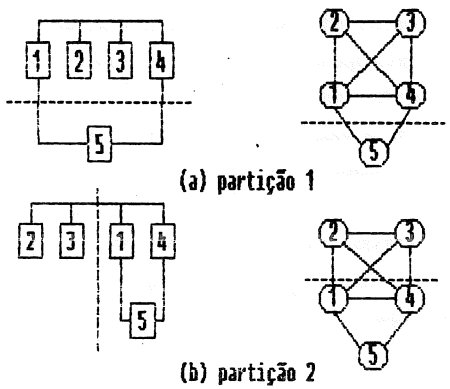


Figura 3.3 - Erro na representação na forma de grafo.

Contudo, quando se quer minimizar o custo de objetos ligados, a melhor solução é adotar uma outra representação gráfica e, principalmente, estabelecer outros critérios para o cálculo de ganho de partição. Neste caso, heurísticas que envolvem *cálculo de ganho por diferença de custos*, como aquela descrita na seção anterior, podem ser facilmente modificadas para suportar o tratamento de redes multiponto.

Para tanto, [SCH 72] propõe uma estratégia de ataque à nova situação.

O ganho passa a ser obtido por:

$$G = D_{ni} + D_{nx} - \text{fator de correção } (n_i, n_x) - \text{fator de correção } (n_x, n_i).$$

onde:

D_{ni} e D_{nx}

Passam a ser calculados segundo as seguintes regras:

Para cada *rede r* à qual a célula pertence:

1. Se a célula é a *única representante de r na partição,* então a parcela do seu custo relacionada a *r* é *positiva* e de *módulo igual ao peso da rede.*

2. Se a célula *não é a única da rede r na partição,* dois casos são possíveis:

(a) Se *não existem células de r na partição complementar,* então a parcela do seu custo relacionada à referida rede é *negativa* e de *módulo igual ao peso de r.*

(b) Se *existem células de r na outra partição,* então a parcela em questão é *nula.*

fator de correção (n_i , n_x) *[fator de correção* (n_x, n_i) *]*

Prevê a possibilidade de n_i e n_x estarem conectados. Se n_i (n_x) é o único elemento de uma determinada rede *r* em g_1 (g_2), e se n_x (n_i) também está em *r*, D_{ni} (D_{nx}) contém uma parcela igual ao *peso da rede* associada ao movimento de n_i (n_x) para g_2 (g_1), que claramente não será computada se n_x (n_i) for simultaneamente movido para g_1 (g_2). Este fator simplesmente subtrai *o peso de r* do ganho nestas situações.

O *recálculo* sugerido pelo passo 4 da heurística Kernighan-Lin (seção 2.3) passa a ser realizado por:

$$D_{ny}' = D_{ny} + [\text{fator de correção } (n_y, n_i) + \text{fator de correção } (n_i, n_y)]_{\text{depois da troca}} - [\text{fator de correção } (n_y, n_x) + \text{fator de correção } (n_x, n_y)]_{\text{antes da troca}}$$

onde n_y *pertence a* $g_1 - \{n_i\}$:

$$D_{nz}' = D_{nz} + [\text{fator de correção } (n_z, n_x) + \text{fator de correção } (n_x, n_z)]_{\text{depois da troca}} - [\text{fator de correção } (n_z, n_i) + \text{fator de correção } (n_i, n_z)]_{\text{antes da troca}}$$

onde n_z *pertence a* $g_2 - \{n_x\}$.

Evidentemente, a implementação de tais critérios é onerosa em termos de tempo de processamento. Todavia, a garantia de melhores resultados [SCH 72] é compensadora.

3.2. Critérios de Balanceamento

Até agora não foi fornecido nenhum detalhe com respeito aos critérios de balanceamento de uma partição. Como já citado, estes podem ser basicamente de dois tipos: *cardinalidade* ou *tamanho das partições*.

Em se tratando de cardinalidade, desde que as trocas de células sejam realizadas aos pares, o equilíbrio inerente às partições iniciais não é jamais perturbado.

Se o critério for tamanho dos subcircuitos, um cuidado maior deve ser dispensado. Neste caso, a cada nova iteração, antes de efetuar o intercâmbio, a condição de equilíbrio deve ser avaliada para tal operação. O que normalmente se faz é estabelecer previamente uma *razão* do tipo

$$lg_1 l \;/\; (lg_1 l + lg_2 l) \;\tilde{=}\; q, \text{ onde}$$

$lg_k l$ é o somatório das larguras das células que constituem o subcircuito k.

Uma vez que a razão na forma como é proposta não é determinística (~=), deve-se especificar o intervalo de tolerância admissível para o desvio da igualdade. [FID 82] sugere

$$q.W - t_{max} <= |g_1| <= q.W + t_{max},$$

onde $W = |g_1| + |g_2|$ e $t_{max} = max\{t_i\}$, e adverte que este critério funciona bem em casos onde a variação nos tamanhos das células não é muito grande. Desta forma, é permitido a *todas* as células moverem-se à partição complementar, desde que o balanço seja respeitado.

Evidentemente, o fato de respeitar sempre tal critério é mais um fator limitante na busca de uma solução ótima.

3.3. Formas de Implementação

Providas soluções para as deficiências e indefinições do procedimento de partição de grafos aplicado a circuitos, passemos à análise geral das implementações que se valem dos princípios até agora estabelecidos.

Quando a questão é posicionamento relativo, o problema se limita a quantas partições realizar e como fazê-lo.

Uma solução é partir de tantos subcircuitos quantos desejados e submetê-los *dois a dois* ao procedimento descrito, até que melhorias não sejam mais possíveis em nenhum par. Novamente se recai no problema de explosão do número de tentativas. Se o número de partições imposto é elevado, o número de vezes de realização do processo (no mínimo $C_{P,2}$, onde P é o número de subcircuitos) pode se tornar muito grande.

Outra alternativa - a mais utilizada, seria a realização de *bipartições sucessivas*. Neste caso, o número de execuções do processo é bastante mais reduzido ($P - 1$). Ainda, as formas de implementação podem ser distintas:

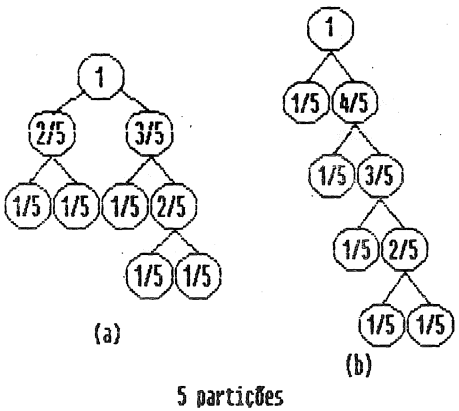


Figura 3.4 - Formas de implementação de bipartições sucessivas.

Computacionalmente falando, a alternativa (b) é bastante mais fácil de implementar. Com relação à efetividade de uma e outra versão, nem a literatura analisada apresenta estudos, nem tampouco é intuitivo discursar a respeito.

No posicionamento absoluto, o *aspecto geométrico* desponta como dominante. Aqui são definidas sucessivamente *linhas imaginárias* que cortam uma estrutura física, e os esforços são direcionados no sentido de minimizar o número de conexões perpendiculares a tais linhas. O controle do critério de balanceamento torna-se crítico. Este procedimento é conhecido na literatura como posicionamento *"min-cut"*. [COR 79] descreve em detalhes esta classe e tece comentários a respeito de resultados obtidos utilizando-se diferentes estratégias.

Da análise dos pontos apresentados nesta seção e daqueles anteriormente estudados, fica evidente que, em se tratando de *partição*, *nada é determinismo, tudo é heurística*.

4. POSICIONAMENTO RELATIVO DE "STANDARD CELLS" COM TRANSPARÊNCIA

4.1. A Metodologia de Projeto

A metodologia de projeto baseia-se em quatro pontos fundamentais: *estrutura de banda*, *transparência horizontal e vertical*, *gerenciamento de trilhas* e *maleabilidade*.

Nesta seção, serão discutidos apenas os aspectos que influenciam o tratamento de posicionamento via partição, ou seja, transparência aplicada ao roteamento. Detalhes sobre a metodologia podem ser encontrados em [REI 87] e [REI 88].

As células são projetadas de forma a disporem de trilhas livres, paralelas às linhas de alimentação, por onde as conexões externas são implementadas em *metal*. A transparência horizontal está relacionada ao número de trilhas que podem atravessar a célula.

O roteamento interno a uma banda é, portanto, realizado sobre as células, dispensando o uso de canais dedicados a conexões (figura 4.1).

O roteamento entre bandas usa células de interconexão [REI 87] para a passagem de sinais. Estas células - implementadas em *polisilício*, são posicionadas (uma em cada banda) entre células funcionais e resgatam os sinais de interesse, passando-os às bandas adjacentes. As conexões são realizadas verticalmente às linhas de alimentação neste caso (figura 4.1).

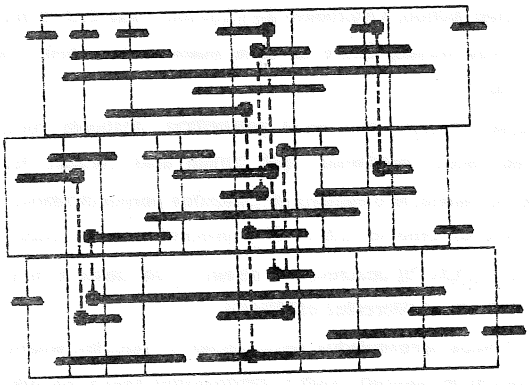


Figura 4.1 - Roteamento sobre as bandas e utilizando células de interconexão.

O objetivo de se utilizar um processo de partição para determinar que células devem compor que bandas reside em:

(a) maximizar o número de conexões realizadas em metal (sobre a banda), visando a obtenção de rotas menos resistivas; e,

(b) minimizar o número de células de interconexão, pela redução do número de redes comuns às bandas, visando a otimização de área de implementação.

4.2. Estratégias de Partição

A solução adotada é a de realização de *bipartições sucessivas*. Procura-se compensar as limitações inerentes à ação de divisão-e-conquista associada a este procedimento, pelo acréscimo de provisões para a otimização das partições aos pares. Para tanto, os pesos das redes múltiplas do tipo interface são reavaliados antes que inicie a próxima partição.

Os subcircuitos são gerados *em fatias*, ou seja, o critério de balanceamento utilizado faz com que o resultado de cada partição seja uma *banda* e um *bloco residual*. Este bloco pode ser ou a última banda gerada pelo processo, ou um subcircuito a particionar. Esta estratégia introduz maior simplicidade de processamento, uma vez que inexistem blocos pendentes na fila de espera do processo. Além do mais, somente nestas condições se torna viável a otimização das partições aos pares citada anteriormente e descrita mais adiante.

Com relação a uma bipartição, a otimização é realizada pela permissão de *transferências individuais de células* quando as trocas aos pares não forem mais possíveis.

4.3. Descrição do Procedimento

As informações necessárias para disparar um processo de partição provêm da *descrição estrutural* do circuito [WAG 87b], da *biblioteca de células* [LUB 88b] e do *local de destino do bloco na planta baixa*.

A extração da conectividade é feita a partir da descrição estrutural do circuito. Para cada *célula* são obtidos *nome, redes conexas* e *células vizinhas*; para cada *rede*, devem ser conhecidos o seu *peso* e as *células* que estão a ela conectadas. Estes dados são do tipo *estático*, isto é, permanecem inalterados durante todo o processo.

No decorrer do procedimento, informações *dinâmicas* são geradas e agregadas às referidas estruturas. Alguns exemplos são: *partição* à qual a célula pertence, *custo* de célula, apontadores de *ordenação de custos* e outras.

Da biblioteca deseja-se a informação estática de *largura* de célula, enquanto que, da planta baixa, se quer saber o *formato* aproximado do bloco a posicionar. Este último é fornecido sob a forma de *relação de aspecto (RA)*, ou seja, razão altura/largura do bloco.

De posse de todos os dados, o primeiro passo consiste no cálculo do número de bandas a gerar (*nb = número de bipartições + 1*). Isto é feito como se mostra a seguir:

$$RA = altura / largura = [\, nb \,.\, k_1 \,] / [\, (1+k_2) \,.\, W / nb \,] \quad \text{(I)}, \text{ onde}$$

k_1 é uma constante - dependente de tecnologia, função da *altura padrão* das células e da *distância mínima* entre bandas; e,

k_2 é uma constante - função de estudos estatísticos, que fornece o acréscimo percentual de área pela *inclusão de células de interconexão* no circuito.

De (I) vem que:

$$nb = [\, (1+k_2) \,.\, W \,.\, RA / k_1 \,]^{1/2} \quad \text{(II)}.$$

Supondo que o valor de k_2 seja super-estimado para a maioria dos casos, *nb* - dado por (II), é truncado para resultados fracionários.

A rigor, k_1 pode ser inferior ao valor previsto quando da partição resultam bandas idênticas (*regularidade*), pois, neste caso, as linhas de alimentação de mesmo potencial de bandas adjacentes poderiam ser sobrepostas [LUB 88b]. Na falta de previsões, prefere-se assumir que isto não ocorre.

O segundo passo é a determinação da largura máxima de célula (l_{max}) para o estabelecimento do critério de balanceamento da bipartição. Este passo é repetido a cada nova bipartição, visando a redução da probabilidade da última banda (gerada como bloco residual) não respeitar o critério (ser maior ou menor que os limites estabelecidos).

Neste ponto, a inequação de equilíbrio fica perfeitamente definida, já que $q = 1 / nb$ e l_{max} é conhecido.

O estabelecimento das *partições iniciais* precede o processo de bipartição propriamente dito. Este é realizado por *pesquisa seqüencial*, procurando aproveitar a tendência do projetista de descrever o circuito agrupando as células de intensa conectividade entre si. Esta atitude deve reduzir o número de trocas de células nas etapas seguintes.

Chega-se, então, ao centro do problema: a execução da heurística Kernighan-Lin. Neste passo, são acrescidas algumas *provisões* e são feitas *modificações*:

(a) a *ordenação de custos* é feita paralelamente ao cálculo destes [FiD 82], o que resulta em economia de tempo na pesquisa do par que maximiza o ganho;

(b) o *recálculo dos custos*, após a escolha de cada par, é realizado somente para as células vizinhas daquelas que constituem o par e não envolve procedimentos que determinem fatores de correção;

(c) a *verificação do equilíbrio (balanço)* entre partições é realizada somente quando a proposta do conjunto de pares a serem transferidos está disponível, possibilitando que as trocas se compensem entre si;

(d) a *transferência individual de células* [FID 82], desde que trocas aos pares não sejam mais proveitosas e que o balanço seja mantido, otimiza a partição.

Caso a compensação citada em (c) não ocorra, os *últimos* pares selecionados são retirados sucessivamente do conjunto, até que o critério passe a ser respeitado.

O objetivo básico de (d) é tratar as células de custo positivo, que não foram movidas pelo fato de constituirem um par que não respeitava o equilíbrio, no sentido de transferi-las, uma a uma, realizando o teste do critério de balanceamento para *cada* caso. Em última análise, o que se tenta fazer é um rearranjo dos pares de forma a manter o balanço.

Entre duas bipartições, ocorre a *modificação dos pesos das redes multiponto da interface* (que inicialmente podem possuir pesos superiores a 1, caso sejam consideradas *críticas* por imposição do projeto). A estas redes são atribuídos pesos sucessivamente maiores, à medida que elas se propagam à banda e ao bloco residual. O valor que se acresce ao peso é função do aumento do número de conexões entre as bandas anteriormente geradas e os blocos resultantes da partição, caso a rede não se limite a um subcircuito deste (figura 4.2).

A exceção à regra é a atribuição de peso nulo às redes (simples ou múltiplas) da interface que possuem uma única célula no bloco residual. Neste caso, a existência de tal rede é irrelevante para a próxima bipartição.

Esta atitude aumenta significativamente a probabilidade destas redes se restringirem ou à banda, ou ao bloco residual, o que resulta em minimização do número de conexões multibanda.

Por fim, a *dissociação da banda gerada* é uma ação importante, no sentido dos passos seguintes manipularem estruturas de dados cada vez menores. A informação de peso das redes múltiplas da interface é a única que precise prosseguir.

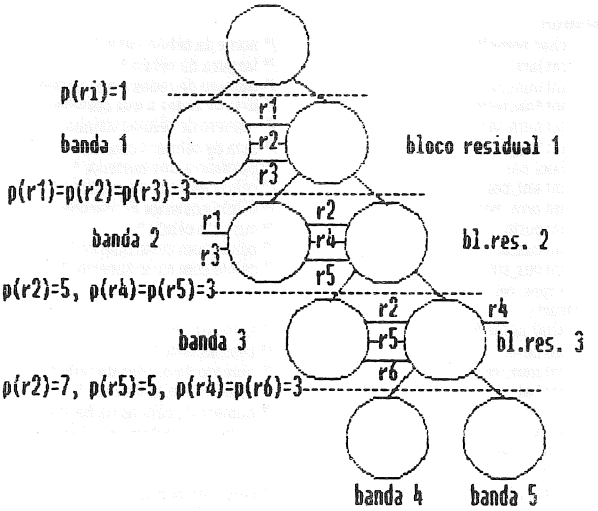


Figura 4.2 - Alteração dos pesos das redes múltiplas da interface.

5. IMPLEMENTAÇÃO

5.1. Estruturas de Dados

A escolha das estruturas apropriadas para o armazenamento de dados em um sistema é fator determinante na performance deste. Tanto que esta escolha deve ser baseada nas operações às quais os dados são submetidos durante o processo.

Em aplicações onde a velocidade de processamento é crítica, a dissociação dos dados em várias estruturas, introduzindo *redundância* de informação, apesar de onerar em termos de memória, reduz significativamente a *profundidade de laços* (laços internos a laços).

Na fase de especificação destas estruturas, a configuração da máquina que se dispõe e a quantidade e tipo de informações que serão manipuladas devem ser analisados cuidadosamente, na busca de um compromisso *memória x velocidade* que minimize as limitações do sistema.

Dentro desta filosofia, optou-se por quatro estruturas de dados para o desenvolvimento do módulo de posicionamento relativo: *vetor de células, vetor de redes, lista de pares selecionados* e *vetor de bandas geradas*. Os campos que compõem estas estruturas aparecem na figura 5.1.

```
typedef struct {
        char nome[9];                   /* nome da célula - gate */
        int larg;                       /* largura da célula */
        int num_red;                    /* número de redes a que pertence */
        int *ap_redes;                  /* lista de redes a que pertence */
        int num_viz;                    /* número de células vizinhas */
        int *ap_viz;                    /* lista de células vizinhas */
        bool part;                      /* partição a que pertence */
        int ant_part;                   /* célula anterior da partição */
        int prox_part;                  /* célula posterior da partição */
        int custo;                      /* custo da célula */
        int cus_inf;                    /* célula com custo inferior */
        int cus_sup                     /* célula com custo superior */
        } type_cel;
typedef struct {
        char nome[9];                   /* nome da rede */
        int peso;                       /* peso da rede */
        int num_cel;                    /* número de células da rede */
        int *ap_cel;                    /* lista de células da rede */
        int kb;                         /* número de células na banda */
        int kr;                         /* número de células no bloco residual */
        } type-red;
typedef struct {
        int cel_ban;                    /* célula da banda */
        int  cel_res;                   /* célula do bloco residual */
        int ganho;                      /* ganho da troca */
        int dif_larg;                   /* diferença de largura da banda */
        } type_par;
typedef struct {
        int prim_ban;                   /* índice da primeira célula da partição */
        } type_ban;
```

Figura 5.1 - Campos das estruturas de dados.

Observe que a redundância de informação está presente em vários níveis. Alguns exemplos são:

- as redes às quais uma determinada célula pertence poderiam ser determinadas percorrendo todo o vetor de redes e verificando se a célula está presente na lista de células de cada rede;

- as células vizinhas seriam obtidas percorrendo-se a lista de células de cada rede à qual a célula de referência pertence.

A partir destes exemplos, pode-se verificar o quão intrincadas seriam as operações de pesquisa. Pode-se, também, ter uma idéia da quantidade de memória que seria necessária para a codificação do procedimento e do tempo de processamento envolvido.

Observe, ainda, que alguns campos são utilizados para o *encadeamento* do vetor de células segundo critérios distintos: ordenação de células e de custos de células que pertencem à mesma partição. A ordenação de custos facilita a pesquisa dos pares que maximizam o ganho, enquanto que a ordenação de células simplifica o vetor de partições geradas, que só necessita a informação de índice da primeira célula da partição.

Neste ponto, cabe salientar que a introdução da lista de redes que estabelecem a vizinhança, para cada célula da lista de vizinhas, reduziria de 1 a profundidade de laços do procedimento, acelerando sobremaneira a execução do processo. No entanto, o número limite de células de um circuito a particionar, que possa ser tratado por uma máquina do tipo IBM-PC compatível, decresceria significativamente.

5.2. Algoritmos

Nesta seção, apresenta-se o algoritmo a nível de sistema e são feitos comentários a respeito de procedimentos relevantes de alguns subsistemas.

A figura 5.2 sintetiza todos os procedimentos envolvidos no módulo de posicionamento relativo.

1. ANÁLISE SINTÁTICA E SEMÂNTICA DA LINGUAGEM DE DESCRIÇÃO DE HARDWARE (LDH) --> GERAÇÃO DAS ESTRUTURAS INTERNAS.
2. CONSULTA À BIBLIOTECA DE CÉLULAS.
3. ENTRADA DA RELAÇÃO DE ASPECTO.
4. DETERMINAÇÃO DO NÚMERO DE PARTIÇÕES.
5. DETERMINAÇÃO DO CRITÉRIO DE BALANCEAMENTO.
6. DETERMINAÇÃO DAS PARTIÇÕES INICIAIS.
7. CÁLCULO E ORDENAÇÃO (DENTRO DA PARTIÇÃO) DOS CUSTOS DE TODAS AS CÉLULAS.
8. FORMAÇÃO DE PARES DE GANHO MÁXIMO E SELEÇÃO DA SEQÜÊNCIA DE PARES QUE MAXIMIZA O GANHO GLOBAL.
9. VERIFICAÇÃO DO CRITÉRIO DE BALANCEAMENTO E TRATAMENTO DE DESEQUILÍBRIO.
10. TROCA DAS CÉLULAS ENTRE PARTIÇÕES.
11. REALIZAÇÃO DOS PASSOS DE 7 A 11 ENQUANTO HOUVER PAR COM GANHO POSITIVO.
12. TRANSFERÊNCIAS INDIVIDUAIS DE CÉLULAS.
13. ALTERAÇÃO DOS PESOS DAS REDES DA INTERFACE E DISSOCIAÇÃO DA BANDA GERADA.
14. REALIZAÇÃO DOS PASSOS DE 5 A 14 ENQUANTO HOUVER BLOCOS A PARTICIONAR.
15. SAÍDA DAS BANDAS GERADAS.

Figura 5.2 - Algoritmo a nível de sistema.

A LDH referida no passo 1 trata-se de NILO [WAG 87b], uma linguagem estrutural no nível de portas lógicas, constante no sistema AMPLO [WAG 87a] em desenvolvimento no PGCC-UFRGS. Esta escolha foi fruto, principalmente, do fato de se dispor de outra ferramenta para suporte ao projeto (*simulador lógico*), e de NILO possibilitar uma *descrição hierárquica* no nível de REDES [WAG 87a]. Posteriormente, pretende-se adotar uma outra LDH que permita a descrição mista de portas lógicas e células que se enquadram no nível de transferências entre registradores (flip-flops, multiplexadores, decodificadores, ...).

A determinação das partições iniciais (passo 6) é regida pela definição das células que constituem a banda. A pesquisa seqüencial é encerrada quando se atinge o valor mínimo para o módulo da diferença entre o tamanho da banda e o equilíbrio (1/nb . W).

A ordenação de custos (passo 7) é realizada por um algoritmo de ordem N^2. A idéia é evoluir no sentido de ordenar os custos sobre uma *estrutura do tipo árvore*, o que reduziria a complexidade do procedimento para $O(N \log N)$.

A formação de pares de ganho máximo (passo 8) deve levar em consideração os *fatores de correção das células aos pares*. Para tanto, as células que concorrem à seleção são aquelas que possuem custo superior à diferença entre o maior custo da partição à qual pertence e a soma dos fatores

de correção das células de maior custo na banda e no bloco residual. Isto diminui consideravelmente o número de células candidatas à formação do par.

Para o tratamento de desequilíbrio (passo 9), a grandeza *desvio* é definida como sendo a distância entre o tamanho da banda, se todos os pares selecionados fossem trocados, e o limite mais próximo do critério. Cada par selecionado, cuja alteração do tamanho da banda coopera com a existência do desvio, é retirado do conjunto sucessivamente (do último ao primeiro), até que o valor atualizado do desvio se torne nulo ou negativo. Neste ponto, ficam definidos os pares que respeitam o critério de balanceamento.

A estratégia para a realização de transferências individuais de células (passo 12) baseia-se na utilização das mesmas rotinas desenvolvidas para trocas aos pares, incluindo-se provisões para o *teste do critério concorrente à seleção.*

Para maiores informações, o autor coloca a disposição dos interessados os algoritmos a nível de subsistema e o programa fonte do módulo de posicionamento relativo.

5.3. Estado Atual do Projeto

Uma primeira versão do módulo de posicionamento relativo está sendo depurada e deve tornar-se disponível muito em breve.

Esta versão foi escrita em Linguagem C e roda em máquinas do tipo IBM-PC compatível. Durante o projeto do software, teve-se o cuidado de gerar uma versão em C *padrão*, com vistas a manutenção da característica de *portabilidade* da linguagem. A próxima versão deve ser implantada em uma estação de trabalho baseada no supermicro ED680 - UCP MC68000, sistema operacional UNIX.

Alguns circuitos já foram submetidos aos procedimentos descritos. Os resultados são alentadores, tanto em termos de produto do sistema, como em termos de tempo de execução. Contudo, ainda é cedo para apresentar estatísticas que comprovem a boa performance da ferramenta, mesmo porque o projeto encontra-se na fase inicial de validação.

O que se pode afirmar, atualmente, com relação à complexidade do algoritmo implementado, é que esta não é de ordem superior a N^3. Tão logo o sistema tenha sido validado, uma análise detalhada de complexidade do algoritmo será realizada.

6. FUTURAS VERSÕES

Como a grande maioria das ferramentas baseadas em partição esbarram em soluções do tipo *mínimo local*, faz-se necessária a introdução de uma etapa de *otimização global*, cujo objetivo é atingir uma solução melhor que aquela fornecida pelo gerador de posicionamento.

Algumas propostas interessantes de procedimentos para otimização global são:

(a) repetir o procedimento Kernighan-Lin para várias partições iniciais diferentes e adotar a melhor solução;

(b) realizar bipartições sobre as partições iniciais e recombinar as quatro partições geradas em outras duas, submetendo-as, após, ao procedimento Kernighan-Lin [KER 70]; e,

(c) submeter as partições de saída do gerador de posicionamento a um algoritmo de relaxação baseado em "simulated annealing", como aquele apresentado em [SEC 85].

A primeira versão do módulo de posicionamento relativo, que abrange procedimentos para otimizações locais, mantém em aberto a etapa de otimização global das partições. Uma futura versão deve incluir provisões neste sentido. Contudo, muita pesquisa deve ser realizada na área, e um bom guia de estudos parece ser aquele indicado pelas alternativas citadas acima.

7. CONCLUSÕES

As contribuições científicas do projeto TRANCA como um todo residem em:

(a) utilizar uma nova metodologia voltada para projeto automático e baseada em um estilo de layout intermediário entre "standard cell" tradicional e "full-custom";

(b) introduzir novos conceitos e concepções em termos de posicionamento e roteamento de células, advindos das vantagens e restrições da própria metodologia de projeto em uso;

(c) avançar no sentido de possibilitar o projeto automático em níveis cada vez mais abstratos de especificação.

Dentro do contexto TRANCA, o módulo de posicionamento relativo contribui de maneira significativa com o item (b) citado acima. Mais especificamente, a colaboração deste módulo apóia-se em:

(a) busca de soluções que reduzam o tempo de processamento envolvido na execução do procedimento de bipartição;

(b) tratamento flexível do critério de balanceamento;

(c) introdução de otimizações locais na heurística Kernighan-Lin adaptada ao tratamento de circuitos;

(d) evolução no sentido de agregar ao sistema procedimentos para otimização global.

8. AGRADECIMENTOS

Este trabalho está sendo realizado sob a orientação do Professor Dr. Ricardo Reis, com o suporte de programação de André Baggio e com o apoio financeiro do CNPq - órgão brasileiro de fomento à pesquisa.

9. REFERÊNCIAS BIBLIOGRÁFICAS

[COR 79] CORRIGAN, L.I.. A Placement Capability Based on Partitioning. In: 16th Design Automation Conference, 1979, pp.406-13.

[FID 82] FIDUCCIA, C.M. & MATHEYSES, R.M.. A Linear-Time Heuristic for Improving Network Partitions. In: 19th Design Automation Conference, 1982, pp.175-81.

[HYA 73] HYAFIL, L. & RIVEST, R.L.. Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems. In: Rep. 33, IRIA-Laboria, 1973.
Apud: DUNLOP, A. E. & KERNIGHAN, B. W.. A Procedure for Placement of Standard-Cell VLSI Circuits. In: IEEE Transactions on CAD, Vol. CAD-4, No. 1, January 1985, pp.92-6.

[KER 70] KERNIGHAN, B.W. & LIN, S.. An Efficient Heuristic Procedure for Partitioning Graphs. In: Bell System Technical Journal, Vol. 49, Feb. 1970, pp. 291-308.

[LIE 85] LIESENBERG, H.K.D.. A Layout Module for a Silicon Compiler. Thesis submitted to the University of Newcastle-Upon-Tyne for the degree of doctor of philosophy. Electrical and Electronic Engineering, University of Newcastle-Upon-Tyne, April 1985, pp. 72-104.

[LUB 88a] LUBASZEWSKI, M.S. et alii. Uma Ferramenta para o Projeto de Circuitos "Standard Cell". In: III Simpósio Brasileiro de Concepção de Circuitos Integrados, Gramado, RS, Brasil, 13-15 Abril 1988, pp. 173-82.

[LUB 88b] LUBASZEWSKI, M. S. et alii. A Biblioteca de Standard Cells do Projeto TRANCA. In: III Congresso da Sociedade Brasileira de Microeletrônica, São Paulo, SP, Brasil, Julho 1988.

[REI 87] REIS, R.A.L.. A New Standard Cell CAD Methodology. In: IEEE Custom Integrated Circuits Conference, Portland, USA, May 1987.

[REI 88] REIS, R.A.L., GOMES, R.F. & LUBASZEWSKI, M.S.. An Efficient Design Methodology for Standard Cell Circuits. In: IEEE International Symposium on Circuits and Systems '88, Helsinki, Finland, June 1988.

[SCH 72] SCHWEIKERT, D.G. & KERNIGHAN, B.W.. A Proper Model for the Partitioning of Electrical Circuits. In: 9th Design Automation Workshop, June 1972, pp. 57-62.

[SEC 85] SECHEN, C. & SANGIOVANNI-VINCENTELLI, A.. The TimberWolf Placement and Routing Package. In: IEEE Journal of Solid-State Circuits, Vol. SC-20, No. 2, April 1985, pp. 510-22.

[WAG 87a] WAGNER, F. R., FREITAS, C. M. D. S. & GOLENDZINER, L. G.. Linguagens de Descrição de Hardware para Suporte à Integração do Processo de Projeto em AMPLO. In: Relatório de Pesquisa 65, Pós-Graduação em Ciência da Computação/UFRGS, Porto Alegre, RS, Brasil, Março 1987.

[WAG 87b] WAGNER, F. R. & FREITAS, C. M. D. S.. NILO - Uma Linguagem para a Descrição de Hardware no nível de Portas Lógicas. In: Relatório de Pesquisa 66, Pós-Graduação em Ciência da Computação/UFRGS, Porto Alegre, RS, Brasil, Março 1987.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

TITULO : Una herramienta para la minimización de desperdicios en el corte de chapas

AUTOR : César Leodegar Cabello.

DIRECCION: Colegio Nacional San Alfonso
Carapeguá - Paraguay

RESUMEN : En este artículo se presenta el esquema de un sistema de minimización de desperdicios en el corte de formas cualesquiera, sobre formas rectangulares.

El proceso seguido es una combinación de heurística y programación lineal de acuerdo a lo que sigue:

1) Carga de datos gráficos y numéricos.
2) Obtención de rotaciones y reflexiones básicas de las figuras.
3) Obtención heurística de "buenos" rectángulos, con las formas a cortar
4) Obtención de un conjunto de rectángulos con mínimo desperdicio y que cuenta con todas las formas a cortar
5) Obtención de las combinaciones verticales, de los rectángulos obtenidos en el paso anterior.
6) Obtención de las combinaciones que minimiza el desperdicio.
7) Graficado de los resultados obtenidos.

El sistema fué desarrollado con el objeto de servir como instrumento para la obtención de buenas distribuciones, que puedan ser, eventualmente, fácilmente modificadas o mejoradas y permitir analizar varias alternativas.

1) CARGA DE DATOS GRAFICOS Y NUMERICOS:

Para una herramienta de esta naturaleza es de fundamental importancia contar con un medio que permita la carga de las figuras a cortar de una manera versátil, ya que de otro forma podría dificultarse la tarea del usuario, teniendo en cuenta la gran variedad e irregularidad de las figuras que aparecen en la práctica.

En el caso concreto de este trabajo la carga de las formas y de las chapas matrices es realizado utilizando un paquete de diseño gráfico llamado Autocad, de la firma Autodesk Inc.

El mencionado paquete permite el dibujo de las formas en escala natural y además asignarle un nombre a las figuras realizadas; así para la operación del sistema, el usuario deberá dibujar las formas a cortar y las chapas matrices en Autocad, para posteriormente en el sistema, expresar el núumero de cortes de cada figura, referenciada por su nombre.

A fin de no desviar la atención de la esencia del trabajo y mantenerse dentro de los límites asignados, se soslayará la operación con Autocad y habrá de considerarse que las figuras ya han sido cargadas y se encuentran expresadas como un conjunto de coordenadas cartesianas.

Inicialmente, por las características del medio de entrada, las coordenadas estan referidas a un sistema de referencia externo.

Si bien este conjunto permite una identificación completa de la figura, no es de gran utilidad en el proceso posterior de encastre para la obtención de "buenos" rectángulos -3- , por lo que se hace necesario un sistema de referencia con origen en la misma figura; la adoptada para este trabajo está referida a la esquina noreste. Las razones de esta elección aparecen en la exposición del proceso - 3 -.

Las chapas matrices si bien son dibujadas no requieren ser expresdas como conjunto de coordenadas, sino solo se requieren sus dimenciones puesto que el sistema opera únicamente con chapas matrices rectangulares.

Al concluir esta etapa el sistema cuenta con los siguientes datos:

Conjunto de coordenadas normalizadas de las figuras a cortar.
Número de cortes de cada una de las figuras.
Dimensiones de la chapa matriz.
Areas de las figuras a cortar.
Cantidad disponible de la chapa matriz.

2) ROTACIONES Y REFLEXIONES BASICAS DE LAS FORMAS A CORTAR.

En esta fase se obtienen las disposiciones básicas ds las formas a cortar, que habrán de considerarse en la etapa siguiente de encastre. En principio dichas disposiciones son infinitas, por lo que se impone establecer un limite práctico; esta podría ser por ejemplo efectuar un incremento de 10 grados en las rotaciones sucesivas, a partir de la posición original.

Debido a la obtención de resultados adecuados, sin desmedro de la velocidad de cálculo, en este trabajo los movimientos de las figuras se han reducido a rotaciones, con incremento de 90 grados, y a reflexiones; en total 8 posiciones básicas. En lo sucesivo cuando se hable de posiciones de una figura deberá entenderse que se hace referencia a alguna de estas 8 posiciones.

Para las rotaciones y reflexiones se utilizarán las siguientes transformaciones: Rotación X' = Y Y' = -X
Reflexión X' = -X Y' = Y

Una vez obtenida las rotaciones y reflexiones básicas, deben distinguirse las disposiciones realmente diferentes, pues según puede observarse en el caso de un rectángulo, por ejemplo, de todas sus rotaciones y reflexiones básicas tan solo dos son diferentes.

A fin de identificar a las disposiciones repetidas se normalizarán las coordenadas obtenidas, según el principio ya presentado, y luego se comparan los conjuntos de coordenadas normalizdas entre sí, en caso de existir conjuntos iguales, se conserva uno de ellos y se desechan los demás; de esta forma se obtienen las disposiciones fundamentales.

Al concluir esta etapa el sistema cuenta con la siguiente información. Conjunto de coordenadas normalizadas de las disposiciones básicas de las figuras a cortar.

En este punto conviene realizar la aclaración de que si bien son 8 las posiciones básicas consideradas en el gráfico de distribución, el usuario puede realizar los cambios o agregados que considere adecuados antes de proceder a la impresión final, utilizando las ventajas proporcionadas por el uso del Autocad. Por otra parte la restricción a ocho en el número de posiciones no es de método, sino de práctica.

3) OBTENCION HEURISTICA DE "BUENOS" RECTANGULOS.

Para comprender cláramente la necesidad de esta etapa, debe mencionarse el hecho de que la idea básica que subyace a este trabajo es la reducción del problema de optimización en el corte de formas cualesquieras al de optimización en el corte de formas rectangulares. Esta reducción se consigue mediante el encastre de las figuras de modo a formar con ellas, lo más ajustadamente posible, rectángulos.

Llevado al extremo de considerar toda la chapa matriz, la formación de "buenos" rectángulos con tal tamaño es ya la solución al problema, pero basta una rápida observación para cerciorarse de la inmensa barrera combinatoria que surge en este punto, para cuya superación se acude en este trabajo a las probabilidades, junto a un algorítmo que utiliza la siguiente estructura, fundamental, de datos.

Matriz de chapa: Esta matriz representa en la máquina a la chapa, sobre ella se realizan los encastres; en este proceso se tienen en cuenta las siguientes convenciones para las posiciones de la matriz: 0 posición libre, 1 posición ocupada por una figura, 2 posición libre, pero verificada para un encastre.

Matriz de figuras aceptadas: esta matriz guarda las coordenadas de la esquina noreste de las figuras admitidas en la formación de los rectángulos, así como un identificador de dichas figuras. Los datos de esta matriz permitirán la construcción del gráfico final de distribución.

Matriz de parámetros de los rectángulos: esta matriz guarda un conjunto de datos útiles sobre los rectángulos formados: dimensiones, figuras que lo conforman, número de los mismos, error cometido al circunscribir un rectángulo al encastre formado. Los datos de esta matriz permitirán la formación de los sistemas de ecuaciones.

Vector de probabilidades: este vector contiene un índice que señala la probabilidad de que determinadas figuras forme parte de los encastres. Inicialmente dichos índices son iguales al número de cortes requeridos de cada figura, y van decrementándose de a uno a medida que dichas figuras vayan apareciendo.

Además de las estructuras citadas son utilizadas las siguientes variables:

Error de ajuste: como es sabido, el objetivo de esta etapa es la conformación con las figuras a cortar de rectángulos o algo lo más ajustadamente cercano a dicha forma. Lo más frecuente es que no se consiga un rectángulo, sino algo aproximado, cuya medida de error, expresada en unidades de áreas, se halla en esta variable.

Indicador de profundidad de error: en el proceso de encastre podría suceder que el agregado de nuevas figuras no mejore la aproximación a una forma rectangular, sino por el contrario lo empeore, esta situación podrá eventualmente revertirse posteriormente, sin embargo es necesario poner una cota a la secuencia de agregado de nuevas figuras. Esta variable señala el número de figuras ultimamente agregadas sin mejorar el ajuste a un rectángulo. La constante LIMITE DE PROFUNDIDAD ESTIMADA señala el valor máximo admisible para la profundidad de error.

Indicador de posibilidad de encastre: esta variable binaria señala la posibilidad o imposibilidad de realizar nuevos encastres; la imposibilida es debido a que fue ocupado todo el espacio disponible de la chapa matriz.

PROCESO DEL ENCASTRE.

Obtención de coordenadas extremas de las figuras. Este paso se efectúa con el objeto de reducir el número de comparaciones durante la ejecución de los encastres y para facilitar la obtención de dimensiones de los rectángulos circunscritos. Las coordenadas extremas utilizadas son las equivalentes a los siguientes puntos cardinales: Nor-este, Nor-oeste, Sur-este, Sur-oeste, Este-norte, Este-sur, Oeste-norte, Oeste-sur.

Tantas veces como formas a cortar haya. Repetir.
  Tantas veces como aspectos tenga la forma a cortar. Repetir.

    Inicializar matriz de probabilidades
    Inicializar matriz de chapas

```
Ubicar la disposición j de la forma i sobre la chapa.
Circunscribir un rectángulo a la figura.
Obtener error de ajuste.
Guardar en una matriz temporal de figuras aceptadas la coordenada de la
esquina NE y un indicador de la figura.

Mientras Sea posible realizar encastres Y Profundidad de error < Límite de
  profundidad estimada Y Error de ajuste > 0. Repetir.

  Realizar elección aleatoria de la forma a encastrar.

  Tantas veces como coordenadas/n existan en la periferia del encastre
    actual. Repetir.

    Para cada coordenada elegida de la periferia; circunscribir un
      rectángulo a la figura obtenida ubicando la esquina NE de la forma
      elegida, en su disposición actual, en la posición mencionada y
      considerando el estado actual del encastre.

    Obtener error de ajuste.
    Si error de ajuste es menor al menor actual de los errores de ajuste.
      Para la forma elegida últimamente.

      Actualizar valor del menor error de ajuste. Para la forma elegida
        últimamente.
      Guardar coordenadas de la esquina NE.
      Guardar nombre de la forma y su disposición.

    Fin

  Fin

  Si fué posible ubicar la forma sobre la chapa.

    Guardar en una matriz temporal de figuras aceptadas la coordenada de
      la esquina NE, y un indicador de la forma y disposición concomitante
      al menor actual de los errores de ajuste. Para la forma últimamente
      elegida.

    Ubicar la forma elegida sobre la chapa.

    Decrementar índice de probabilidad de elección.

    Si error de ajuste del nuevo encastre es mayor al de encastre anterior.

      Incrementar indicador de profundidad de error.

    Si no.

      Poner a cero indicador de profundidad de error.

      Si error de ajuste es menor o igual al menor de los errores de
        ajuste. Para los encastres.
```

Actualizar valor del menor error de ajuste. Para los encastres. Guardar coordenadas de la esquina NE y nombre de la figura para el que que se cumplió la comparación.

Fin

Fin

Fin

Fin

Transpasar el contenido de la matriz temporal de figuras aceptadas a la matriz de figuras aceptadas. Hasta el punto señalado por el menor error de ajuste, para encastres.

Cargar matriz de parámetros de los rectángulos.

Fin
Fin.

La expresión, Tantas veces como coordenadas n existan en la periferia del encastre actual. Repetir; señala uno de los aspectos fundamentales de la estrategia seguida en la solución al problema, la cual puede resumirse en lo sgte: Encastre de las figuras a cortar de modo a intentar formar con ellas rectángulos, las que a su vez serán distribuidas, en forma óptima sobre las chapas utilizando programación lineal.

A fin de asegurar que cada figura, y todas sus disposiciones, aparezca al menos una vez entre los encastres generados, empezar cada nuevo encastre, con una figura en disposición diferente, en el orden señalado por los dos ciclos iniciales del algoritmo anterior, cada nueva figura a formar parte del encastre es elegida en un sorteo en la que las figuras con mayor número de cortes esperados poseen mayor posibilidad, en la medida que no hayan sido elegidas aún.

El lugar preciso en el cual habrá de ser añadida la figura elegida, al encastre, se efectúa haciendo un recorrido de la periferia del encastre en los que se van probando el grado de ajuste a un rectángulo, se escoge la posición que haya tenido el último menor error de ajuste. Este recorrido podría incluir todas las coordenadas de la periferia, sin embargo dicha restricción impone un incremento del tiempo de operación sin un aumento sustancial en la mejoría de los resultados, según pudo observarse, por lo que la prueba se efectúa con un paso de n, el cual depende de las formas, pero en general se obtienen buenos resultados con tres. El encastre así formado va creciendo en tanto sea posible realizar nuevos encastres y profundidad de error < límite de profundidad estimada y error de ajuste > 0.

El número de encastres a obtener según el esquema anterior es igual al producto del número de formas por el de disposiciones. Esta cifra podría resultar pequeña, por lo que en este trabajo se agregó un ciclo adicional. El límite de este ciclo se toma en base a consideraciones prácticas de tiempo de respuesta; por supuesto cuanto mayor sea el número de rectángulos generados tanto más probable habrá de ser la obtención de resultados convenientes.

Al finalizar esta etapa, el sistema cuenta con los parámetros de n rectángulos, formados por encatres entre las figuras a cortar.

4) OBTENCION DE UN CONJUNTO DE RECTANGULOS CON MINIMO DESPERDICIOS, QUE CUENTA CON TODAS LAS FORMAS A CORTAR.

El número de rectángulos obtenidos en el paso anterior resulta muy elevado para su empleo en la etapa siguiente, por lo que se impone una selección de los mismos. Dicha selección se realiza en base a la necesidad que se tiene de contar en los rectángulos escogidos con todas las figuras a cortar, en la proporción requerida y al mismo tiempo minimizando el desperdicio total que existe en los rectángulos.

Para la consecución de los mencionados objetivos el problema se planteó en doble etapa.

a) Elección de un subconjunto de rectángulos convenientes.
b) Obtención, mediante programación lineal, de un subconjunto óptimo de rectángulos.

La necesidad de la fase a) se hace evidente cuando se considera que de no ser por ella, la cantidad de rectángulos a ser considerados en el sistema de programación lineal obstaculizaría el objetivo de obtener respuestas rápidas. A fin de resolver el problema se ordena el conjunto de rectángulos por un índice con rango [0,1], calculado para cada rectángulo, bajo las siguientes condiciones.

El índice mencionado es la suma de dos indicadores; un indicador del error de ajuste del rectángulo y un indicador del error de proporción. El significado atribuído al primero de estos índices ya resulta conocido, el segundo señala el desvío de la proporción de formas presentes en el encastre, de la proporción de cortes requeridos.

El índice del error de ajuste es calculado dividiendo el mínimo de los errores de ajuste, del conjunto de los rectángulos, por el error de ajuste del rectángulo en consideración y luego multiplicando el resultado por 0.7. Esto último señala que se le atribuye una importancia del 70% al error de ajuste.

El indicador del error de proporción es calculado mediante la siguiente fórmula: 1-(i(Pi-Ci)2/ iCi2), cuyo resultado es multiplicado por 0.3; lo que equivale a asignarle a este indicador una importancia del 30%. En la fórmula se utilizan las siguientes convenciones.

i = 1,...,número de figuras a cortar.
Pi Frecuencia de la figura i en el rectángulo.
Ci Cantidad de cortes esperados de la figura i.

Ordenados los rectángulos, de mayor a menor, por la clave descrita, se eligen los n primeros que permitan un adecuado tratamiento mediante un sistema de programación lineal; previa verificación de la existencia en el subconjunto escogido de todas las figuras a cortar, y si así no fuere el agregado de los mejores rectángulos, de acuerdo al índice, que permita el cumplimiento de esta condición.

En la fase b) de esta etapa, se realiza la elección del número óptimo de rectángulos mediante la resolución del siguiente sistema lineal.

Min $\sum_i$ Ei Xi

Sujeto a

$\sum_i$ Fij Xi >= Cj

Donde
i=1,..., número de rectángulos.
j=1,..., número de figuras a cortar.
Ei es el error de ajuste del rectángulo i.
Xi es el número de rectángulos i requeridos.
Fij es la frecuencia de aparición, en el rectángulo i de la figura j.
Cj es la cantidad esperada de cortes de la figura j.

Al finalizar esta fase el sitema cuenta con la información de cuales rectángulos y cuantos de ellos se requieren para conseguir todos los cortes requeridos, con mínimo desperdicio dentro de los rectángulos.

El problema de la distribución óptima de los rectángulos elegidos, en la etapa anterior, sobre las chapas matrices, se planteó como el siguiente problema de programación lineal.

Min $\sum_i \sum_j$ ((Ai * Xij- $\sum_k$ ak * lk * Rkij * Ykij) / Ai * Cij) * Xij

Sujeto a

$\sum_i \sum_j$ Rkij * Ykij >= NRk (1)

Xij >= lk * Ykij (2)

$\sum_j$ Xij <= Li (3)

Donde
i=1,..., número de chapas matrices.
j=1,..., número de combinaciones verticales de los rectángulos.
k=1,..., número de rectángulos.
Ai es el ancho de la chapa i.
Xij largo ocupado por la combinación j sobre la chapa i.
ak ancho del rectángulo k.
lk largo del rectángulo k.
Rkij número de repeticiones transversales del rectángulo k, en la chapa i, dentro de la combinación j.
Ykij número de repeticiones longitudinales del rectángulo k, en la chapa i, dentro de la combinación j.
Ek error de ajuste del rectángulo k.
NRk número requerido de cortes, del rectángulo k.
Li largo de la chapa i.

En la función objetivo , Ai * Xij representa la superficie total utilizada por la combinacion j en la chapa i; ak * lk * Rkij * Ykij representa la superficie cubierta por la combinación j en la chapa i. Así, la función

objetivo representa el porcentaje de desperdicio generado por la combinación j sobre la chapa i, ponderada por el largo de la combinación.

La restricción (1) expresa la necesidad de cumplir con el número exigido de rectángulos, las siguientes restricciones las condiciones que debe cumplir el largo de cada combinación.

La resolución de este sistema corresponde a lo que en el Resúmen se denominó, Etapa de obtención de las combinaciones que minimiza el desperdicio.

Esta resulución exige el cálculo previo de Rkij, lo cual depende de lo que se nombró como:

6) OBTENCION HEURISTICA DE COMBINACIONES VERTICALES.

Este problema podría encararse como el de la generación de un árbol; pero, si bien este enfoque permite una solución teórica completa, presenta serios inconvenientes prácticos debido al elevado número de rectángulos que deben ser distribuidos, y en el caso de reducir el número de ramas para mantener el caso dentro de límites manejables, la decisión de lo que debe desecharse es, apriori, aleatoria. Por este motivo se decidió probar la generación de las combinaciones mediante una adaptación del proceso de encastre a este caso. Los resultados obtenidos fueron satisfactorios, por esto y las ventajas de simplicidad y velocidad que representan se optó por este método. También aqui se presenta el inconveniente de que el número de combinaciones obtenidos resulta muy elevado para ser directamente utilizados en el sistema de programación lineal, por lo que se hace necesaria la elección previa de un subconjunto; dicha eleccion se efectúa siguiendo los mismos lineamientos de la fase a) de la Obtención de un conjunto de rectángulos con mínimo desperdicio, con la única salvedad de que el índice de ordenación se compone de los índices de error lateral de la combinación, error de ajuste de los rectángulos que componen la combinación y error de proporción; en ese orden de importancia. La eliminación de un conjunto de combinaciones no significa que con alguna de ellas no podría haberse obtenido resultados satisfactorios, sino que ello es poco probable, y la eliminación es necesaria para la reducción del problema a límites prácticos.

7) GRAFICADO DE LOS RESULTADOS.

Una vez resuelto el problema de programación lineal, de la etapa 6), se cuenta con toda la información necesaria para el graficado de la distribución final. El gráfico es obtenido en Autocad, lo cual permite introducir fácilmente cambios o hacer comparaciones entre resultados alternativos.

REFERENCIAS

[1] Gilmore, P.C. y Gomory, R.E., 'A linear programing approach to the cutting stock problem', Operations Research, Vol 9, 849- 859, 1961.
[2] Gilmore, P.C. y Gomory, R.E., 'A linear programinh approach to the cutting stock problem, part II', Operations Research, Vol 11, 863-888, 1963.
[3] Gilmore, P.C. y Gomory, R.E., 'Multistage cuttinúg stock problems for two or more dimensions', Operations Research, Vol 13, 94-120, 1965.
[4] Prawda, J., 'Métodos y modelos de la investigación de operaciones', Limusa, México, 1982.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

UNA GENERALIZACION AL PROBLEMA DEL VIAJANTE GTSP

Kesner Delia

Yankelevich Daniel

ESLAI - Escuela Superior Latinoamericana de Informatica - cc 3103 (1000) -
Buenos Aires - Argentina

RESUMEN

Se plantea una generalizacion al problema del viajante (GTSP), analizando su complejidad. Se proponen distintas soluciones aproximadas haciendo uso de la heuristica de Christofides, la estrategia Greedy, una extension al Tree-Algorithm y el metodo de Local Search.

RESUMEN

Se plantea una generalizacion al problema del viajante (GTSP), analizando su complejidad. Se proponen distintas soluciones aproximadas haciendo uso de la heuristica de Christofides, la estrategia Greedy, una extension al Tree-Algorithm y el metodo de Local Search.

1 - INTRODUCCION

Los problemas combinatorios pueden ser resueltos de una manera bastante sencilla con algoritmos exponenciales. En la practica, la mayoria de esos algoritmos tardarian miles o millones de años en hallar la solucion optima, aun en las computadoras mas poderosas.

Seria ideal encontrar algoritmos mas veloces que resuelvan esos mismos problemas, pero en muchos casos se sospecha que dichos algoritmos no existen. Es por eso, que se intenta encontrar algoritmos que obtengan soluciones aproximadas que no disten demasiado de la optima. Para esto existen diversos metodos y tecnicas.

En este trabajo, se presenta una generalizacion al problema del viajante (GTSP), que posee las caracteristicas mencionadas.

En el capitulo 2 se plantea el problema y se propone un modelo utilizando grafos. En el tercer capitulo se introducen algunos conceptos sobre complejidad y se estudia en particular la del GTSP. El siguiente capitulo muestra el algoritmo exponencial que resuelve el problema. En el capitulo 5 se presentan los tres algoritmos polinomicos desarrollados para hallar soluciones aproximadas al GTSP. Finalmente, se introduce la tecnica de local search, aplicandola al GTSP, exponiendo ademas algunos resultados teoricos obtenidos.

2 - EL GTSP

2.1 - PLANTEO DEL PROBLEMA

Una empresa debe vender sus productos en las K ciudades mas importantes del pais, y dispone para ello de N viajantes que viven en algunas de estas ciudades. No hay dos viajantes que viven en la misma ciudad. A cada uno de ellos se le asigna la cantidad de ciudades a recorrer, sin determinar cuales seran estas. Se desea vender el producto en todas las ciudades, pero un viajante no podra pasar por una ciudad recorrida anteriormente por otro viajante o por el mismo.

Se quiere determinar cuales seran las ciudades que visitara cada viajante, de tal manera que la suma de las distancias recorridas por todos los empleados de la empresa sea minima.

2.2 - MODELO

En principio el modelo mas natural del problema es un grafo simple no dirigido [Har 69], donde los nodos representan ciudades y los arcos las posibles rutas. Es posible acceder desde una ciudad a cualquier otra, por lo que el grafo sera completo.

Cada arco tiene ademas asociado un costo, que representa la distancia de una ciudad a otra. Notar que la distancia de la ciudad A a la B es la misma que de B a A.

Al viajante i se le asigna un numero K_i de ciudades a recorrer.

Resolver el GTSP es dar a cada viajante el conjunto de ciudades a recorrer y la forma de hacerlo.

Notacion:

G=(V,E) grafo completo con #V=K
A en $R^{K X K}$ tal que $A_{i,i} = A_{j,j}$ y $A_{i,i}$ = infinito, para todo i = 1..K (matriz de costos)
V' incluido en V y #V' = N (ciudades donde viven los viajantes)

los nodos de V' estan numerados de 1 a N
sumatoria $K_i = K$; $K_i >= 1$ para todo i=1..N

Una solucion sera una **particion** de V, P_1.....P_N, cada P_i asociado a un A_i tal que:

i) A_i incluido en E para todo i
ii) $A_i \quad A_j = ()$ -------> i=j
iii) $\#P_i = K_i$
iv) Para todo e=(v_1, v_2) perteneciente a A_i ; v_1 , v_2 pertenecen a P_i y existen unicos u_1 , u_2 en P_i ($u_1 <> v_2$, $u_2 <> v_1$),tales que (u_1 , v_1) y (v_2 , u_2) pertenezcan a A_i .

Tal vez resulte contraintuitivo que la diagonal de la matriz de costos contenga infinito, pero es la forma en la que se indica que un arco no puede utilizarse, ademas este planteo es util para algunas de las soluciones que se presentan.

Se podria pensar que el problema se reduce a buscar el camino hamiltoniano de costo minimo de longitud K_1, luego el de longitud K_2 en el grafo resultante de G menos los nodos del primer ciclo, y asi sucesivamente, es decir, a resolver N veces el problema del viajante.

Lamentablemente hay contraejemplos, es decir grafos en los cuales el algoritmo insinuado no da la solucion optima.

3 - COMPLEJIDAD

3.1 - ALGUNOS CONCEPTOS SOBRE COMPLEJIDAD

La clasificacion de los problemas se ha deducido de las actuales conjeturas sobre la existencia o inexistencia de algoritmos polinomicos para resolverlos. Los problemas "demostrablemente irresolubles" son aquellos para los que no puede haber ningun tipo de algoritmos, los "demostrablemente dificiles" solamente tienen algoritmos de tiempo exponencial.

Los problemas para los que se conocen algoritmos de tiempo polinomico se asignan a la clase P. El caracter de los demas problemas es menos seguro: solamente se conocen algoritmos suyos de tiempo exponencial, pero no esta demostrada la inexsistencia de algoritmos polinomicos.

Estos ultimos problemas se asignan a las clases NP y co_NP. Los problemas de la clase NP se podrian resolver con suerte en tiempo polinomico por tanteo. La clase co_NP contiene todos los problemas cuya solucion es la disyuntiva "si" o "no", y cuyas versiones complementarias "no" o "si" pertenecen a la clase NP. La clase NP_completa es un subconjunto de la clase NP compuesta por los problemas que tienen la siguiente propiedad: si uno de ellos pudiera resolverse mediante un algoritmo de tiempo polinomico, entonces todos los demas problemas de la clase NP podrian resolverse en tiempo polinomico. El descubrimiento de un tal algoritmo seria demostracion de que las clases P y NP son identicas.

Para que un problema forme parte de la clase NP no es preciso que exista un metodo eficiente de responder las preguntas afirmativa o negativamente. Lo que se quiere es que cuando la respuesta sea afirmativa exista un metodo breve y convincente que asi lo demuestre.

Al parecer todos los especialistas en ciencias de la computacion estan de acuerdo actualmente en que los algoritmos cuyo tiempo de ejecucion crece exponencialmente en funcion del tamaño de la entrada carecen de valor practico. Otro punto a tener en cuenta es que la rapidez de un algoritmo es propiedad intrinseca del mismo, y es independiente de la maquina que lo ejecute **[LePa 78]**.

Puede parecer grotesco definir un metodo matematico en terminos de felices ideas y tanteos afortunados, pero no obstante,es un metodo perfectamente legitimo de precisar cuales son los problemas que pertenecen a la clase NP. En teoria se podria incluso mecanizar el procedimiento construyendo un artificio denominado "maquina de Turing no deterministica" Este dispositivo puede efectuar todas las funciones de una maquina de Turing deterministica, pero ademas en algunos puntos, puede elegir de mas de una manera cual va a ser la etapa siguiente. La clase NP de los problemas nodeterministicamente polinomicos esta formada precisamente por aquellos problemas cuyos casos particulares de solucion afirmativa podrian ser identificados por maquinas efectuando una sucesion relativamente breve de computos por tanteo. Se considera que si la maquina no "adivino" correctamente, el tiempo invertido hasta ese momento no se toma en cuenta, y se vuelve hacia el punto de partida olvidandose de lo ocurrido. Esto tambien se podria explicar en los siguientes terminos: cada vez que se llega a una solucion no satisfactoria, se hace backtracking y se resta al tiempo de ejecucion total aquel que fue ocupado en el camino hacia dicha solucion.

La frase "algoritmo nodeterministico" fue propuesta por Floyd en 1967. En estos algoritmos Floyd alude al uso de la funcion "choice" para simplificar la descripcion exhaustiva en estrategias de busqueda. En el año 1970 Fikes describe un sistema completo de resolucion de problemas en el cual estos son especificados en un lenguaje procedural que permite el uso de las funciones choice nodeterministicas. Para que ello sea posible tambien se introdujeron los conceptos de "succes" y "failure", donde "succes" hace exitoso al algoritmo y "failure" indica un fracaso. De esta manera, un algoritmo nodeterministico puede pensarse como un algoritmo que realiza copias de si mismo al hacer un choice y una copia "muere" si ejecuta un failure, es decir, si falla **[Nil 71]**.

Se dice que un problema pertenece a la clase NP_HARD si tiene la siguiente propiedad : no se sabe de su pertenencia a la clase NP, pero si fuera NP seria NP_completo. Esto quiere decir que es por lo menos tan complicado como un problema NP_completo.

3.2 - ALGUNOS CONCEPTOS SOBRE OPTIMIZACION

DEFINICION 1:

Un problema de optimizacion es un conjunto M de instancias.Cada instancia es un par (F,c) ,donde F es un conjunto de soluciones factibles y c es una funcion de costo , c : F------> R.

DEFINICION 2:

Un problema de optimizacion combinatorio (POC) es una problema de optimizacion tal que para toda instancia (F, c) , F es finito.

De ahora en adelante se asume que F y c son dadas implicitamente en terminos de dos algoritmos A_f y A_c , ya que la cantidad de soluciones factibles puede ser muy grande.

El algoritmo A_f , dado un objeto combinatorial f y un conjunto S de parametros,decide cuando f es un elemento del conjunto F (el conjunto de soluciones factibles especificado por los parametros dados).

El algoritmo A_c , dada una solucion factible f y otro conjunto de parametros Q, devuelve el valor de c(f).

Una instancia de un POC puede ser definido entonces como la representacion de los parametros S y Q.

Ejemplo:

Una instancia del problema del viajante con N ciudades tiene como parametro S al entero N, y como parametro Q a la matriz simetrica de distancias entre las distintas ciudades. El algoritmo A_f, dado un objeto f y el parametro N, examinara cuando f es un tour de N ciudades. El algoritmo A_c, dado un tour f y la matriz simetrica de distancias, calcula el costo c(f) sumando todas las distancias atravesadas por el tour.

DEFINICION 3:

La version de optimizacion de un POC es aquella que dados parametros S y Q para los algoritmos A_f y A_c, encuentra la solucion optima factible.

DEFINICION 4:

La version de evaluacion de un POC es aquella que dados S y Q, encuentra el costo de la solucion optima.

Bajo la hipotesis que A_c es un algoritmo de tiempo polinomico (el costo c no es dificil de computar), la version de evaluacion no puede ser mas dificil que la version de optimizacion, pues dada la solucion optima, hallar su costo solo lleva tiempo polinomico.

DEFINICION 5:

La version de reconocimiento de un POC es aquella que dados S,Q y un entero L responde a la pregunta Existe una solucion factible f en F tal que c(f)<=L?

La version de reconocimiento es una pregunta que puede ser respondida por **SI** o por **NO**. Responder esta pregunta no es mas dificil que resolver la version de evaluacion, pues una vez resuelta esta, solo se debe comparar el costo optimo c(f') con L, y responder **SI** si c(f')<=L.

Por lo tanto cada una de las versiones de optimizacion, evaluacion y reconocimiento (en ese orden) no es mas dificil que las anteriores asumiendo que c es computable en tiempo polinomico.

Cabe la pregunta: Son las tres versiones de la misma complejidad?. Bajo la hipotesis de que el costo de una solucion optima es un entero cuyo logaritmo esta acotado por un polinomio en el tamaño de la entrada, se puede mostrar que la version de evaluacion puede ser resuelta eficientemente cuando se resuelve la version de reconocimiento. Para mostrar esto debemos evaluar el costo optimo c(f') formulando la pregunta Es c(g')<=L? para varios valores de L. Esto puede ser hecho mediante busqueda binaria y mediante la hipotesis que log(c(f')) esta acotado por un polinomio en el tamaño de la entrada, podemos hacer eficiente uso del algoritmo que resuelve la version de reconocimiento para resolver la version de evaluacion.

No hay un metodo general para resolver la version de optimizacion haciendo un uso eficiente del algoritmo para la version de evaluacion, pero para algunos problemas esto se pudo hacer.

3.3 - COMPLEJIDAD DEL TSP

El GTSP es una generalizacion del problema del viajante (TSP), en el sentido que esta planteado para N viajantes y si N=1 es justamente el TSP. Por lo tanto como el problema del viajante es NP_completo [Lepa 81], el GTSP es por lo menos NP_completo.

Intuitivamente la complejidad del GTSP aparece por dos lados:

1) Hallar la particion optima que cumpla las restricciones. Cada conjunto de la particion representara las ciudades que debe visitar cada viajante.

2) Hallar dentro de cada conjunto de la particion, un camino hamiltoniano de costo minimo; el camino que cada viajante debera recorrer.

TEOREMA 1:

Las versiones de evaluacion y de reconocimiento del GTSP son NP_COMPLETAS, la de optimizacion es NP_HARD.

Demostracion:

La version de evaluacion se puede formular del siguiente modo: Dados N nodos de un grafo de K nodos, y numeros $K_i > 0$, con la suma de los $K_i = K$, se quiere hallar una particion de los K nodos en N subconjuntos, tal que cada subconjunto contenga uno y solo uno de los N nodos, y tal que cada subconjunto contenga K_i nodos. Llamemos a esta condicion que ponemos a la particion condicion Cond_part.

El siguiente algoritmo nodeterministico resuelve el problema en tiempo polinomico. Este algoritmo falla si en el grafo dado no es posible hallar la particion que cumpla las condiciones o si no hay ninguna solucion de costo menor o igual que L, y tiene exito si existe una solucion de costo menor o igual a L.

```
Procedure VIAJANTE GENERALIZADO( G=(V,E):GRAFO, L:ENTERO);

var

  { V es el conjunto de los nodos del grafo }
  conj, aux_conj:conjunto de conjuntos de nodos;
  aux_V,vert:nodo;
  bool:boolean;
  costo:ENTERO;

begin
  aux_V:=choice(V);
  V:=V-aux;
  conj:{{aux}};
  while V <> {} do
  begin
    vert:=choice(V);
    bool:=choice({true,false});
    if bool then
          begin
            conj:=conj U {vert};
          end
          else
          begin
            aux_conj:=choice(conj);
            conj:=conj-aux_conj;
            aux_conj:=aux_conj U {vert};
            conj:=conj U aux_conj;
          end;
    V:=V-vert;
  end;

  {se obtuvo una particion del conjunto de nodos del grafo, en conj}

  if Cond_part then
          begin
            costo:=0;
            while conj <> {} do
            begin
              aux_conj:= choice(conj);
              costo:= costo + EVAL_TSP(SUBGRAFO_EXPANDIDO(aux_conj));
              conj:= conj-aux_conj;
            end;
          end;
          else failure;

  if costo <= L then succes
          else failure
end;
```

Es sencillo observar que este algoritmo es polinomico. La primer parte, que halla la particion, ejecuta K ciclos, ya que en cada ciclo retira un nodo del conjunto de vertices y termina cuando el conjunto esta vacio

La segunda parte es del orden de la cantidad de conjuntos de la particion, este numero es igual al numero de viajantes que esta acotado por la cantidad de nodos del grafo, es decir por K, y, para cada conjunto, se realiza la llamada a *EVAL_TSP* que es polinomico. Luego, el algoritmo es polinomico.

EVAL_TSP es un algoritmo que dado un grafo (en este caso el subgrafo expandido del conjunto de nodos pasados como parametros) devuelve el costo de la solucion optima. Este algoritmo es polinomico, ya que el problema del viajante pertenece a NP (polinomico no deterministico).

Como el problema del viajante es un caso particular de este, y el problema del viajante es NP_COMPLETO este problema es por lo tanto NP_COMPLETO (en la version de reconocimiento).

Por lo argumentado mas arriba, dado que la version de reconocimiento del GTSP es NP, tambien lo es en su version de evaluacion. La idea es la misma, y pese a no ser una prueba formal lo expuesto es suficiente para decir que el GTSP es entonces NP_COMPLETO, aun en su version de evaluacion. Es sencillo hallar un algoritmo polinomico que dada una solucion calcule su costo.

Para completar la demostracion, por lo ya señalado, es conveniente ademas pedir las siguientes condiciones:

1) Los costos de las soluciones factibles al problema tienen una cota superior. Esta cota debe ser una funcion polinomica de la entrada del problema, en nuestro ejemplo, el numero de nodos. Esto es posible teniendo, por ejemplo, una cota al costo de un arco. Si esta cota es H se puede acotar el costo de la solucion por :

$$H * \text{suma}_{i=1}^{N} K_i$$

Este numero a su vez es menor que $H * N^2$, ya que el numero de viajantes es menor que la cantidad de nodos del grafo ($\#V' < \#V$).

2) Los costos de las soluciones deben ser enteros. En realidad, alcanza con pedir que los costos sean discretizables, en la practica esto siempre sucede, ya que jamas se trabaja con un numero no acotado de decimales. Por lo tanto, siempre podemos pensar a los costos como enteros.

La version de optimizacion es NP_HARD, ya que trivialmente, si fuera NP seria NP_COMPLETO, por ser el viajante un caso particular de ella.

4 - SOLUCION EXPONENCIAL AL GTSP

Como en todo POC existe una solucion trivial, y consiste simplemente en explorar exhaustivamente todas las posibilidades.

Se hallan todas las particiones posibles del conjunto de ciudades tal que el cardinal de los conjuntos de la particion sean la cantidad de ciudades que debe recorrer cada viajante.

En cada una de estas particiones, y para cada uno de los conjuntos de ciudades de cada particion, se busca un camino hamiltoniano de costo minimo. Asi se obtiene para cada particion un costo que resultaria de sumar todos los costos de los caminos hamiltonianos minimos hallados.

Dado un algoritmo que halla particiones distintas de un conjunto de cardinal K **[NiWi 75]**, es muy facil agregar restricciones tales como :

-la particion debe contener N conjuntos.
-cada subconjunto i de la particion debe tener cardinal K_i - 1.

De las K ciudades ,solo interesan aquellas de las que no parte ningun viajante (estas son K- N). De ellas se debe ver todas las maneras de tomar K_1-1 ciudades para el primer viajante(ya que la ciudad de donde parte seguro la va a recorrer), K_2-1 para el segundo viajante ,y asi sucesivamente. Si llamamos C(t,r) al numero combinatorio 't tomados de a r', el numero de particiones que nos interesan son:

$$C(K-N, K1-1) * C(K- N.- (K_1-1), K_2 -1) * ..... * C(K_N -1, K_N -1)$$

este numero es precisamente

$$\frac{(K- N)!}{(K_1 -1)! * ..... * (K_N -1)!}$$

5 - SOLUCIONES NO OPTIMAS AL GTSP

5.1 - RAZONES PARA DISEÑAR ALGORITMOS POLINOMICOS

El GTSP dista mucho de ser solo un problema teorico. Este problema tiene un numero muy grande de aplicaciones, desde recorridos y viajes hasta tareas de scheduling, y quizas aun mas que las del viajante al ser mas general que aquel. Por este motivo es importante y necesario plantear algoritmos que aunque no den soluciones optimas, hallen algunas "buenas"

Dos tecnicas pueden ser usadas como modo de resolucion de problemas para los cuales no se conocen algoritmos polinomicos:

1-Un problema Q para el que no se conoce algoritmo polinomico es reemplazado por un problema Q' el cual tiene un algoritmo polinomico conocido que lo resuelve cuya solucion es aproximada a la de Q.

2-Un subconjunto relativamente pequeño de posibles soluciones de Q es examinado usando un criterio de seleccion razonable ,intuitivo y generalmente pobre. La 'mejor' de estas soluciones potenciales es tomada como solucion del problema Q.
[Ha 78].

5.2 - ALGORITMOS DE TIEMPO POLINOMICO PARA EL TSP

Para el TSP existen algoritmos que dan soluciones aproximadas en tiempo polinomico. Entre estos algoritmos se encuentran el GREEDY **[AHU 83]**, el TREE - ALGORITHM, y el de CHRISTOFIDES **[Pap 82]**.

DEFINICION 6:

Sea A un algoritmo de optimizacion con funcion de costo positiva C, sea A' otro algoritmo que da una solucion factible F' y sea F la solucion optima que da el algoritmo A. Diremos que A' es K-aproximado para A (K>=0) si y solo si

$$\frac{c(F')-c(F)}{c(F)} <= K$$

TEOREMA 2: El algoritmo TREE_ALGORITHM es 1-aproximado **[Pap 82]**.

TEOREMA 3: El algoritmo de Christofides es 1/2-aproximado **[Pap 82]**.

TEOREMA 4:

A menos que P = NP no hay algoritmos polinomicos e-aproximados para el problema del viajante para ningun e > 0.

Es necesario observar que en los algoritmos no optimos citados para el TSP, se supone que los grafos sobre los que se van a aplicar los algoritmos cumplen la desigualdad triangular, dicho de otra forma, que los costos de los arcos tienen las propiedades de la distancia euclidea. La desigualdad triangular puede plantearse como una restriccion a la matriz de costos A pidiendo que $A_{i\,j} + A_{j\,k} >= A_{i\,k}$ para cualquier tripla i, j, k. En algunos casos se pide la desigualdad estricta.

Este pedido, sin embargo, no se requiere para probar el algoritmo (es decir, para que funcione) sino para garantizar que la solucion sea "buena" en algun sentido. Se pide que sea k-aproximada (k finito), o por lo menos, en caso de no ser posible acotar la solucion en funcion del optimo, que en la mayoria de los casos de un resultado no muchas veces mayor que el optimo.

Sin la propiedad de la desigualdad triangular es posible hallar grafos para los cuales estos algoritmos encuentren una solucion de costo tan grande como se quiera. Intuitivamente, bastaria con darle a alguno de los arcos que el algoritmo elige en forma obligada (sin tener en cuenta su costo), un costo tan grande como se quiera.

Es interesante mencionar que en los casos en los que se cumple esta propiedad, valiendo ademas el mayor estricto, cosa que sucede siempre con las distancias euclideas, el circuito minimo resulta hamiltoniano, aunque no se lo exija.

Agregar otras restricciones, como por ejemplo limitar el grado de los nodos del grafo no resulta demasiado util:

TEOREMA 5:

El problema del circuito hamiltoniano es NP-COMPLETO aun si el grafo cumple la restriccion de tener nodos solo de grado cuatro o menor.

TEOREMA 6:

El problema del circuito hamiltoniano para grafos con todos los nodos de grado 3 es NP-COMPLETO .

[Pap 82]

Pese a ser restricciones muy fuertes que eliminarian la mayoria de las aplicaciones, no ayudan tanto como para que valga la pena adoptarlas.

5.4 - ALGORITMOS DE TIEMPO POLINOMICO PARA EL GTSP

La primer desicion tomada, es, entonces, restringirse a los grafos cuya matriz de costo respete la desigualdad triangular.

5.4.1 - PRIMERA SOLUCION

La primera de estas soluciones fue una generalizacion del algoritmo greedy para el TSP **[AHU 83]** que dio como resultado un algoritmo greedy para el GTSP. Este algoritmo tambien esta basado en los de Kruskal y Prim de busqueda de arboles expandidos de costo minimo **[AHU 83]**.

1-Crear para cada viajante una componente conexa que solo contenga a la ciudad donde el viva.
2-Ordenar los arcos del grafo de menor a mayor.
3-Mientras exista algun viajante i tal que su componente tenga menos de K_i vertices hacer:
  3.1-Elegir la arista L=(U,V) tal que:
    - U y V no esten en la misma componente, ni en distintas (sin perdida de generalidad U esta en una componente y V no)
    -Sea de costo minimo
    - No pertenezca a ninguna componente
    - grado(U) <2 en el arbol expandido
    -La componente a la cual pertenece U aun admita vertices
  3.2-Agregar V a dicha componente
  3.3-Agregar L a dicha componente
4-Agregar el arco que forme un ciclo en cada una de las componentes. Notar que este queda univocamente determinado.

5.4.2 - SEGUNDA SOLUCION

La idea consiste en hallar una particion prometedora de alguna manera, y luego para cada elemento de la particion aplicar el TREE ALGORITHM.

1-Ordenar los arcos del grafo de menor a mayor.
2-Colocar a cada nodo de V' en una nueva componente.
3-Ir agregando los arcos a las componentes hasta que todas tengan la cantidad de nodos deseada, es decir hasta que cada una tenga cardinal K_i, usar las siguientes restricciones:

3.1-Si el arco une dos componentes distintas o
3.2-Si el arco une dos nodos de una misma componente y forma ciclo dentro de esta o
3.3-Si el arco incorpora a la componenete un nodo y entonces el cardinal de esa componente supera el numero deseado (K_i), no se agrega el arco
3.4-cualquier otro caso, se agrega el y se lo quita del conjunto de arcos a revisar

(se obtuvo una particion del conjunto de vertices del grafo,
y para cada conjunto de la particion se obtuvo un arbol expandido T
de costo minimo)

4-para cada conjunto de la particion hallada hacer:
4.1-crear un multigrafo G donde por cada arco e=(x1,x2) de T crear los arcos e'=(x1,x2) e''=(x2,x1).
4.2-encontrar un camino euleriano de G.
4.3-encontrar un tour en dicho camino euleriano.

5.4.3 - TERCERA SOLUCION

Aqui la idea tambien consiste en hallar una particion prometedora, pero luego para cada elemento de la particion aplicamos el algoritmo de Christofides. Los puntos 1, 2 y 3 son los mismos que antes.

1- 2- 3

(se obtuvo una particion del conjunto de vertices del grafo,
y para cada conjunto de la particion se obtuvo un arbol expandido T
de costo minimo)

4-para cada conjunto de la particion hallada hacer:
4.1-sea T'=los vertices de T que tengan grafo impar.
4.2-encontrar el minimo matching completo M en el grafo completo compuesto solamente por los vertices T'.
4.3-sea G el multigrafo con vertices=(1,2,...,n) y arcos=arcos de T y arcos de M.Notar que los vertices de G son los mismos que los del grafo original.
4.4-encontrar un camino euleriano en G.
4.5-encontrar en dicho camino euleriano un tour.

5.4.4 - TIEMPO DE EJECUCION

La tres soluciones son de orden $O(K^3)$.
El tree algorithm es de orden $O(K^2)$ y el de Christofides es de $O(K^3)$ **[Pap 82]**.
La cantidad de arcos en un grafo completo de K nodos es $K * (K - 1) / 2 = a$.
Usando estos resultados se hallan los ordenes de cada paso de cada uno de los algoritmos presentados.

primera solucion:
1- O(N), ya que hay N viajantes
2- $O(2\ a \log a) = O(a * \log a)$, es el tiempo necesario para ordenar una lista de a elementos.
3 - El cuerpo del Mientras es de orden O(a), ya que cada vez se busca secuencialmente en la lista. En el peor caso, se debera recorrer toda la lista de a elementos. Este cuerpo se ejecuta K veces, ya que cada vez se agrega un arco y se termina cuando en ninguna componente se pueden agregar mas arcos, esto es, sumatoria (K_i) veces y sumatoria $(K_i) = K$. Por lo tanto el orden es $O(K * a)$.
4 - O(N), ya que hay N componentes.

El orden de este primer algoritmo es entonces $O(\text{maximo}(a * \log a; K * a)) = O(\text{maximo}(K^2 * \log K^2, K^3) = O(K^3)$.

segunda y tercera solucion:

1 - O(a * log a)
2 - O(N).
3 - O(K * a), por lo mismo que en el punto 3 del algoritmo anterior.

Para cada elemento de la particion se aplica alguno de los dos algoritmos propuestos, por lo tanto si un elemento de la particion tiene K_i vertices y el orden del TREE ALGORITHM o del algoritmo de CHRISTOFIDES es $O(K_i)$, el orden total es la suma de los ordenes $O(K_i)$.

En la segunda solucion el orden total es entonces $O(\text{maximo}(K^3, \text{sumatoria } K_i^2) = O(K^3)$, ya que $K_i < K$ y $N < K$.

En la tercera solucion el orden total resulta $O(\text{maximo}(K^3, \text{sumatoria } K_i^3) = O(K^3)$, ya que los K_i son enteros positivos y por lo tanto la sumatoria $K_i^3 <= K^3$.

6 - LOCAL SEARCH

Local Search es un metodo que sirve para hallar soluciones aproximadas a problemas de optimizacion. Consiste en hallar una solucion inicial y a partir de ella buscar soluciones cercanas (en algun sentido) que permitan mejorarla. Al conjunto de soluciones cercanas a una solucion dada se lo llamara vecindad.

La intencion de definir optimos locales en la vecindad de una solucion es buscar entre las cercanas a ella la mejor.

Una posibilidad es definir la vecindad de una solucion como todas las soluciones posibles. La opuesta es definirla como el conjunto formado por ella misma. Entre estos dos extremos pueden elegirse muchisimas vecindades, estas deben ser sencillas de explorar pero no demasiado pequeñas, de manera que el optimo local sea lo bastante bueno.

Puede ser conveniente elegir una vecindad pequeña, aun a riesgo de revisar varias vecindades antes de hallar una solucion buena, o trabajar con vecindades grandes, pagando el costo de un mayor tiempo de busqueda.

Todos estos conceptos de LOCAL SEARCH se apoyan en la practica. Solo mediante experimentacion puede resolverse si una vecindad es mejor que otra o si una solucion inicial o conjunto de soluciones iniciales es mejor que otro.

6.1 - CONCEPTOS GENERALES

DEFINICION 7:

Un optimo global de una instancia (F,c) es un f perteneciente a F tal que $c(f) <= c(y)$ para todo y perteneciente a F.

Si (F_1, c_1) es una instancia del GTSP, F_1 es el conjunto de todos los ciclos que cumplen las restricciones para un grafo dado y la funcion c_1 de costo esta dada por la suma de los costos de los arcos de la solucion factible. El GTSP es el conjunto de todas las instancias correspondientes a grafos completos con funciones de costo simetricas que cumplen la desigualdad triangular.

DEFINICION 8:

Dado un problema de optimizacion Z con instancias (F_Z, c_Z) una vecindad es una funcion N: F_Z --> partes(F_Z) , definida para cada instancia.

DEFINICION 9.

Dada una instancia (F_Z, c_Z) de un problema de optimizacion Z y una vecindad N para dicha instancia, una solucion factible f de F_Z sera un optimo local con respecto a N si $c(f) <= c(y)$ para todo y perteneciente a N(f).

Ejemplo:

Para el problema del TSP se define la vecindad 2-change como $N_2(f)$ = { g / g pertenece a F y g puede obtenerse de f removiendo dos arcos del ciclo y reemplazandolos por dos arcos }

Se podria generalizar esta idea para una vecindad R-change.

TEOREMA 7:

Para el TSP el optimo local de N_2 no es necesariamente el optimo global pero el optimo local de N_K (K numero de vertices) es el optimo global.

[Pap 82]

DEFINICION 10:

Una solucion local optima para el TSP en una vecindad R-change se llama R-opt.

Para hallar una solucion R-opt, se define la funcion MEJORAR:

MEJORAR(f) = si existe s perteneciente a $N_R(f)$ tal que c(s) < c(f) entonces s
sino 'no'

Un algoritmo para hallar un ciclo R-opt es entonces:

```
procedure R-opt (var t)
  t= alguna solucion inicial;
  while MEJORAR(t) <> 'no' do
     t= MEJORAR(t);
```

Este algoritmo busca el optimo en una vecindad.
El problema ahora es

a) encontrar una vecindad adecuada
b) dar una solucion inicial (o mas)

Por otro lado, se tiene la posibilidad de buscar pocos optimos locales en vecindades muy grandes, o buscar muchos en varias vecindades pequeñas.

Las vecindades R-change fueron estudiadas para el TSP, haciendo enfasis en N_2 y N_3 **[Bo 58, Cr 58, Li 65, HK 62]**, con las siguientes conclusiones empiricas:

a) Las vecindades 2-change y 3-change resultaron muy buenas y relativamente sencillas de utilizar.

b) La vecindad 4-change resulto mejor que la 3-change, pero las mejoras son tan pequenas que no conviene enfrentar el costo mayor de busqueda.

c) La vecindad 3-change resulta tan poderosa, que en la practica es casi lo mismo comenzar con una solucion 'buena' (aproximada) que con una hallada al azar. Esto es completamente contraintuitivo, en general se esperaria que empezando con soluciones buenas se consiguieran aproximaciones mejores.

d) La probabilidad (hallada empiricamente) de que una solucion 3-opt para un problema de 48 ciudades dado sea optimo global, es aproximadamente de 0.04. Luego, para 100 corridas, comenzando de soluciones 'al azar' la probabilidad de hallar el optimo global es de 0.99

e) La probabilidad (hallada empiricamente) de que una solucion 3-opt sea el optimo global es aproximadamente de $2^{-N/10}$.

Como conclusion de lo anterior, se ve que para el TSP es efectivo utilizar soluciones iniciales elegidas al azar, que 3-change es una vecindad buena y que muchas corridas ayudan mucho para encontrar una solucion aproximada. El exito obtenido con el TSP impulso a utilizar esta tecnica con otros problemas. Sin embargo, en otros problemas en los que se deben definir vecindades mas debiles, puede resultar conveniente comenzar con una solucion buena, tal vez hallada con un algoritmo de aproximacion. Lamentablemete, todos los resultados en este area no estan apoyados sobre una base teorica, por lo que toda hipotesis debe justificarse empiricamente.

El parecido entre el problema del viajante y el GTSP y la posibilidad de encontrar con relativa facilidad una buena solucion inicial con los algoritmos propuestos anteriormente, llevan a contemplar la posibilidad de utilizar este metodo para el GTSP.

6.2 - SOLUCIONES INICIALES AL GTSP

a-Para el problema del TSP es efectivo usar soluciones iniciales elegidas al azar **[Pap 82]**. Otra posibilidad seria entonces generar varias soluciones al azar para el GTSP y tomarlas como soluciones iniciales.

b-Una posibilidad es tomar como soluciones iniciales aquellas que se obtienen por los algoritmos no optimos ya propuestos. Esto daria tres soluciones iniciales (Greedy, tree algorithm y Christofides generalizado), o menos pues podrian coincidir algunas.

c-Otra posibilidad es generar las x primeras soluciones con el algoritmo exponencial. Pero las obtenidas en a seran en la mayoria de los casos mejores con respecto al costo.Por otro lado esta idea no da soluciones totalmente al azar, ni soluciones 'buenas', ni soluciones demasiado lejanas, por lo tanto esta tecnica no se tomara en cuenta.

6.3 - VECINDADES PARA EL GTSP

a-La primera propuesta es hallar la vecindad 2-change o 3-change para cada subconjunto de una particion.
El cardinal de 2-change es:

$$\#N_2(f) = \text{sumatoria}_{i=1}^{N} [K_i * (K_i - 3)] / 2$$

con $K_i >= 3$. Es importante que este numero sea polinomico con relacion al numero de nodos, ya que da una cota polinomica del tiempo que se requiere para realizar una busqueda en la vecindad.

Esto se debe a que elegido un arco ,no puedo elegir ninguno de los adyacentes a el (ni el mismo arco) pues el resultado de remover esos dos arcos y agregar otros dos (resultarian ser los mismos) daria el mismo grafo . La division por dos es porque a los efectos del problema resulta lo mismo elegir el arco (u,y) que el arco (y,u). Ademas se pide $K_i >=3$ porque sino el numero $[K_i * (K_i - 3)] / 2$ seria negativo y no tendria sentido. Esto es porque remover dos arcos en un ciclo de dos o tres arcos obligaria a elegir dos arcos adyacentes y eso no puede pasar por lo dicho anteriormente.

b-Llamemos 1-part a la vecindad que se obtiene al aplicar una funcion N_{1p} tal que : $N_{1p}(f)=\{ g / g$ pertenece a F y g puede obtenerse de f intercambiando 2 nodos del grafo de manera tal que cada nodo pertenezca a un subconjunto de la particion distinto$\}$

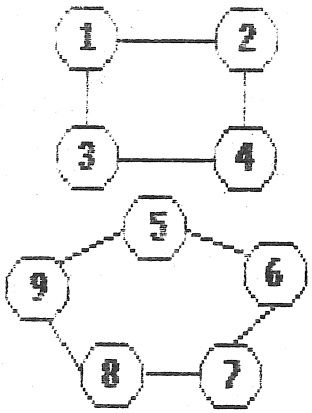
De igual manera se puede definir una vecindad m-part, obtenida a partir de una solucion factible aplicando la funcion N_{mp} tal que: $N_{mp}(f)=\{ g / g$ pertenece a F y g puede obtenerse de f intercambiando 2*m nodos del grafo de manera tal que m pertenezcan a un subconjunto de la particion, y los otros m nodos a otra distinta$\}$.
Para ello se pide que los dos subconjuntos tengan cardinal estrictamente mayor que m.

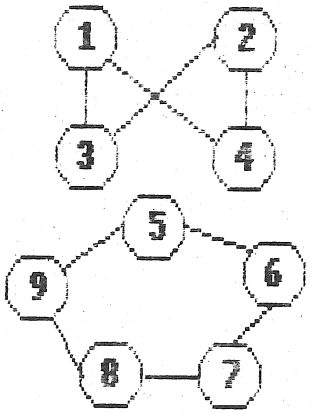
El cardinal de la vecindad 1-part esta dado por:

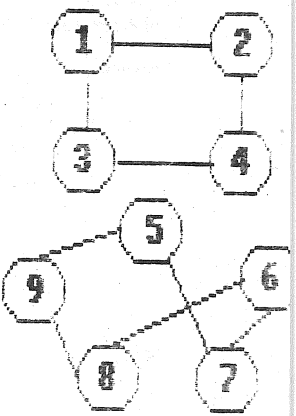
$$\#N_{1p}(f) = \frac{\text{sumatoria}_{i=1}^{N} [K_i * (K - K_i)]}{2}$$

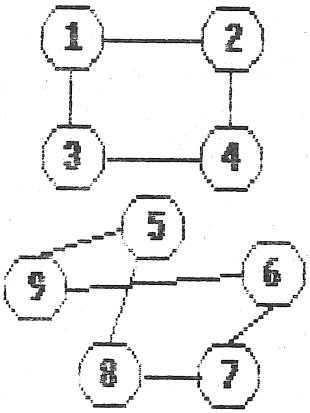
Este numero se debe a que cada ciudad del grafo puede ser intercambiada con cualquiera que no este en su mismo subconjunto de la particion. Se divide por 2 para no contar arcos repetidos.

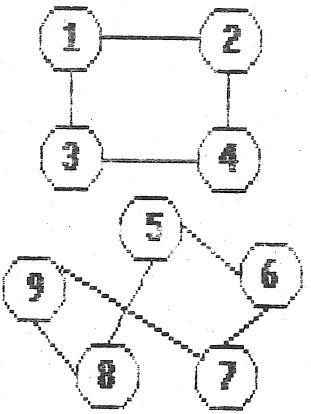
solucion inicial

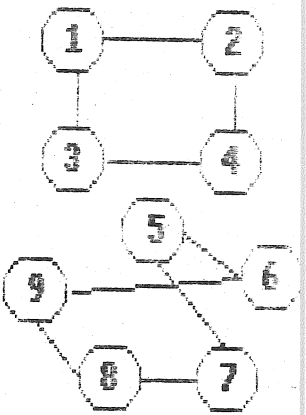












Algunos de los vecinos N2 de la solucion inicial

Para formalizar el concepto de vecindad 1-part se introducen los siguientes conceptos:

DEFINICION 11:

Dada una particion uniforme (A,B) y elementos a en A y b en B, la operacion definida por:

A'=(A-{a}) U {b}
B'=(B-{b}) U {a} se llama swap.

Una particion uniforme (A,B) es tal que #A = #B
Consideremos el efecto que tiene el swap sobre el costo de la particion A,B.

DEFINICION 12:

Llamaremos costo externo asociado a un elemento a perteneciente a A.

$E(a) = \sum_{j \text{ en } B} d_{aj}$ (d_{xy} es el costo del arco (x,y))

Llamaremos costo interno asociado a un elemento a

$I(a) = \sum_{j \text{ en } A} d_{aj}$

D(a) sera la suma E(a) + I(a), el costo de un nodo.

LEMA 1:

Dada una solucion factible f, el swap de a y b resulta en una reduccion de costo de:

$g(a,b) = c(f) - D(a) - D(b) + d_{av_3} + d_{v_4a} + d_{bv_1} + d_{v_2b}$, donde (a,v_3), (v_4,a), (b,v_1) y (v_2,b) son los nuevos arcos incorporados.

DEFINICION 13:

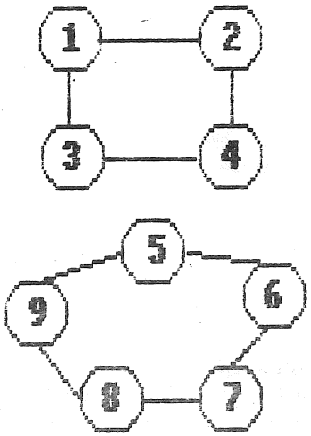
La VECINDAD SWAP (Ns) para un problema de particiones uniformes es Ns(a,b)={todas las particiones uniformes A',B' que pueden ser obtenidas de la particion uniforme A,B por un simple swap}

Se puede encontrar un swap favorable en $O(N^2)$, examinando g(a,b) para todo par a en A, b en B. Se puede investigar la vecindad definida por el intercambio de dos elementos de A con dos elementos de B en $O(N^4)$. Esta es la idea de N_{2p} . se podria formalizar del siguiente modo:

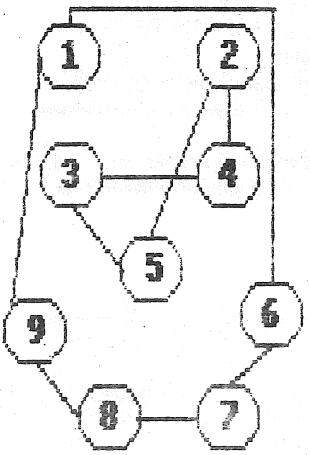
Dada una particion A1=A11 U A12, A2=A21 U A22, A3, ..., AN, un elemento de la vecindad N_{mp} seria:

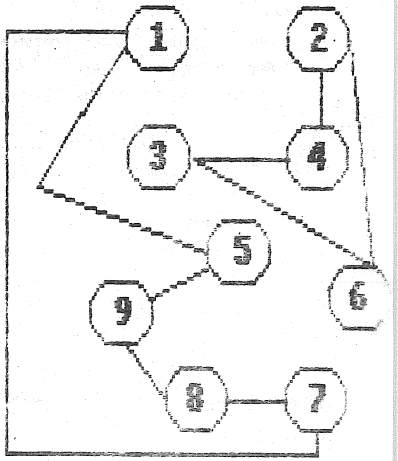
A1'=A11 U A22, A2'=A21 U A12, A3, ..., AN

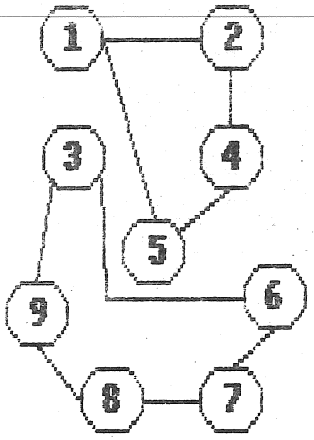
donde #A12=#A22=m

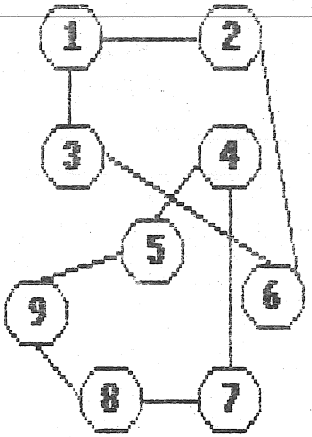


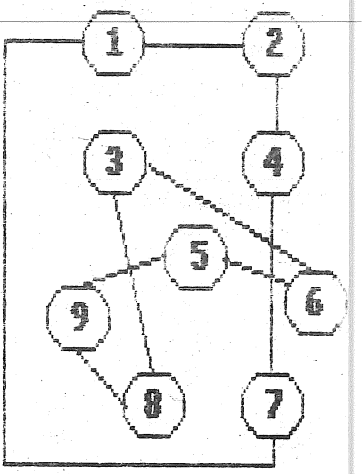
solucion inicial











Algunos de los vecinos Nip de la solucion inicial

La idea es reemplazar la busqueda de un swap favorable por una secuencia favorable de swaps, usando el costo de la instancia del problema. Asi una secuencia favorable de k swaps es obtenida :

1-calcular D(v) para todos los elementos v en V=A U B.
2-buscar el par a_1',b_1' tal que $g_1 = g(a_1',b_1')$ sea lo mas chico posible.
3-intercambiar a1' y b1' y recalcular D.
4-repetir los pasos 2 y 3, obteniendo una secuencia de pares a_2', b_2'; ...; a_k', b_k'.
Una vez que un par fue elegido no se puede elegir nuevamente.

De esta forma se obtiene una secuencia $g_1, ..., g_k$ que corresponde a los pares elegidos y tal que el resultado de intercambiar el conjunto $X=\{a_1', ...,a_k'\}$ con $Y=\{b_1', ...b_k'\}$ es

$$G(k)=\sum_{i=1}^{k} g_i$$

Cuando k = #A = #B se estan intercambiando todos los nodos de A con los de B por lo tanto G(k)=0. La idea es buscar k tal que G(k) sea minimo

En el GTSP de todos los subconjuntos de la particion solo se trabaja con dos y estos pueden tener cardinales diferentes f_1 y f_2; por lo tanto se pide $k<=f_1$, $k<=f_2$.

c-A partir de la idea sugerida en b se propone una vecindad con la funcion definida por el siguiente algoritmo:

1-Sea A la particion dada por la solucion factible. Se pide que A tenga al menos dos miembros (por lo menos dos viajantes).

2-Mientras haya mas de un miembro en A hacer

2.1-Elegir dos miembros A1 y A2 de A y aplicar N_2.
2.2- A=A-(A1 U A2) ,volver a 2.

A lo sumo quedo un miembro de A igual que antes para cada elemento de la vecindad (ya que N puede ser impar)

e-La ultima posibilidad seria combinar todas estas ideas. Pero aqui tambien se presentan varias maneras posibles de hacerlo.
Una de ellas seria dada una solucion factible hallar la vecindad con la funcion definida en c,y para cada elemento de la vecindad aplicar la funcion N_2
Otra forma seria hallar la vecindad 1-part mediante la funcion N_{1p} y para cada elemento de la vecindad aplicar N_2 . Tambien se podria hallar primero 2-change y luego 1-part para cada elemento de 2-change. Este metodo podria generalizarse:

$N_2(N_{mp}(f))$ para R-change de m-part
$N_{mp}(N_R(f))$ para m-part de R-change

(admitiendo el abuso de notacion, pues N_2 y N_{mp} estan definidas de F en partes(F))

Es claro que la mejor opcion es una estrategia combinada. Este concepto intuitivo se justifica de la siguiente manera

TEOREMA 8:

$$\#N_2(N_{1p}(f)) = \#N_{1p}(N_2(f)) = \#N_{1p}(f) * \#N_2(f)$$

Sea f una solucion inicial, y sean g1= $\#N_{1p}(f)$, G1= $N_{1p}(f)$, g2= $\#N_2(f)$, G2=$N_2(f)$.
$(N_2\ N_{1p})(f)=(N_2(N_{1p}(f)))$; esto es aplicar la funcion N_{1p} a la solucion inicial f, y luego aplicar N2 a cada elemento del conjunto G1.

$$\#(N_2(N_{1p}(f)) = \text{suma}_{g \text{ en } N_{1p}(f)}\ \#N_2(g)$$

Pero como $\#N_{1p}(f)$= g1 para cualquier f, entonces $\#(N_2(N_{1p}(f)))$= g1*g2

$(N_{1p}\ N_2)(f)=(N_{1p}(N_2(f)))$; esto es aplicar la funcion N_2 a la solucion inicial f, y luego aplicar N_{1p} a cada elemento del conjunto G2.

$$\#(N_{1p}(N_2(f))) = \text{suma}_{g \text{ en } N_2(f)}\ \#N_{1p}(g)$$

Nuevamente como $\#N_2(f)$=g2 para cualquier f, entonces $\#(N_{1p}(N_2(f)))$=g2*g1.

TEOREMA 9:

R-change esta incluida en R-change de m-part (dualmente m-part esta incluida en m-part de R-change) y m-part esta incluida en R-change de m-part (dualmente R-change esta incluida en m-part de R-change).

La demostracion es trivial. Dada la funcion N_R y N_{mp} se quiere ver que $N_R(f)$ esta incluida en $N_R(N_{mp}(f))$. f pertenece a $N_R(f)$ y a $N_{mp}(f)$. Por otro lado, $N_R(N_{mp}(f))$ es un abuso de notacion que representa en realidad

$$N_R(N_{mp}(f)) = U\ \{\ N_R(g)\ /\ g \text{ pertenece a } N_{mp}(f)\ \}$$

En particular,como f pertenece a $N_{mp}(f)$, $N_R(f)$ esta incluido en la gran union luego $N_R(f)$ esta incluido en $N_R(N_{mp}(f))$.

Como h pertenece a $N_R(h)$., para cada g perteneciente a $N_{mp}(f)$, g pertenece a $N_R(g)$. Luego para todo g en $N_{mp}(f)$, g esta en la union definida mas arriba. Luego, $N_{mp}(f)$ esta incluido en $N_R(N_{mp}(f))$.

Dualmente se demuestra la segunda afirmacion.

Quedaria ahora por definir que estrategia combinada de las posibles elegimos.

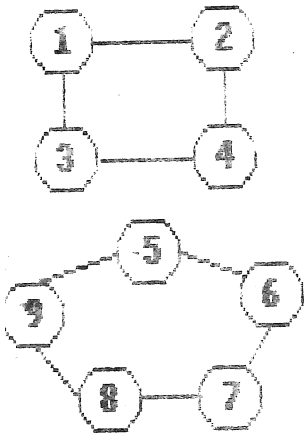
LEMA 2:

En la vecindad 2-change de una solucion factible f, dadas los arcos (s_1,s_2) y (s_3,s_4) de un ciclo C de la solucion f, queda determinado un unico vecino de f sacando esos arcos; y los arcos que los substituyen son (s_2,s_4) y (s_1,s_3).

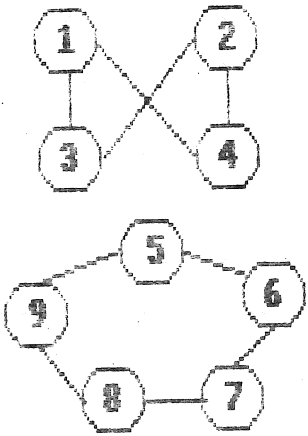
Este lema se observa graficamente. Deben verificarse por separado los casos (a) los arcos son adyacentes y (b) los arcos no son adyacentes.

TEOREMA 10:

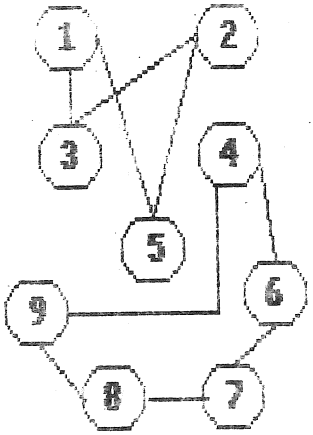
$$N_2(N_{1p}) = N_{1p}(N_2)$$



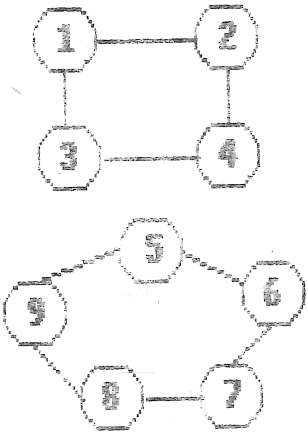
solucion inicial



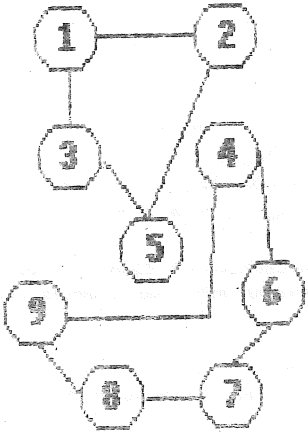
vecino N2



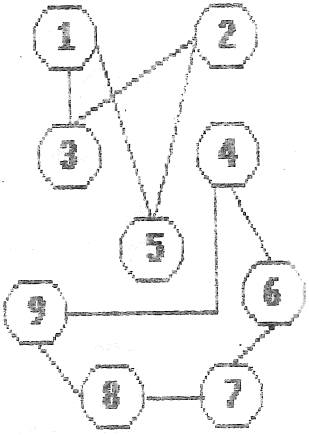
vecino N1p.N2



solucion inicial



vecino N1p



vecino N2.N1p

Demostracion:

$$N_{1p}(N_2(f)) = \bigcup_{g \text{ en } N_2(f)} N_{1p}(g)$$

Dado r en $N_{1p}(N_2(f))$ existe h en $N_2(f)$ tal que r pertenece a $N_{1p}(h)$.
Por el lema anterior, como h pertenece a N2(f), h esta univocamente determinado por los arcos elegidos.
Sean (s_1 , s_2) y (s_3 , s_4) los arcos elegidos. Por el lema anterior, los cambios fueron:

(s_1 , s_2) --> (s_1 , s_3)
(s_3 , s_4) --> (s_2 , s_4)

Si $\{(s_1 , s_2) , (s_3 , s_4)\} = \{(s_1 , s_3) , (s_2 , s_4)\}$ (en este caso los arcos serian adyacentes) se tiene h=f, luego $N_{1p}(h) = N_{1p}(f)$ por un teorema anterior esta incluido en $N_2(N_{1p}(f))$ luego $N_{1p}(N_2(f))$ esta incluido en $N_2(N_{1p}(f))$.
Es sencillo ver que si r esta en $N_{1p}(h)$, r esta univocamente determinado por los vertices v_1, v_2 elegidos para el cambio.
Analisis por casos:

a) $v_1 = v_2$, luego r = h, luego r esta en $N_2(f)$ por el teorema anterior esta en $N_2(N_{1p}(f))$

b) $v_1 <> v_2$ ==> v_1 esta en P1 y v_2 esta en P2 con P1<>P2 elementos de la particion inducida por la solucion factible f.
Hay dos posibilidades:
b1) sin perdida de generalidad, $v_1 = s_1$.
b2) $v_i <> s_j$ para j = 1..4; i = 1,2.
Por el lema anterior, toda otra posibilidad es equivaleme (s.p.g.) a alguno de estos dos.

En el caso b2) la aplicacion de las funciones es totalmente conmutativa, es decir, si se llega de f a h eligiendo (s_1,s_2) y (s_3,s_4) y de h a r eligiendo v_1, v_2, se puede llegar de f a h' eligiendo primero v_1 y v_2 y luego de h' a r eligiendo (s_1,s_2) y (s_3,s_4). La verificacion de que ambas transformaciones son validas (llevan de una solucion factible a otra solucion factible) es muy sencilla.

En el caso b1) tenemos que:
$v_1 = s_1$, luego como P1 <> P2, $v_2 <> s_i$; i = 1..4
de f se llega a h por una transformacion asociada a N_2 y de h a r por una asociada a N_{1p}.

f --> h --> r

Esta transformacion, con la misma notacion que antes, es:

N_1 $N_{1'}$

$(v_1,s_2) \dashrightarrow (v_1,s_3) \dashrightarrow (v_2,s_3)$
$(s_3,s_4) \dashrightarrow (s_2,s_4) \dashrightarrow (s_2,s_4)$
$(u_1,v_1) \dashrightarrow (u_1,v_2)$
$(u_2,v_1) \dashrightarrow (u_2,v_1)$
$(v_2,u_3) \dashrightarrow (v_1,u_3)$
$P1 \dashrightarrow P1 - \{v_1\} + \{v_2\}$
$P2 \dashrightarrow P2 - \{v_2\} + \{v_1\}$

Donde u_1 es el vertice adyacente a v_1 tal que $u_1 <> s_2$ y u_2, u_3 son los vertices adyacentes a v_2.

Definidas las siguientes transformaciones:

$N_{2'}$ N_2

$(v_1,s_2) \dashrightarrow (v_2,s_2) \dashrightarrow (v_2,s_3)$
$(s_3,s_4) \dashrightarrow (s_3,s_4) \dashrightarrow (s_2,s_4)$
$P1 \dashrightarrow P1 - \{v_1\} + \{v_2\}$
$P2 \dashrightarrow P2 - \{v_2\} + \{v_1\}$
$(u_1,v_1) \dashrightarrow (u_1,v_2)$
$(u_2,v_2) \dashrightarrow (u_2,v_1)$
$(v_2,u_3) \dashrightarrow (v_1,u_3)$

Es sencillo verificar que son validas. Por lo visto mas arriba y el lema anterior, estas transformaciones definen univocamente un vecino que cumple las transformaciones en las vecindades dadas. El vecino hallado es precisamente r.

Luego, se ha probado que $N_{1p}(N_2(f))$ esta incluido en $N_2(N_{1p}(f))$ para todo f solucion factible. Como $\#N_{1p}(N_2(f)) = \#N_2(N_{1p}(f))$ para todo f se deduce que $N_{1p}(N_2) = N_2(N_{1p})$.

Posiblemente este resultado pueda generalizarse para m y R arbitrarios aunque no sera demostrado.

CONJETURA:
$N_{mR}(N_2) = N_2(N_{mR})$ para R, m naturales.

Esto demuestra que al elegir una estrategia combinada no interesa el orden en que se aplican las funciones para encontrar vecindades.

6.4 - METODO DE BUSQUEDA

Pueden aplicarse varios metodos de busqueda. Uno de ellos seria adoptar la primer solucion que se encuentre en la vecindad que sea mejor que la que se tiene. El otro extremo seria buscar en toda la vecindad hasta encontrar la mejor solucion y adoptarla.

Se debe tener en cuenta, tambien, la idea de *REDUCCION*. Un ejemplo de la aplicacion de esta idea en el TSP es: al obtener varios optimos locales, se observa que tienen arcos en comun. Una estrategia es, entonces, fijar esos arcos y seguir buscando pero ahora con menos arcos: se redujo el espacio de busqueda y se trabaja en subconjuntos de las vecindades.

Otra idea, llamada *DENIAL*, consiste en hacer exactamente lo opuesto. Estos arcos que aparecen repetidos se prohiben, ya que se piensa que buscar siempre "por ese lado" puede estar enganando a la estrategia. Se fuerza a respetar la heuristica pero para un conjunto de soluciones alejado al que se estuvo revisando hasta el momento. Esto sirve sobre todo si la solucion obtenida se aleja mucho del optimo.

En el momento de implementar el algoritmo, tambien debe tenerse en cuenta el orden en el que se va a realizar la busqueda. Este orden puede tener o no importancia, dependiendo del problema.

7 - CONCLUSIONES

En este trabajo se presenta una generalizacion de un problema muy conocido. A traves del GTSP se intenta ejemplificar una metodologia de trabajo para enfrentar y resolver problemas muy complejos, ya sea usando soluciones a problemas similares o tecnicas modernas como local search.

Al haber contado con una formalizacion del concepto de vecindad, se pudieron obtener valiosos resultados teoricos.

Las conclusiones halladas acerca de la complejidad del GTSP sirvieron de impulso para buscar distintas soluciones aproximadas de tiempo polinomico.

Es importante tener en cuenta que el GTSP es una posible generalizacion al problema del viajante, otras generalizaciones podrian aportar distintas ideas a los temas planteados.

En el trabajo se presentaron varios algoritmos. Para poder compararlos y obtener conclusiones mas precisas seria interesante implementarlos y hacer estadisticas sobre ellos.

8 - BIBLIOGRAFIA

[AHU 83] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, "Data structures and algorithms", editorial Addison-Wesley, 1983

[Bo 58] F. Bock, "An algorithm for solving TS and related network optimization problems", 14 National Meeting de la ORS de America, St Louis, Missouri, 1958

[Cr 58] G. A. Croes, "A method for solving TSP", OR, 6, numero 6, paginas 791-812, 1958

[Eve 79] Shimon Even, "Graph algoritms", Technion Institute, editorial Computer Science Press, 1979

[Fl 59] Merrill M. Flood, "Operations research for management", editores Joseph Mc Closkey y John M. Coppinger, capitulo 19, "The traveling salesman problem", 1959

[Fri 83] A. M. Frieze, "An extension of Christofides heuristic to the K-person travelling salesman problem", University of London, Discrete Applied Mathematics, numero 6, paginas 79-83, 1983

[Ha 78] John P. Hayes, "Computer architecture and organization", University of Southern, California, editorial McGraw-Hill, 1978

[Har 69] Frank Harary, "Graph theory", University of Michigan, editorial Adison-Wesley Publishing Company, 1969

[Hk 62] M. Held, "A dinamic programming approach to sequencing problems", J. SIAM, 10, numero 1, paginas 196-210, 1962

[LePa 78] Harry R. Lewis, Christos H. Papadimitrou, "La eficiencia de los algoritmos", revista Investigacion y Ciencia, numero 18, paginas 78 a 91, 1978

[LePa 81] Harry R. Lewis, Christos H. Papadimitriou, "Elements of the theory of computation", editorial Prentice-Hall, 1981

[Li 65] S. Lin, "Computer solution of the TSP", BSTJ, 44, numero 10, paginas 2245-2269, 1965

[Nil 71] Nils J. Nilsson, "Problem-solving methods in artificial intelligence", editorial McGraw-Hill, 1971

[NiWi 75] Albert Nijenhuis, Herbert S. Wilf, "Combinatorial algorithms", editorial Academic Press, 1975

[Pap 82] Christos H. Papadimitriou, Kenneth Steiglitz, "Combinatorial optimization, algorithms and complexity", editorial Prentice-Hall, 1982

[Pap 84] Christos H. Papadimitrou, "On the complexity of unique solutions", Stanford University and National Technical University of Athens, Journal of the Association for Computing Machinery vol. 31, numero 2, paginas 392 a 400, 1984

[RND 77] Edward M. Reingold, Jurg Nievergelt, Narsingh Deo, "Combinatorial algorithms, theory and practice", editorial Prentice-Hall, 1977

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

UN ALGORITMO PARA ENRUTAMIENTO NO LOCAL

Autor : Ing. Diego Andrade Stacey (*)

Pontificia Universidad Católica del Ecuador
Centro de Computación e Informática
Apartado 2184
Quito - Ecuador

RESUMEN

Se presenta en este trabajo un método para determinar la ruta más corta entre dos puntos del plano en presencia de obstáculos rectilíneos. La etapa inicial constituye la generación de una gráfica no dirigida - la grafica de figuras -, a partir de los obstáculos; los nodos de la gráfica son figuras geométricas (cuadriláteros y triángulos) y, existe un lado de la gráfica siempre y cuando las figuras correspondientes son adyacentes y no existe un obstáculo entre ellas. En una etapa siguiente se desarrolla el algoritmo de enrutamiento el cual consiste en recorrer la gráfica de figuras partiendo del nodo en el cual se ubica el punto inicial; el recorrido de la gráfica se lleva adelante con el concepto de propagación de ranuras mediante el cual se pasa del lado de una figura hacia otro lado que sirve de comunicación con una figura adyacente. Si m es el número de obstáculos el número de figuras geométricas n es de orden O (m), mientras que los algoritmos tanto de construcción de la gráfica como del proceso de enrutamiento son en esperanza de O (nlog(n)) pero pueden llegar a ser de O (n^2).

(*) Director del Centro de Computación e Informática de la Pontificia Universidad Católica del Ecuador.

Profesor de la Universidad Central del Ecuador.

1.INTRODUCCION.

El problema de determinar la ruta o trayectoria de longitud mínima entre dos puntos de un plano referenciado por un sistema de ejes coordenados (x,y) ha sido tratado extensamente en diversas áreas tales como la investigacion de operaciones, diseño de circuitos integrados [1],[2],[4], movimiento de robots [3],[5]; entre otras aplicaciones. Un aspecto especial es aquel de obtener la ruta mas corta entre dos puntos en presencia de obstáculos los cuales pueden corresponder sea a poligonos cerrados o a segmentos rectilíneos.

La solución tradicional que se ha dado a la obtención de rutas de longitud mínima esta enmarcada en la teoria de gráticas, con algoritmos como el de Dijsktra.

En el ámbito de la determinacion de la ruta más corta en presencia de obstáculos en los métodos de enrutamiento locales los vecinos de un punto son algunos de los puntos de una figura geométrica que rodea al punto.

Algunos de estos métodos locales y los consiguientes algoritmos que de ellos se desprenden idealizan al plano como una malla o retícula [2], dividiendo al plano en intervalos de longitud unitaria en cada dirección. Los puntos de intersección de las líneas trazadas por estos intervalos corresponden a los vértices de una gráfica. Véase fig. 1.1

En este tipo de modelo la ruta se genera pasando de punto a punto de la retícula y tomando en cuenta como entorno de un punto a los puntos adyacentes a ese vértice, exclusivamente.

En contraposición otros métodos consideran un enfoque global del plano para la obtención de la ruta, es decir los vecinos de un punto corresponden a toda una área.

Se presenta en este artículo un algoritmo no local para obtener la ruta más corta entre dos puntos en presencia de obstáculos rectilíneos, el mismo que esta basado en la partición del plano de estudio en un conjunto de figuras geométricas (cuadriláteros y triángulos). Estas figuras se forman conforme se introducen los obstáculos, de uno en uno. Véase fig. 1.2

Para obtener un grafo a partir de la división del plano en figuras, cada figura es un nodo de una gráfica no dirigida. Entre dos nodos hay un lado siempre y cuando las figuras correspondientes sean adyacentes y no exista un obstáculo entre ellas.

El algoritmo de enrutamiento que se describe en este trabajo consiste en efectuar el recorrido de la gráfica de figuras desde la que contiene al punto inicial hasta aquella en la que se ubica el punto final. El recorrido de la gráfica se lleva a cabo con el concepto de propagación de ranuras, es decir, pasar de un lado de una figura hacia otro lado que sirve de comunicación con una

figura adyacente.

Finalmente, el método descrito en este trabajo permite obtener una ruta de longitud mínima entre dos puntos tanto con la norma de medida l_1 -ruta de Manhattan- como con la norma de medida l_∞.

Si m es el número de segmentos rectilíneos (obstáculos), el número de figuras geométricas (n) en las que se descompone el plano es generalmente del orden de n = 3m + 1 y en el peor de los casos n = 9m + 1. Los algoritmos de construcción de la gráfica de figuras y del proceso de enrutamiento son en esperanza de O (nlog(n)) pero pueden llegar a ser de O (n^2).

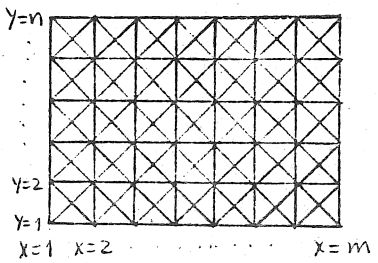


Fig. 1.1 Modelo de enrutamiento local

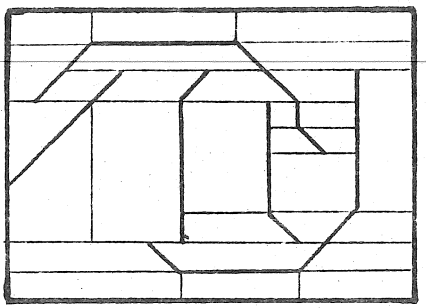


Fig. 1.2 Modelo de enrutamiento no local

2.LA GRAFICA DE FIGURAS.

Se introduce aquí uno de los conceptos importantes de este trabajo y que se lo ha denominado gráfica de figuras, es decir el conjunto de figuras geométricas y su concepción como una gráfica en que se divide el plano original como consecuencia de la introducción de los obstáculos en el plano.

2.1 OBSTACULOS Y SU REPRESENTACION.

El espacio bajo consideración es el plano referenciado con un sistema de ejes coordenados X,Y.

Se define a los obstáculos como los segmentos de recta horizontal, vertical u oblicuo a 45 o 135 grados, definidos por las coordenadas de sus puntos extremos.

Los obstáculos pueden estar aislados o coincidiendo en uno de sus extremos en cuyo caso se tendría un polígono abierto.

2.2 TIPOS DE FIGURAS GEOMETRICAS Y SU REPRESENTACION.

Los tipos de figuras que se considerarán en la gráfica de figuras son cuadriláteros y triángulos.

Los cuadriláteros corresponden a las siguientes figuras :

- Rectángulos
- Trapecios rectángulos
- Trapecios isósceles
- Romboides

Todos los triángulos son rectángulos.

La representación general de estas figuras geométricas será en base a las coordenadas de sus vértices.

En la figura 2.1 se indican los diferentes tipos de figuras geométricas consideradas.

No siempre las figuras geométricas así definidas se encntrarán en la posición mostrada en la Fig. 2.1, sino que pueden aparecer rotadas a 90, 180 o 270 grados.

Finalmente, al menos un lado de la figura debe coincidir con todo o parte de un obstáculo.

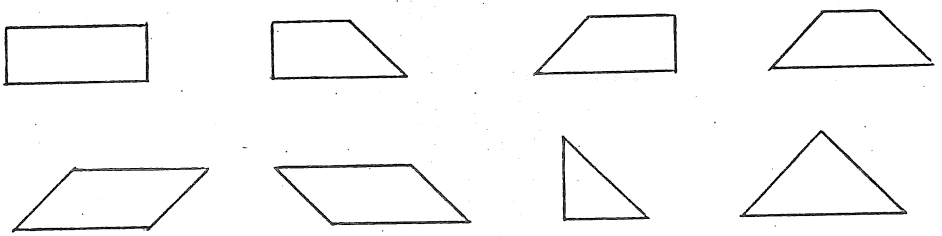


Fig. 2.1 Tipos de figuras geométricas.

2.3 CONSTRUCCION DE LA GRAFICA DE FIGURAS.

2.3.1 GENERACION DE LAS FIGURAS DE LA GRAFICA.

La gráfica de figuras se construye siguiendo un procedimiento repetitivo y uniforme conforme se van introduciendo uno a uno los obstáculos. El orden en que se introducen los obstáculos es irrelevante aunque un determinado orden puede ser más favorable que otro. El procedimeiento que se sigue con cada obstáculo esta encaminado a dividir en otras figuras geométricas a la figura en la cual se encuentra el punto inicial del obstáculo. Los aspectos importantes con los que se efectúa esta división son los siguientes :

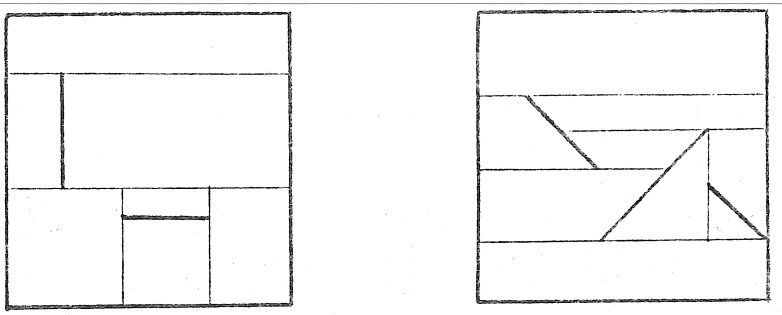
a) se trazan lineas perpendiculares por los extremos de cada uno de los segmentos rectilineos horizontales y verticales correspondientes a obstáculos de esos tipos.

b) se trazan lineas horizontales o verticales por los extremos de los segmentos rectilineos correspondientes a los obstáculos oblicuos, sean a 45 o 135 grados. No se trazan lineas perpendiculares por los extremos de los obstáculos oblicuos para asegurar que las particiones de la figura sigan siendo a lo más cuadriláteros.

c) se extienden estas lineas trazadas por los extremos de los obstáculos en los dos sentidos hasta encontrar una de las siguientes limitaciones :

- los límites del espacio de estudio.
- a otro obstáculo.
- otras lineas similares construidas previamente con relación a otros obstáculos.

En la figura 2.2 se ilustra una manera de efectuar la división de una figura.



(a) por un obstáculo vertical u horizontal. (b) por un obstáculo oblicuo

Fig. 2.2 Proceso de división de una figura.

d) se efectúan uniones de dos o mas figuras en una sola siempre que la figura resultante continúe perteneciendo al conjunto de figuras definido en la sección anterior. Claramente, estas uniones pueden efectuarse a través de los lados ficticios

de las figuras generados mediante lo descrito en a) y b).

e) se determinan las figuras adyacentes o vecinas con las que una figura tiene comunicación, a través de aquellos lados que no son obstáculos. Se liga a la figura considerada el conjunto de sus figuras vecinas. Las partes de la figura original, así conformadas finalmente, substituyen a la figura que se ha dividido.

En consecuencia, un elemento o nodo de la gráfica de figuras estará constituido por la figura en sí y por una lista de figuras vecinas.

f) se actualizan las listas de vecinos de las figuras en las cuales aparezca la figura que se ha dividido. En esas listas una o varias partes de la figura dividida van a substituir a esa figura original.

2.3.2 OBSERVACIONES RELATIVAS A LA DIVISION DE FIGURAS.

En el proceso descrito en la sección anterior, relativo a la división de figuras pueden darse las siguientes situaciones:

a) el número de partes en las que un obstáculo divide a la figura en la que se encuentra el punto inicial depende sobretodo de si el obstáculo esta contenido o no íntegramente en esa figura, influye también el tipo de figura que se va a dividir.

b) se efectúan los pasos descritos para la división solamente con figuras en la posición normal; si la figura se encuentra en una posición de rotación se modifica previamente, tanto la figura como el obstáculo con un sencillo proceso de rotación de ejes, a fin de pasar de la posición de rotación a la posición normal. Una vez completada la división de la figura se aplica otro proceso de rotación para volver a las partes a su posición original de rotación. De esta manera se reducen los casos de división.

c) si un obstáculo coincide con un lado de la figura a la que va a dividir, es decir es común a dos figuras deben dividirse estas dos figuras independientemente.

d) Si el punto final del obstáculo se encuentra fuera de la figura que actualmente se esta dividiendo, debe aplicarse el proceso de división a la o las figuras adyacentes a las que afecta el mismo obstáculo hasta que se considere una figura para la cual el punto final del obstáculo sea interior.

En la figura 2.3 se muestra un plano y su división en regiones geométricas.

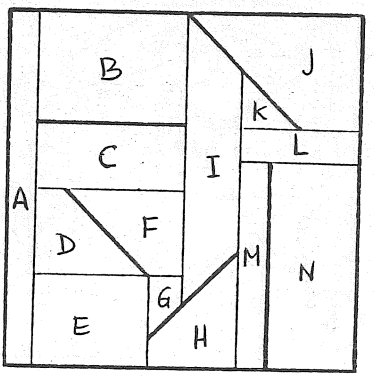


Fig. 2.3 Conformación de las regiones geométricas.

2.3.3 NUMERO DE FIGURAS GEOMETRICAS QUE SE GENERAN.

Respecto al número total de figuras que se generan una vez introducidos todos los obstáculos este depende del orden en que fueron introducidos los obstáculos y de qué tipo de obstáculo esta incidiendo sobre un determinado tipo de figura, en la partición. Así, si m es el número de obstáculos y estos son horizontales o verticales exclusivamente el número de figuras se puede probar que es de $3m + 1$. En cambio, al considerar también obstáculos oblicuos el número de figuras geométricas puede subir hasta $9m + 1$.

2.4. EL MODELO DE GRAFICA A PARTIR DE LAS FIGURAS GEOMETRICAS.

Una vez que se ha formado el conjunto de figuras geométricas mediante el procedimiento descrito anteriormente puede efectuarse la correspondencia del plano asi particionado en figuras con una gráfica no dirigida en la siguiente manera:

- cada figura geométrica corresponde a un vértice de la gráfica.

- un vértice tiene relación con otro si en las figuras correspondientes una figura es adyacente con otra a través de un lado ficticio, es decir los lados de figuras que representan obstáculos no expresan relación o comunicación con una figura adyacente. En base a esta consideración se construyen las aristas o lados de la gráfica.

En este trabajo la gráfica de figuras esta constituida como una lista ligada donde cada nodo es una figura geométrica del plano, la cual, como se recordará tiene asociada una lista de figuras vecinas.

En la figura 2.4 se indica la gráfica correspondiente a la división en figuras de la figura 2.3.

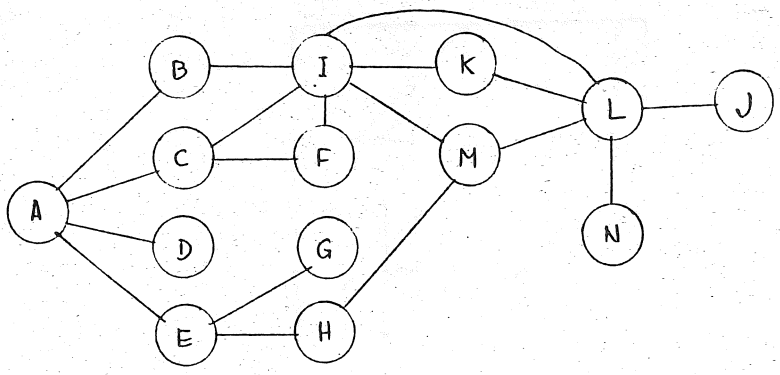


Fig. 2.4 Gráfica representativa de las figuras geométricas.

2.5 ALGORITMOS UTILIZADOS PARA LA CONSTRUCCION DE LA GRAFICA DE FIGURAS.

Tal como se ha descrito en la sección 2.3 el procedimiento general de formación de la gráfica de figuras consta de tres etapas bien definidas:

a) la división de una figura en sus partes,

b) la obtención de la lista de figuras vecinas para cada parte y,

c) la actualización de la gráfica en general como consecuencia de la inclusión de las partes y la supresión de la figura que se dividió.

A continuación se presenta el algoritmo general de generación de la gráfica de figuras.

2.5.1 ALGORITMO GENERAL PARA LA CONSTRUCCION DE LA GRAFICA.

Mientras se ingresan obstáculos se efectúa el siguiente procedimiento:

1. Se ingresan los datos de los puntos extremos del obstáculo.

2. Se obtiene el apuntador de la figura en la que empieza el obstáculo.
   Si el obstáculo es común a dos figuras se obtiene el apuntador de la segunda figura en la cual empieza también el obstáculo.
   Para cada figura determinada por su apuntador se aplican los siguientes pasos, del 3 al 8:

3. Se efectúa la división de la figura obteniéndose las partes correspondientes a la figura original.

4. Se obtiene la lista de figuras vecinas para cada una de las partes

5. Se insertan las partes en la estructura general de la gráfica en el sitio determinado por el apuntador a la figura original.

6. Se actualizan las listas de figuras vecinas para aquellas figuras de la gráfica que tienen como vecina a la figura original.

7. Se elimina de la gráfica la figura que se dividió.

8. Si el obstáculo no continúa en otra figura se termina con el procedimiento.
   Si el obstáculo continúa en otra figura se busca en la lista de figuras vecinas cual es la siguiente figura a dividirse, se obtiene su apuntador y se repiten los pasos del 3 al 8 inclusive.

2.6 COMPLEJIDAD DE LOS ALGORITMOS.

Si m es el número de obstáculos presentes en el plano, el número de figuras geométricas obtenidas es de O (m), pues corresponde a lo más 3m + 1 en el caso usual o en el más desfavorable 9m + 1.

La gráfica de figuras puede almacenarse utilizando diversas estructuras de datos. Posiblemente las estructuras más convenientes que se pueden utilizar son las de árbol y entre estos los árboles R [6] o los árboles cuaternarios (quad trees) [7].

Sea n el número de figuras geométricas de la gráfica de figuras en un instante dado de su generación, al introducir un nuevo obstáculo se tienen los siguientes órdenes de complejidad en los diferentes pasos que se siguen para la obtención de un nuevo nodo de la gráfica:

En la fase de búsqueda de la figura en la cual esta el punto inicial el algoritmo es del orden de O (log(n)).

En la etapa de la obtención de la lista de vecinos puede presentarse la situación más desfavorable que un mismo obstáculo tenga que recorrer hasta n/2 nodos, con lo cual el orden del algoritmo sería de O (n).

En consecuencia, el orden del algoritmo de generación de la gráfica, para n obstáculos es de:

$$O(n^2).$$

El orden de $O(n^2)$ es el precio que se paga al introducir los obstáculos de manera totalmente aleatoria y que ese orden aleatorio provoque la complejidad indicada. Sin embargo, en la mayoría de casos reales se tiene la convicción que el orden del algoritmo es en esperanza de O (n).

Cabe señalar finalmente que existen algoritmos con orden $\Theta(n \log(n))$ los mismos que generan la gratica en un solo paso y no de manera incremental como se describe en este trabajo.

3.METODO DE PROPAGACION DE RANURAS PARA OBTENER LA RUTA MINIMA.

Se presenta en este punto el método para determinar la ruta de longitud mínima entre dos puntos en presencia de obstáculos basado en el concepto de **propagación de ranuras**.

3.1 CONCEPTOS PREVIOS.

En el método de enrutamiento que se propone en el presente trabajo frecuentemente va a ser necesario obtener la distancia mínima con relación a un punto que se denominará origen, a lo largo de todo un lado de una figura.

Este problema se va a incluir dentro de uno más general como es el de obtener las distancias entre el punto origen y todos los puntos de una recta. Este planteamiento se va a estudiar en dos instancias del mismo:

a) cuando entre el origen y la recta no existen obstáculos y

b) cuando entre el origen y la recta existen obstáculos.

El problema se va a resolver tanto para la norma l_1 como para la norma l_∞.

Con fines explicativos se indica en los puntos 3.1.1 y 3.1.2 la obtención de funciones que permitan calcular la distancia desde el origen hasta cualquier punto de la recta en las dos instancias antes indicadas.

3.1.1 FUNCIONES DE DISTANCIA DESDE UN PUNTO ORIGEN HASTA UNA RECTA CUANDO NO EXISTEN OBSTACULOS DE POR MEDIO.

Sea un punto origen P dado por sus coordenadas (X_P, Y_P) y una recta definida por sus extremos A y B, de coordenadas (X_a, Y_a) y (X_b, Y_b), respectivamente. Sin pérdida de generalidad se supone a la recta horizontal, es decir $Y_a = Y_b = Y_Y$ e igualmente se supone a P situado fuera de la recta, por ejemplo por debajo de ella.

Se trata de determinar la función de distancia de los puntos de la recta AB en relación con el punto origen P.

La función de distancia variará conforme se recorra la recta a lo largo de su eje desde el un extremo hasta el otro y dependerá en su expresión algebraica de la posición de P respecto a los extremos de la recta.

La función de distancia, en el intervalo $[X_a, X_b]$, corresponderá a uno de los siguientes tipos:

- Función tipo f1, cuando en todo el intervalo es una recta de pendiente negativa.
- Función tipo f2, cuando en todo el intervalo es una recta de pendiente positiva.

- Función tipo f3 cuando en todo el intervalo es una recta paralela al eje x.

- Función tipo f4 cuando en el subintervalo $[X_a, X_1]$ es del tipo f1 y en el subintervalo $[X_1, X_b]$ es del tipo f3.

- Función tipo f5 cuando en el subintervalo $[X_a, X_1]$ es del tipo f1 y en subintervalo $[X_1, X_b]$ es del tipo f2.

- Función tipo f6 cuando en el subintervalo $[X_a, X_1]$ es del tipo f3 y en el subintervalo $[X_1, X_b]$ es del tipo f2.

- Función tipo f7 cuando en el subintervalo $[X_a, X_1]$ es del tipo f1, en el subintervalo $[X_1, X_2]$ es del tipo f3 y en el subintervalo $[X_2, X_b]$ es del tipo f2.

La Fig. 3.1 ilustra estos tipos de funciones.

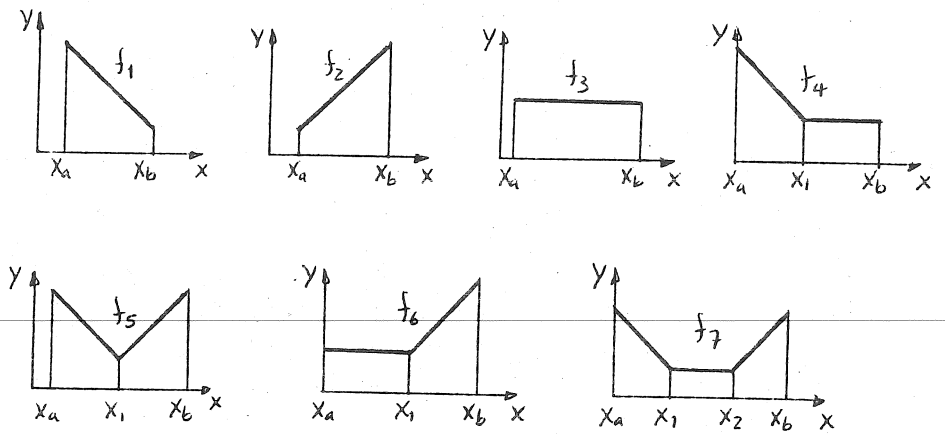


Fig. 3.1 Tipos de funciones de distancia.

Se destaca el hecho de que las funciones de distancia mostradas presentan un punto o un conjunto de puntos para los cuales el valor de la distancia es mínima.

3.1.2 FUNCIONES DE DISTANCIA DESDE UN PUNTO ORIGEN HASTA UNA RECTA CUANDO EXISTEN OBSTACULOS DE POR MEDIO.

Las funciones de distancia desde un punto origen hasta una recta, cuando entre el origen y la recta existen obstáculos se expresan en términos de las funciones de distancia obtenidas en la sección 3.1.1 con un proceso adicional de traslado del punto origen hacia los extremos del obstáculo. En otras palabras el punto origen se substituye por dos o más orígenes, cada uno ubicado en los extremos del obstáculo, desde cada uno de estos

orígenes se pueden plantear las funciones de distancia ya vistas para cuando no existen obstáculos entre el origen y la recta. Para obtener el valor de la distancia de un punto de la recta respecto al origen inicial se añadirán los valores de la distancia desde el origen inicial hasta en nuevo origen y el valor de la distancia desde el nuevo origen hasta el punto considerado en la recta.

Al haber transformado la obtención de la distancia respecto a un punto origen al cálculo referente a varios orígenes se tendrá una función diferente con su correspondiente dominio para cada origen.

La función de distancia, dividida así en varias, esta constituida de todas maneras por las funciones tipo f1 a f7 vistas anteriormente y es continua en su conjunto.

En la Fig. 3.2 se ilustra el proceso aquí descrito referente a la consideración de varios orígenes, así como las diferentes funciones de distancia respecto a cada origen.

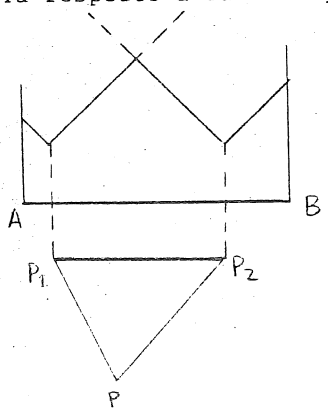


Fig. 3.2 Distancia desde un origen a una recta cuando existe un obstáculo entre el origen y la recta.

3.2 EL CONCEPTO DE RANURA.

Sea una recta representativa del lado de una figura. Se desea accesar a la recta desde uno o varios puntos orígenes y obtener sobre la recta, con relación a cada punto origen, tanto la función de distancia como los puntos para los cuales la distancia respecto al origen es mínima.

Como se puede ver en la Fig. 3.3 estas funciones de distancia se intersectan y traslapan si se consideran varios orígenes. Cada punto origen definirá sobre la recta una función de distancia con un dominio tal que los puntos de la recta pertenecientes a ese dominio estarán más cercanos al origen respectivo que a los restantes orígenes. Esta es la noción fundamental de una ranura.

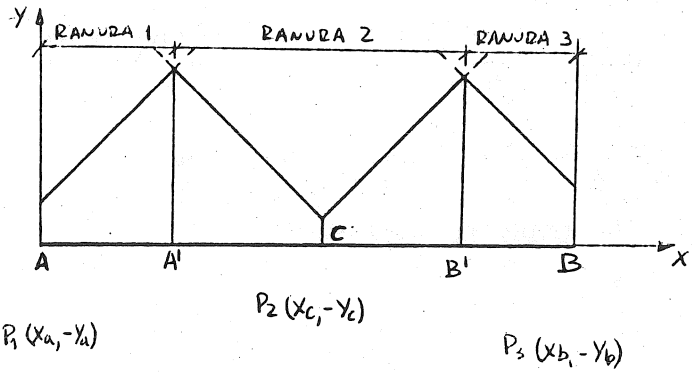


Fig 3.3 Ranuras formadas en una recta con tres orígenes.

Sin embargo, para los fines que se persiguen con este trabajo el concepto de ranura tiene asociado varios hechos y elementos [3]:

a) la definición sobre la ranura de un punto o un conjunto de puntos que son los de distancia mínima desde el punto origen de la ranura.

b) el guardar este valor de distancia mínima.

c) el recordar cual es el punto origen de la ranura.

Si bien una ranura se define por la existencia de una función de distancia con su correspondiente dominio, lo que supone la obtención de un segmento de recta; en lo sucesivo se tomarán en cuenta solamente el o los puntos de la ranura que tienen distancia mínima con relación al origen.

3.3 OBSERVACIONES RESPECTO A LAS RANURAS.

Todo lado de una figura puede tener dos caras, siendo cada cara la que dá a cada figura adyacente a través del lado comun. Sobre cada cara pueden existir ranuras.

El proceso de determinación de la ruta más corta entre dos puntos mediante el algoritmo que se describirá en el punto 3.4 puede ser visualizada como la obtención de una cadena de ranuras, una para cada cara de las figuras que atraviesa la ruta óptima.

Las ranuras se deben mantener conexas , es decir no debe quedar espacio en la recta sin que pertenezca a una ranura.

El número de componentes conexas de una ranura que se pueden generar sobre una recta es $O(m^2)$, donde m es el número de obstaculos. Es decir, se tienen $O(m^2)$ ranuras que llegan a un lado por los dos sentidos (por las dos caras).[3]

Sin embargo, si se toma en cuenta cada sentido separadamente y se tiene como condición necesaria la existencia de monotonicidad el número de componentes conexas se reduce a O (m) ranuras.

Por monotonicidad se entiende el hecho que si dos ranuras conexas se intersectan el resultado continua siendo conexo.

3.4 DESCRIPCION DEL METODO DE ENRUTAMIENTO.

3.4.1 INTRODUCCION.

El algoritmo es semejante al de Dijkstra.

Se trata de determinar la ruta de longitud mínima desde un punto inicial "s" hasta un punto final "t", tratando de no intersectar los obstáculos presentes.

Se dispone de una cola prioritaria para guardar las ranuras, ordenada ascendentemente en relación a la distancia al origen.

Existe un caso trivial: si entre los puntos "s" y "t" no existen obstáculos la longitud de ruta puede calcularse directamente como:

- Con norma l_1:

$$d = |X_t - X_s| + |Y_t - Y_s|$$

- Con norma l_∞:

$$d = \max (|X_t - X_s| , |Y_t - Y_s|)$$

En los siguientes párrafos, del 3.4.2 hasta el 3.4.6, se explican los pasos del algoritmo.

3.4.2 INICIO DEL PROCESO.

Se inicia el proceso determinando, en la gráfica de figuras a la figura que contiene al punto inicial "s", obteniéndose para todos los lados de la figura, mediante los cuales se comunica con las figuras vecinas (lados que no son obstáculos), las funciones de distancia y por consiguiente las ranuras respectivas. Se determinan además los valores de distancia mínima desde "s" hasta puntos de referencia ubicados en las ranuras obtenidas.

Se guardan en la cola prioritaria las ranuras. La cola esta ordenada ascendentemente en relación a la distancia al origen medida sobre la ranura.

3.4.3 PROPAGACION DE RANURAS.

Un vez inicializado asi el procedimiento de enrutamiento, este continua mediante una sucesión de etapas similares a las del algoritmo de Dijkstra y que efectuan la propagación de la ruta a

través de las ranuras.

La ranura a propagarse es aquella para la cual se tiene la distancia mínima en el frente de la cola, asignando además a su punto de distancia mínima como origen local para la propagación. Si la cola esta vacia se termina el algoritmo con falla.

Definida así la ranura a propagarse se efectúan la secuencia de pasos indicada en la inicialización :

a) se obtienen las funciones de distancia para cada lado de la figura a la que pertenece la ranura que se propaga, siempre que mediante estos lados exista comunicación hacia las figuras vecinas.

b) se establece el dominio de cada función, es decir se definen las nuevas ranuras.

c) se determina sobre cada ranura un punto referencial que es aquel de distancia mínima respecto al origen local.

d) se actualiza el contenido de la cola prioritaria. Si las ranuras generadas son nuevas se introducen en la cola, si son existentes se modifican. Las distancias a guardarse correspondientes a los puntos referenciales deben ser las totales desde el punto inicial "s". Es decir, si O es el origen local, "p" el punto referencial encontrado para la nueva ranura y d la distancia desde "s" hasta O, la distancia total desde "s" hasta "p" es |Op| + d , valor que corresponderá a d cuando a esta nueva ranura le toque propagarse [3].

Como se explicó en el punto 3.3, con el fin de mantener ranuras conexas, es decir, que en el proceso de obtención de ranuras no queden espacios sin analizarse y además para evitar que se produzcan ciclos a lo largo de sucesivas propagaciones, es decir que al propagar una determinada ranura se regrese a la ranura original de la cual provino, se asumen sentidos para la propagación de ranuras. Se tiene entonces que en un lado de una figura se pueden considerar dos caras las que corresponden al acceso a ese lado desde dos sentidos diferentes, para cada cara del lado podrán determinarse funciones de distancia y, de hecho, cada ranura que corresponda a cada cara se maneja independientemente. La Fig. 3.4 ilustra los sentidos de propagación y las caras de los lados de las figuras.

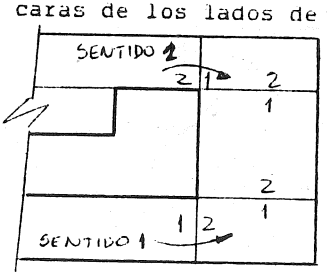


Fig. 3.4 Sentido de propagación de ranuras.

3.4.4 INTERSECCION DE RANURAS.

A lo largo de la propagación de la ruta a través de las ranuras puede presentarse el hecho de que se llegue al mismo lado de una figura y a su misma cara, por diferentes caminos, es decir desde diferentes ranuras pero mediante el mismo sentido de propagación. Esto es equivalente a tener sobre un mismo lado varias ranuras, es decir definir varias funciones de distancia en relación a diferentes puntos de origen, tal como se explicó en el punto 3.2. Debido a que cada función de distancia tiene su correspondiente dominio existirán puntos de intersección de estas funciones, esto se denomina intersección de ranuras.

La intersección se presenta en diferentes etapas de propagación de la ruta, pues en una etapa cualquiera se generará sobre el lado una sola ranura. Se supone que posteriormente en otra etapa se llegó al mismo lado, generándose otra ranura que modificará a la ranura original en el sentido que cada una de ellas cubrirá en su dominio un porcentaje del lado de la figura. Los diferentes dominios de las ranuras quedarán definidos por los puntos de intersección de las funciones de distancia correspondientes.

En la Fig. 3.5 se tienen, por ejemplo, dos ranuras que se intersectan en un punto C. Supongamos que la primera ranura en generarse sobre el lado AB es aquella con punto origen el punto P1, esta ranura tendrá como dominio la totalidad del lado pues se trata de la única ranura generada en AB, el punto de referncia para esta ranura es el vértice A. Posteriormente se genera una segunda ranura, la que tiene como punto de origen el punto P2 y como punto de referncia el vértice B. Cada ranura tendrá como dominio un porcentaje del lado AB. Interesa determinar el valor de la abscisa del punto de intersección Xc.

El valor de Xc se calcula con la expresión:

$$Xc = \frac{Xa + Xb}{2} + \frac{Yb - Ya}{2}$$

es decir una ranura tendrá como dominio el intervalo Xa <= x <= Xc y la otra tiene como dominio el intervalo Xc <= x <= Xb.

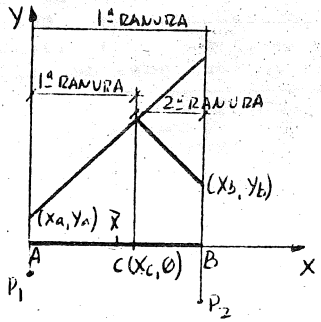


Fig. 3.5 Intersección de ranuras.

Esta situación de intersección de ranuras no se presenta, en cambio, cuando se consideran las dos caras de un lado de una figura es decir diferentes ranuras y funciones de distancia sobre el mismo lado pero desde diferentes sentidos de propagación.

Si dos ranuras obtenidas en diferentes etapas de generación y con el mismo sentido de propagación coinciden en el mismo punto de referencia provocan la siguiente decisión:

sea d1 la distancia hasta el punto de referencia de la primera ranura generada y d2 la distancia hasta el mismo punto pero considerando la segunda ranura.

si d1 > d2 se elimina la primera ranura y queda solamente la segunda.

si d1 <= d2 permanece la primera ranura y la segunda no se mantiene para futuras propagaciones.

3.4.5 CULMINACION DEL PROCESO DE ENRUTAMIENTO.

Si en el proceso de propagación de ruta se llega hasta la ranura desde la cual se accede a "t" entonces la obtención de la ruta termina con éxito, calculándose la distancia hasta "t", pero el proceso de propagación de ranuras continúa y se detiene solamente cuando la distancia mínima contenida en la ranura corriente que se va a propagar es mayor que la distancia alcanzada hasta "t". Esto se comporta así debido a que pueden existir varias rutas desde "s" hasta "t".

3.4.6 PERMANENCIA DE UNA RANURA ACTIVA.

Las funciones de distancia que definen una ranura consideran un valor de distancia mínima hasta un punto de la ranura y un valor de distancia máxima hasta otro punto de ella.

Una ranura permanecerá activa, aún después de haberse propagado mientras la distancia mínima guardada y actualizada en la ranura corriente que se va a propagar sea menor que la distancia máxima para la ranura.

En aquellos casos en los que exista intersección de ranuras es claro que la ranura adicional que modifica a una ya existente puede propagarse en su dominio aún después que la ranura original se haya propagado y siga activa; esto se debe a que, como ya se indicó, cada ranura se maneja de manera independiente. Por otro lado, si la ranura original ya no esta activa, la nueva ranura actuará en un dominio que cubrirá un mayor porcentaje (posiblemente todo el lado) y tendrá la posibilidad de propagarse de manera normal, como cualquier otra ranura.

3.5 ALGORITMO DETALLADO DE ENRUTAMIENTO.

Los conceptos vertidos en los puntos anteriores se sintetizan en el siguiente algoritmo:

Sea el espacio R^2 en el cual se tienen m obstáculos horizontales, verticales y oblicuos. Sea n el número total de figuras geométricas en que estos obstáculos dividen al espacio R .

Se definen "s" el punto inicial y "t" el punto final o destino, dados por sus coordenadas.

0. Hagamos:
   COLA = NIL, lista ordenada de ranuras activas.
   RUTA = NIL, lista de ranuras propagadas.
   O = s , O será el origen para ranuras.
   d = 0 , d guardará la distancia total desde "s".

1. Para la figura a la que pertenece "s" se obtienen, con origen en "s" las ranuras hacia las figuras adyacentes y se guardan en COLA las mismas en un orden ascendente respecto a la distancia (dist) hasta su punto de referencia.

2. Si COLA esta vacio entonces se finaliza sin éxito la obtención de la ruta (si no se ha encontrado ninguna), si no se sigue a 3.

3. Se saca de la COLA la ranura que tenga dist minima.

4. Se añade esta ranura a RUTA.

5. Se propaga a esta ranura, para esto se hace O = Pi, d = dist.

6. Se ingresan las nuevas ranuras a la COLA y se actualiza esta, es decir se reordena de manera ascendente en relación a las distancias totales desde "s" hasta los puntos de referencia de las ranuras, las cuales se calculan como dist = d + |OPi|.

   Si más de una ranura de la COLA contiene la misma dist minima para todas ellas se sigue el mismo procedimiento de propagación.

   Si una ranura no puede propagarse se vuelve a 2.

7. Si en la propagación se ha llegado a la ranura que contiene a "t" se calcula la distancia hasta t, se guarda ese valor y se pasa a 8; de lo contrario se pasa directamente a 8.

8. Si no existe valor para la distancia hasta "t" se vuelve al paso 3.

   Si existe valor para la distancia hasta "t" se compara el valor del contenido de la COLA con esa distancia, si el contenido de la COLA sigue siendo menor que la distancia hasta "t" se vuelve a 3, de lo contrario finaliza el procedimiento y se reconstruye la sucesión de puntos que determinan la ruta, mediante los apuntadores hacia atras contenidos en cada ranura propagada y que se encuentran en RUTA.

3.5.1.1 VALIDEZ DEL ALGORITMO QUE OBTIENE LA RUTA MINIMA.

El algoritmo presentado y que obtiene la ruta mínima entre dos puntos del plano opera de manera similar al de Dijkstra. Los vértices de la gráfica con los que en general se trabaja en el algoritmo de Dijkstra corresponden a las ranuras generadas sobre los lados de las figuras geométricas.

Las diferencias fundamentales del algoritmo de enrutamiento propuesto con el de Dijkstra consisten, por un lado en el hecho de que en la propagación de ranuras se van generando diferentes rutas desde el punto inicial s hasta el punto final t y; por otro, los pesos de los lados de la gráfica se van determinando conforme se van propagando las ranuras, es decir, inicialmente no se conocen esos pesos.

El algoritmo de Dijkstra supone la formación de un subconjunto S del total de vértices de la gráfica. Los elementos de S son vértices para los cuales ya se conoce la distancia más corta desde el vértice inicial. Al inicio del algoritmo S contiene solamente al vértice inicial y luego, en cada etapa de avance del algoritmo se añade a S un nuevo vértice v cuya distancia desde el vértice inicial es la más corta. En definitiva la secuencia de vértices de la gráfica que conforman la ruta más corta desde el vértice inicial hasta v esta contenida integramente en S.

En el método de enrutamiento de propagación de ranuras se forma también este subconjunto S, uno para cada ruta que parte desde el punto inicial s. Estos subconjuntos estan guardados en la lista de ranuras propagadas RUTA, esta lista esta inicialmente vacia (paso 0 del algoritmo) y se añade un elemento a ella cada vez que se recupera de COLA el elemento de distancia mínima (paso 4 del algoritmo).

Para verificar la validez del algoritmo de enrutamiento propuesto se va a probar por inducción que para cada ranura r perteneciente a uno de los subconjuntos S contenidos en RUTA la distancia d[r] corresponde a la longitud de la ruta mínima desde el punto inicial s hasta la ranura r.

En efecto, cuando el subconjunto S contiene un solo elemento este corresponde a la ranura cuyo punto de distancia mínima al origen es el punto inicial s. La longitud de la ruta mínima desde el punto inicial s hasta si mismo es 0, de ahí que se inicializa la variable d en cero (paso 0 del algoritmo).

En el caso general, cuando el subconjunto S contiene más de un elemento. Sea w la ranura recuperada desde el frente de la cola de prioridades (paso 3 del algoritmo). Si d[w] no es la longitud de la ruta mínima desde el punto inicial s hasta la ranura w entonces debe existir una ruta de longitud mínima P la cual debe contener alguna ranura diferente a w y que no esta contenida en S.

Sea x la primera de estas ranuras en P. Entonces la distancia desde el punto inicial s hasta la ranura x es más corta que la distancia d[w] y además, la ruta mínima desde el punto inicial s hasta la ranura x esta contenida integramente en S excepto por la misma ranura x, tal como se muestra en la Fig.3.6.

Entonces, por la hipótesis de inducción se tendría que d[v] < d[w], lo que constituye una contradicción al haberse recuperado a w del frente de la cola como la ranura de distancia mínima.

Se concluye entonces que no puede existir la ruta P y d[w] es la longitud de la ruta mínima desde el punto inicial s hasta la ranura w. Para más detalle referente a este punto véase [10] y [3].

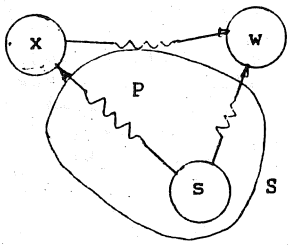


Fig. 3.6 Ruta mínima P.

3.6 COMPLEJIDAD DEL ALGORITMO.

Se debe distinguir al efectuar el análisis del algoritmo de enrutamiento las diversas operaciones que se ejecutan en los diferentes pasos del algoritmo:

Sea n el número de figuras de la gráfica de figuras.

La determinación de la figura a la cual pertenece la ranura que se propaga es O (log(n)) por cada vez que se busque, pues las figuras de la gráfica estan guardadas en una estructura de árbol.

La obtención de las nuevas ranuras provenientes de una propagación tiene que ver con el recorrido de la lista de figuras vecinas y ya se vió que ese recorrido puede ser, en el peor de los casos de n/2 figuras cada vez, por lo que el algoritmo en esta parte es O (n^2) en el peor de los casos pudiendo llegar a ser O (n).

La inserción de una ranura en la cola, la recuperación de la ranura de distancia mínima, y la eliminación de un elemento de la cola pueden ser efectuadas, cada una de ellas con un orden de O (log(r)) cada vez, donde r es el número de ranuras.

Como pueden haber O (n^2) ranuras el orden global del algoritmo de enrutamiento es de:

$$O(n^2 \log(n)).$$

REFERENCIAS Y BIBLIOGRAFIA.

[1] Y.F.Wu, P. Widmayer, M.D.F. Schlag, C.K. Wong
"Rectilinear Shortest Paths and Minimun Spanning Trees in the Presence of Rectilinear Obstacles" IEEE Transactions on Computers. Vol C-36 No.3, March 1987. pp 321-331.

[2] S.Futagami, J. Shirikawa, H. Ozaki.
"An Automatic Routing Sistem for single-layer Printed Wiring Boards". IEEE Transactions on Circuits and Systems, Vol CAS-29, No.1, January 1982.pp. 46-51.

[3] D.M. Mount
"Voronoi Diagrams on the Surface of a Polyhedron" Tech. Rept 121, Center for Automation Research, University of Maryland, (May 1985).

[4] C.Y. Lee
"An algorithm for path connections and its applications". IRE Trans. Electron. Comp., vol EC-10, pp 346-365, Sept 1961.

[5] T. Lozano Pérez
"Automated planning of manipulator transfer movements". IEEE Trans. Syst., Man, Cybern, vol SMC-11, pp 681-698, Oct 1981.

[6] A. Guttman
"R-Trees: A dynamic index structure for spatial searching". ACM SIGMOD, Proc. SIGMOD 84, Vol 14, No 2, June 1984.

[7] H. Samet
"The quadtree and related hierarchical data structures". Computing Surveys Vol 16, No 2, June 1984.

[8] R.E. Tarjan
"Data Structures and Networks Algorithms". Society for Industrial and Applied Mathematics, 1983.

[9] N. Deo
"Graph Theory with Applications to Engineering and Computer Science" Prentice-Hall, Inc. 1974.

[10] A. Aho, J. Hopcroft, J. Ullman
"The Design and Analysis of Computer Algorithms". Addison-Wesley Publishing Co. 1974.

Capítulo 2

Programación Funcional y Lógica

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

Un Sistema de Tipos Polimórfico para un Lenguaje Funcional con Definiciones de Constantes y Tipos Abstractos

Verónica Gaspes
Alvaro Tasistro *

ESLAI - Argentina
Escuela Superior Latinoamericana de Informática.
Casilla de Correo 9193
(1000) Buenos Aires - Rep. Argentina

Resumen. Se define un lenguaje funcional con definiciones de constantes globales y locales y de tipos abstractos. Se define un sistema de tipos polimórfico para ese lenguaje. La semántica del lenguaje y el sistema de tipos se expresan como sistemas de deducción en el estilo de la semántica natural.

Contenido.

1. Introducción.
   1.1 La noción de tipo en programación.
   1.2 Sistemas de tipos.
   1.3 Estructura del trabajo.

2. Especificación del lenguaje.
   2.1 Sintaxis y descripción informal.
   2.2 Fòrmalismo de especificación.
   2.3 Semántica formal.

3. Sistema de Tipos.
   3.1 Especificación.
   3.2 Variables, constantes y polimorfismo.

4. Desarrollos Complementarios y Conclusiones.
   4.1 Propiedades del sistema de tipos.
   4.2 Conclusiones.

Referencias.

* Tambien: Universidad de la República - Uruguay.
Instituto de Computación. Julio Herrera y Reissig 565, Piso 5
Montevideo, Uruguay

1. Introducción.

1.1 La noción de tipo en programación.

El desarrollo del concepto de tipo adquiere motivaciones en dominios más amplios que el de los lenguajes de programación, en general relacionados con teorías formales (ej.: teoría de conjuntos) y se liga al de la clasificación de los objetos de un cierto universo en términos de los propósitos para los cuales ellos son usados [CarWeg 85].

En particular en el cálculo λ todos los objetos son, en principio, representados por una clase de objetos sintácticos (expresiones) interpretados uniformemente como funciones. La noción de tipo aparece a raíz de la especialización de ciertas expresiones como representaciones de funciones específicas.

Con este enfoque, las aplicaciones sólo podrán ser interpretadas cuando el operando denote un objeto del dominio de la función representada por el operador. Si esta restricción no es impuesta en las reglas de formación de expresiones, entonces el lenguaje admitirá expresiones sintácticamente válidas sin significado.

Las expresiones construidas sin respetar una semántica preestablecida son llamadas en programación programas erróneos. Los aspectos de *seguridad* en programación tienen que ver con los mecanismos en base a los cuales puede detectarse y evitarse la construcción de programas erróneos.

Desde este punto de vista, el desarrollo del concepto de tipo en los lenguajes de programación tiene por objeto la definición de lenguajes en los cuales pueda imponerse que cada programa bien expresado deba satisfacer una cierta especificación. En un lenguaje con tipos cada expresión bien formada satisface una cierta especificación (eventualmente incompleta) que es llamada su tipo.

El concepto de tipo es también útil como herramienta de diseño. Los programas pueden ser interpretados como transformaciones entre objetos con propiedades conocidas. Las propiedades de los objetos y las transformaciones, una vez especificadas, deben ser representadas en el lenguaje de programación. Las primitivas del lenguaje definen un nivel de especificaciones para las cuales la construcción es trivial. Para problemas arbitrariamente complejos es útil considerar niveles de descripción en los cuales ciertas especificaciones convenientemente escogidas son usadas como primitivas. Cada una de ellas se representa separadamente en forma análoga hasta alcanzar el nivel de las primitivas del lenguaje de programación.

La anterior estrategia es la base de las técnicas de partición y refinamiento en programación y puede explicarse en términos de tipos: en un lenguaje dado, los tipos primitivos definen una clase de objetos con propiedades precisamente especificadas y representación inmediata. La posibilidad de definir tipos adecuados para construir descripciones abstractas de problemas y desarrollar las representaciones de estos tipos en forma modular constituye otra línea de interés en el desarrollo de los lenguajes de programación, asociada a los conceptos de *tipo abstracto de datos*, *encapsulamiento* y *modularización*.

1.2 Sistemas de tipos.

Los sistemas de tipos son los mecanismos que formalizan la introducción del concepto de tipo en los lenguajes de programación. Los tipos de las expresiones pueden estar ya presentes cuando se define la sintaxis y la semántica de éstas, o bien pueden introducirse a posteriori, sobre un lenguaje originalmente definido sin tipos. Este último es el enfoque seguido en este trabajo.

Un sistema de tipos define un conjunto de expresiones como tipos válidos y reglas de asociación de tipos a expresiones del lenguaje. Aquellas expresiones a las que el sistema no asigna tipo son, de hecho, excluídas del lenguaje. El objetivo de seguridad comentado antes puede ahora ser formulado como un requerimiento para el sistema de tipos [NorPet 89]: *serán excluídas del lenguaje exactamente aquellas expresiones en las cuales una operación sea usada fuera de su dominio.*

La tradición predominante en los lenguajes de programación impone también la restricción de que la verificación de tipos de las expresiones sea realizada en tiempo de compilación, por lo cual la relación entre expresiones y tipos:

a tiene tipo *A*

debe ser decidible. Este último requisito excluye los sistemas en los que los tipos representan especificaciones completas de programas, como los desarrollados a partir de [M-Lof 82], como en [NorPet 89].

Como consecuencia, en la generalidad de los lenguajes de programación los tipos denotan propiedades parciales de las expresiones. Más aún, el requisito de seguridad sobre las expresiones con tipo sólo se satisface parcialmente: son permitidas ciertas expresiones que, al ser evaluadas, resultan en la aplicación de operaciones fuera de su dominio (ej.: división por cero, cabeza o resto de la lista vacía). Estos casos son resueltos adaptando la semántica del lenguaje de modo que les sean asignados valores de error especiales. En general, la *robustez* de un sistema de tipos tiene que ver con la clase de restricciones que es impuesta a las expresiones para garantizar su buen comportamiento.

La noción de robustez es la versión informal del concepto de consistencia del sistema de tipos como sistema formal en relación a un modelo de comportamiento de las expresiones determinado por la semántica del lenguaje. Simétrico al de consistencia, existe el concepto de completitud, que podría expresarse por el requisito: *todas aquellas expresiones con sentido deben ser aceptadas por el sistema de tipos.* Informalmente, esta propiedad es manejada como la *flexibilidad* del sistema de tipos.

Obviamente el extremo de flexibilidad es ocupado por los lenguajes sin tipos (LISP, Scheme entre los funcionales). Opuestos a ellos son los lenguajes en que los tipos se introducen ya al definir la sintaxis y la semántica de las expresiones de modo de asegurar que cada una de ellas recibe un único tipo. En estos casos sólo se asigna semántica a las expresiones con tipo, de modo que el sistema es robusto por definición. El caso caracteriza a los sistemas *monomórficos*. En estos sistemas son expresiones diferentes (y deben ser definidas por separado) la que toma la cabeza de una lista de enteros y la que aplica idéntica transformación a una lista de booleanos. El caso es común en los lenguajes imperativos (ej.: Pascal) donde esta decisión de diseño puede aprovecharse para optimizar el código generado en la compilación, pero sus desventajas en flexibilidad ya comienzan a

pesar cuando esos lenguajes son usados en la práctica y son definitivamente inaceptables para lenguajes funcionales cuyas mejores propiedades provienen de las facilidades que proveen para construir módulos de aplicación general. Frente a este panorama, la práctica comprueba la utilidad de la noción de tipo desde el punto de vista de la seguridad de la programación y justifica el interés por definir un compromiso aceptable entre los extremos arriba mencionados.

La alternativa a la rigidez de los sistemas monomórficos reside precisamente en un enfoque en que la semántica de las expresiones sea definida sin utilizar el concepto de tipo. De este modo, operadores genéricos (como "cabeza de lista") pueden ser definidos sin restricciones. Luego se da un sistema de reglas que asigna tipos a las expresiones del lenguaje según su forma, y en el cual distintas ocurrencias de un operador genérico pueden recibir distintos tipos según cómo sean utilizadas. Este enfoque caracteriza a los sistemas *polimórficos*.

Vinculada a la noción de polimorfismo está la de *inferencia de tipos*. Como se ha dicho, la asignación de tipos a expresiones pueden darse por reglas que siguen la estructura de las últimas, por lo cual la decisión del tipo de una expresión puede hacerse sin necesidad de declaraciones redundantes como en los lenguajes convencionales. El uso obligatorio de las declaraciones es incómodo, sobre todo cuando no representan una herramienta de estructuración sino apenas un requisito formal.

Por último, la utilización de la idea de tipo como mecanismo abstracto de diseño puede incorporarse en los lenguajes de programación en la forma de definiciones de tipos, usualmente llamados *tipos abstractos*. Definir un tipo abstracto implica definir una clase de expresiones cuyas propiedades deben ser especificadas como los resultados de un cierto conjunto de operaciones asociadas al tipo. Tales resultados se definen en términos de expresiones de otros tipos predefinidos. Por lo anterior, las definiciones de tipos abstractos deben proveer medios para especificar el conjunto de operaciones asociadas.

1.3 Estructura del trabajo.

En [Milner 78] y [Holm 89] se definen sistemas de tipos polimórficos para un lenguaje funcional mínimo.

El propósito de este trabajo es proveer una definición análoga para un lenguaje funcional extendido. Las extensiones propuestas son:

- Añadir como constructores de tipos los operadores de *producto cartesiano*, *unión disjunta* y el *tipo de un elemento* ("1").
- Introducir *definiciones de tipos abstractos*.
- Introducir *definiciones de constantes globales*.

Para los dos primeros puntos se siguen las sugerencias de [Holm 89].

Para cumplir el propósito mencionado, debe definirse un lenguaje funcional con las referidas construcciones. En el capítulo 2, acompañadas de comentarios informales, aparecen la sintaxis y la semántica del lenguaje, presentándose asimismo el formalismo utilizado para la especificación, que es del estilo de la semántica natural [Kahn 87].

El capítulo 3 presenta el sistema de tipos en el mismo formalismo. Finalmente, el capítulo 4 contiene conclusiones y comentarios sobre desarrollos complementarios.

2. Especificación del lenguaje.

2.1 Sintaxis y descripción informal.

Se considera un lenguaje funcional con dos clases de expresiones: términos evaluables y definiciones. Los términos evaluables son básicamente los términos del cálculo λ equipado con constantes primitivas y definiciones de constantes locales. La siguiente es su sintaxis:

[Términos Evaluables]
$t \in$ E-TERM sii

$t ::= x$		*[Variable]*
c		*[Constante]*
$\lambda x.d$		*[Abstracción Funcional]*
$(d\ e)$		*[Aplicación]*
true	**false**	*[Booleanos]*
num n		*[Números]*
()		*[Nulo]*
$+\ d\ e$	$-\ d\ e$	
$*\ d\ e$	$/\ d\ e$	*[Aritméticas]*
$=\ d\ e$		*[Igualdad]*
and $d\ e$		
or $d\ e$	**not** d	*[Lógicas]*
pair $d\ e$		
fst d	**snd** d	*[de Pares]*
inl d	**inr** d	
case $d\ x.e\ y.f$		*[de Uniones Disjuntas]*
if $d\ e\ f$		*[Condicional]*
local $c\ e$ **in** d		*[Def. de constante local]*
abs$(T)\ d$	**rep**$(T)\ d$	*[de Tipos Abstractos]*

donde
$x, y \in$ VAR (variables)
$c \in$ CNST (constantes)
$n \in \{1, 2, \ldots$
$d, e \in$ E-TERM (términos evaluables)
$T \in$ TYPE-CNST (constantes de tipos)
VAR, CNST, TYPE-CNST disjuntos

En el lenguaje se distinguen las constantes de las variables: las primeras son introducidas por definiciones y las segundas son las variables ligadas en las abstracciones funcionales. Esto se expresa en la existencia de los conjuntos disjuntos VAR y CNST.

Las primeras cuatro formas corresponden a los términos del cálculo λ con constantes. Luego hay constantes primitivas: los booleanos, números, el elemento nulo (único de un tipo 1) y los operadores aritméticos, relacionales y lógicos. Luego se incluyen los operadores asociados a los pares ordenados (constructor pair y selectores fst y snd) y uniones disjuntas (constructores inl y inr y discriminador case). Hay, naturalmente, un condicional. La construcción local corresponde a la definición de constantes locales y abs y rep están relacionados con la manipulación de elementos de tipos abstractos como se explicará más adelante.

Las expresiones del lenguaje se completan considerando las definiciones: hay definiciones de constantes globales y de tipos abstractos, con las formas:

[*Definición de constante global*]
cdef ∈ CNST-DEF sii

cdef ::= define *c d*

donde *c* ∈ CNST, *d* ∈ E-TERM

■

[*Definición de tipo abstracto*]
tdef ∈ TYPE-DEF sii

tdef ::= abstype *T* (*Xlist*) = *Tterm* with *cdeflist*

donde *T* ∈ TYPE-CNST, *Xlist* ∈ TYPE-VAR*,
Tterm ∈ TYPE-TERM, *cdeflist* ∈ CNST-DEF*

■

Las definiciones de constantes globales introducen constantes asociadas a términos evaluables. Informalmente, una constante es un nombre para un término evaluable complejo.

El lenguaje tiene también un lenguaje de tipos, según se define en la siguiente sintaxis:

[*Términos de Tipos*]
T ∈ TYPE-TERM sii

Tterm ::=	*t*	¦	[*Variable de tipo*]
	1	¦	[*Tipo de un elemento*]
	bool	¦	[*Booleano*]
	number	¦	[*Números*]
	$A \longrightarrow B$	¦	[*Funciones*]
	$A \times B$	¦	[*Pares*]
	$A \cup B$	¦	[*Uniones Disjuntas*]
	T (*Ttermlist*)		[*Tipo abstracto*]

donde *t* ∈ TYPE-VAR, *A*, *B* ∈ TYPE-TERM
T ∈ TYPE-CNST, *Ttermlist* ∈ TYPE-TERM*

■

Las definiciones de tipos abstractos introducen constantes de tipos asociadas a términos del lenguaje de tipos, como en el siguiente ejemplo, que corresponde a la definición del tipo *lista de objetos genéricos*:

abstype list (X) = 1 ∪ (X × list(X))
with [...]

La definición especifica básicamente la representación del tipo abstracto introducido en términos de tipos predefinidos. Lo primero es el nombre del nuevo tipo (una constante de tipo). Luego sigue la lista de variables de tipo de las que depende la definición. Las variables de tipo denotan, obviamente, tipos genéricos. A la derecha de =, un término de tipo describe la representación del tipo definido.

Lo que sigue a with es una lista de definiciones de constantes globales que representarán las operaciones asociadas al tipo abstracto. Para dar las definiciones de estas operaciones se usa un mecanismo similar al usado en ML [COUHP 85] y sugerido también en [Holm 89]:

Cada constante de tipo *T* introducida por una definición de tipo abstracto tiene asociadas dos constantes primitivas abs(*T*) y rep(*T*) que pueden evaluarse cuando son aplicados a un término

evaluable, (ver sintaxis de los términos evaluables). Informalmente: abs(*T*) aplicado a un término evaluable que tiene por tipo a la representación de *T* da como resultado el objeto de tipo *T* representado por ese término evaluable. Simétricamente, rep(*T*) aplicado a un objeto de tipo *T* da como resultado un objeto cuyo tipo es el de la representación de *T*. De acuerdo a lo sugerido por esta definición, abs(*T*) será llamado el *constructor* del tipo abstracto *T* y rep(*T*) su *destructor*.

Usando estas primitivas pueden definirse operaciones para cada tipo abstracto en términos de las operaciones asociadas a los tipos usados en su representación. Un ejemplo es la definición de la operación que toma cabeza de una lista genérica:

define hd λx.case (rep(list) x) (z.*fail*) (z.fst z)

La constante -hd- se define como una abstracción funcional, que espera un argumento -x- que debe ser una lista (definida como en el ejemplo anterior). Las listas se representan por uniones, por lo cual la expresión -rep(list) x- es, o bien el objeto () (único del tipo 1), o bien un par. Para discriminar entre objetos de una unión se usa la construcción case.

La semántica de -case *d* x.*e* y.*f*- es como sigue: *d* es un objeto de una unión disjunta A ∪ B. Se aplica una de las dos abstracciones -λx.*e* ó λy.*f*- a *d* según éste pertenezca a A o B, respectivamente. El orden entre los operandos de la unión es, pues, relevante para interpretar case.

En el ejemplo, cada uno de los tres operandos de case se encierra entre paréntesis, para distinguirlos. Si -rep(list) x- es () entonces le será aplicada la abstracción de resultado constante *fail*, que es una expresión convencional de error agregada al conjunto de valores. En otro caso, la representación de x es un par, del cual se toma el primer componente, que es precisamente la cabeza de la lista.

A cada tipo abstracto puede, como se ha dicho, asociarse una lista de definiciones como la anterior. Puede imponerse que las primitivas abs y rep sólo sean usadas en el alcance de with, dentro de la definición del tipo abstracto, con lo cual las operaciones definidas para éste esconderían su representación. Esta es la propiedad de encapsulamiento en las definiciones de tipos abstractos.

El efecto de las definiciones es esencialmente modificar el contexto en que se computan los términos evaluables. Para especificar este efecto debe darse significado a una secuencia de expresiones del lenguaje. Esta secuencia puede pensarse como una abstracción de un ciclo del estilo "*Read-Eval-Print*". Aquí se define como una sesión:

[*Sesiones*]
SESS = MESS*

[*Mensajes*]
MESS = CNST-DEF ∪ TYPE-DEF ∪ E-TERM

Ahora puede definirse formalmente la semántica del lenguaje pero previamente se presentará el formalismo a ser usado.

2.2 Formalismo de especificación.

Para dar la semántica del lenguaje podría usarse semántica denotacional (como en [Milner 78]) o de reducciones (como en [Holm 89]). Aquí se prueba como alternativa una variante en el estilo de la semántica natural presentada en [Kahn 87] como un lenguaje para definir uniformemente semántica y sistemas de tipos, y aplicada de esa forma en [CLÓDDK 86]. La aproximación natural supera a la denotacional en claridad y en la abstracción ganada al definir el sistema de tipos. No es incompatible con la especificación de cálculo de reducciones como en [Holm 89]. Como ella, es operacional, en el sentido que especifica un procedimiento para la computación de las expresiones y estructural, en el sentido que sigue la estructura de las expresiones. En el capítulo 4 el formalismo elegido es evaluado en relación al enfoque de reducciones.

La idea general en la semántica natural es construir *sistemas de deducción* que caractericen predicados a ser definidos sobre una clase de expresiones. Por ejemplo, la evaluación de una *expresión* E a un *valor* v, realizada en un *contexto* ρ puede caracterizarse por un predicado de la forma:

$$\rho \vdash E \longrightarrow v$$

llamado un *secuente*.

En el caso anterior, el valor de la expresión E puede depender no sólo de la forma de E sino de valores asociados a ciertos objetos sintácticos que ocurren en E (en general, identificadores). Los contextos representan, precisamente, la referida asociación entre dichos objetos sintácticos y valores.

Una especificación en este formalismo debe interpretarse como un sistema de deducción, en el que se pretende verificar si aserciones como la anterior son teoremas. El sistema de deducción se define dando axiomas y reglas de inferencia.

Las reglas de inferencia son de la forma:

$$\frac{f_1 \; . \; . \; . \; f_n \;\|\; c_1 \; . \; . \; c_m}{f}$$

es decir, constan de un *numerador* y un *denominador*. El primero es un conjunto no ordenado de secuentes $-f_i-$ (*premisas*) y un conjunto de fórmulas $-c_j-$ (*condiciones* que deben verificarse para aplicar la regla). El denominador es un secuente llamado la *conclusión* de la regla.

Los axiomas pueden verse como reglas sin numerador. Una *especificación* es un conjunto no ordenado de reglas.

Para especificar semántica de un lenguaje como el considerado se usarán, en general, reglas de la forma:

$$\frac{\rho_1 \vdash E_1 \longrightarrow v_1 \qquad \rho_n \vdash E_n \longrightarrow v_n}{\rho \vdash E \longrightarrow f(v_1, \ldots, v_n)}$$

y, por supuesto, axiomas:

$$\rho \vdash E \longrightarrow v$$

Evaluar una expresión e en un contexto ρ significa, con esta semántica, encontrar una demostración de

$$\rho \vdash e \longrightarrow v$$

para algún v.

Con esto en mente, las reglas con premisas pueden interpretarse como sigue: para evaluar E en ρ, evalúense $E_1, \ldots, E_n$ en $\rho_1, \ldots, \rho_n$ y apliquese f a los resultados obtenidos. Los axiomas

corresponden a expresiones que pueden evaluarse sin realizar ulteriores cálculos. Esta interpretación da el carácter operacional de la especificación.

Los secuentes y condiciones en semántica natural son fórmulas de un lenguaje de primer orden, de modo que, para construir la especificación requerida, debe darse una sintaxis para contextos, expresiones y valores. Más aún, símbolos como f en las reglas de arriba son constructores sintácticos, en el caso, de valores.

En este trabajo se usan sobre contextos y expresiones funciones que no son constructores sintácticos y que son especificados separadamente. Esto se aparta de la semántica natural pura pero simplifica significativamente la especificación.

Se usarán también algunas convenciones de modularización: una especificación puede recibir un nombre y ser referenciada por él en la construcción de otras especificaciones. Esto se indica escribiendo el nombre de la especificación referenciada como superíndice del símbolo $\vdash$.

La semántica del lenguaje considerado se da en la sección siguiente.

2.3 Semántica formal.

Primeramente se definen los contextos, expresiones y valores sobre los que se construirá la especificación.

Por supuesto, las expresiones a ser consideradas serán en este caso los mensajes y sesiones del lenguaje. La evaluación de una expresión será caracterizada por una regla que, o bien es axioma, o bien tiene premisas que representan las evaluaciones de las expresiones componentes de la original. Esto muestra la condición estructural de la especificación. Los valores serán un subconjunto de las expresiones del lenguaje.

Por su parte, los contextos para la evaluación de mensajes y sesiones deben reflejar el conjunto de constantes y tipos que las sucesivas definiciones introducen. Para esto se definen:

[*Contexto de Constantes*]
$\gamma \in$ CNST-CNTXT sii
γ: CNST $\rightarrow\!\!\!\!\!+$ E-TERM
dada como conjunto de pares
■
[*Contexto de Tipos*]
$\tau \subseteq$ TYPE-CNST
■
[*Contexto de Constantes y Tipos*]
$\rho \in$ CT-CNTXT sii
$\rho \in$ CNST-CNTXT $\times$ TYPE-CNTXT
■

Además de estas definiciones se usarán, como se ha dicho, algunas operaciones sobre contextos y expresiones que no son constructores sintácticos. En particular, ciertos contextos son representados por funciones parciales dadas por extensión. Sobre éstas se usará:

[*Unión*]

$\chi \cup \langle\langle a,b\rangle\rangle$

donde χ es una función parcial dada por extensión si $\langle a,c\rangle \in \chi$ entonces se sustituye por $\langle a,b\rangle$

■

[*Selector*]

$\chi(a)$ está definido si $\exists\ \langle a,b\rangle \in \chi$ y, en ese caso, denota b

■

Ahora pueden presentarse las reglas (resumidas) que definen la evaluación de los términos evaluables. La especificación es llamada **TermEval**.

TermEval

Se definen predicados de la forma:

$$\rho \vdash e \longrightarrow v$$

donde ρ es un contexto de constantes y tipos,
$e \in$ E-TERM $\cup$ FIX
$v \in$ E-TERM

y

FIX = $\langle$fix x e, con $x \in$ VAR, $e \in$ E-TERM $\cup$ FIX$\rangle$

En lo que sigue, $c \in$ CNST, $x,y \in$ VAR,
$d,e,f,g \in$ E-TERM
$n \in \langle 1,2,\ldots$

[*Términos Canónicos*]

$\rho \vdash \lambda x.d \longrightarrow \lambda x.d$
$\rho \vdash$ **true** $\longrightarrow$ **true**
$\rho \vdash$ **false** $\longrightarrow$ **false**
$\rho \vdash$ **num** n $\longrightarrow$ **num** n
$\rho \vdash$ **()** $\longrightarrow$ **()**
$\rho \vdash$ **pair** d e $\longrightarrow$ **pair** d e
$\rho \vdash$ **inl** d $\longrightarrow$ **inl** d
$\rho \vdash$ **inr** d $\longrightarrow$ **inr** d
$\rho \vdash$ **abs**(T) d $\longrightarrow$ **abs**(T) d

[*Constante*]

$$\frac{\langle \gamma,\tau\rangle \vdash \gamma(c) \longrightarrow v}{\langle \gamma,\tau\rangle \vdash c \longrightarrow v}$$

[*Aplicación*]

$$\frac{\rho \vdash d \longrightarrow \lambda x.f \quad \rho \vdash [e/x]f \longrightarrow v}{\rho \vdash (d\ e) \longrightarrow v}$$

[*Aritméticas*]
[*Igualdad*]

$$\frac{\rho \vdash d \longrightarrow \textbf{num}\ m \quad \rho \vdash e \longrightarrow \textbf{num}\ n}{\rho \vdash +\ d\ e \longrightarrow \textbf{sum}\ (m,n)}$$

[*Lógicas*]

$$\frac{\rho \vdash d \longrightarrow \textbf{false}}{\rho \vdash \textbf{and}\ d\ e \longrightarrow \textbf{false}}$$

.....

$$\frac{\rho \vdash e \longrightarrow \text{false}}{\rho \vdash \text{and } d\ e \longrightarrow \text{false}}$$

$$\frac{\rho \vdash d \longrightarrow \text{true} \quad \rho \vdash e \longrightarrow \text{true}}{\rho \vdash \text{and } d\ e \longrightarrow \text{true}}$$

[...]

[Selectores de Pares Ordenados]

$$\frac{\rho \vdash d \longrightarrow \text{pair } e\ f \quad \rho \vdash e \longrightarrow v}{\rho \vdash \text{fst } d \longrightarrow v}$$

$$\frac{\rho \vdash d \longrightarrow \text{pair } e\ f \quad \rho \vdash f \longrightarrow v}{\rho \vdash \text{snd } d \longrightarrow v}$$

[Discriminador de Uniones Disjuntas]

$$\frac{\rho \vdash d \longrightarrow \text{inl } g \quad \rho \vdash [d/x]e \longrightarrow v}{\rho \vdash \text{case } d\ x.e\ y.f \longrightarrow v}$$

$$\frac{\rho \vdash d \longrightarrow \text{inr } g \quad \rho \vdash [d/y]f \longrightarrow v}{\rho \vdash \text{case } d\ x.e\ y.f \longrightarrow v}$$

[Condicional]

$$\frac{\rho \vdash d \longrightarrow \text{true} \quad \rho \vdash e \longrightarrow v}{\rho \vdash \text{if } d\ e\ f \longrightarrow v}$$

$$\frac{\rho \vdash d \longrightarrow \text{false} \quad \rho \vdash f \longrightarrow v}{\rho \vdash \text{if } d\ e\ f \longrightarrow v}$$

[Definición de Constante Local]

$$\frac{\langle \gamma \cup \{\langle c, \text{fix } xc\ xce\rangle\}, \tau\rangle \vdash d \longrightarrow v}{\langle \gamma, \tau\rangle \vdash \text{local } c\ e \text{ in } d \longrightarrow v}$$

$[xc = \text{CTOV}\ \langle c, c\rangle,\ xce = \text{CTOV}\ \langle c, e\rangle]$

[Operador de Punto Fijo]

$$\frac{\rho \vdash [\text{fix } x\ e/x]e \longrightarrow v}{\rho \vdash \text{fix } x\ e \longrightarrow v}$$

[Destructor de Tipos Abstractos]

$$\frac{\langle \gamma, \tau\rangle \vdash e \longrightarrow \text{abs}(T)\ f \quad \langle \gamma, \tau\rangle \vdash f \longrightarrow v \parallel T \in \tau}{\langle \gamma, \tau\rangle \vdash \text{rep}(T)\ e \longrightarrow v}$$

■

Las primeras reglas son axiomas. Corresponden a términos que denotan ellos mismas su propio valor. Usando terminología de reducciones se denomina a éstos, términos canónicos. De hecho los términos canónicos definen el conjunto de valores posibles de las términos evaluables. Es decir, si un término evaluable puede evaluar a v en cualquier contexto, entonces v es un término canónico, según se deduce fácilmente de las formas de las reglas.

El siguiente caso corresponde a las constantes definidas. La evaluación de una constante se expresa como la evaluación del

término evaluable asociado a ella por la definición que la introdujo. Esta asociación debe estar definida en el contexto de nconstantes.

Luego se define la evaluación de una aplicación funcional: se evalúa el operador de la aplicación que debe resultar en una abstracción funcional. Luego se evalúa el cuerpo de la abstracción resultante, donde las ocurrencias del argumento nominal son sustituídas por el operando de la aplicación. Se usa la *sustitución* en términos evaluables, definida como en el cálculo λ.

Claramente, la regla de la aplicación representa la regla *β* del cálculo λ. Desde que el operando de la aplicación no es evaluado, se está sugiriendo una evaluación en orden normal. De hecho, la intención es que la especificación admita una implementación con estrategia de evaluación perezosa ("lazy").

Las reglas *Aritméticas* y de *Igualdad* corresponden a las evaluaciones de los correspondientes términos evaluables (ver sintaxis) y son todas análogas a la de la suma. Los operandos deben ser evaluados a números y se usa una primitiva SUM con operandos números y la semántica habitual de la suma. Las otras reglas se omiten para ahorrar espacio.

Las reglas de evaluación llamadas *Lógicas* no son, en principio, muy diferentes de las anteriores. Sin embargo, se hace notar en este caso la intención de implementar una estrategia de evaluación perezosa, desde que se evalúan sólo los operandos que sean necesarios para determinar el resultado de la operación.

Las reglas para *Selectores de Pares Ordenados* son bastante autoexplicativas. El operando en ambos casos debe ser evaluado a un par y el resultado de la operación original es el operando del constructor de pares que corresponde a cada proyección.

Las reglas para el *Discriminador de Uniones Disjuntas* formalizan la explicación dada más arriba. Se omite, pues, mayor detalle.

Las reglas para el *Condicional* son también bastante obvias. Nuevamente se hace notar la estrategia perezosa de evaluación.

Más interesante es la evaluación de las definiciones locales. En *local c ≡ e in d* se define la constante *c* como una abreviatura para el término evaluable *e*, lo cual debe considerarse al evaluar *d*. El término *e* puede contener ocurrencias de la constante *c*, lo cual puede interpretarse, o bien como una definición recursiva o bien como una redefinición de *c*. En este último caso, las ocurrencias de *c* en *e* se interpretan como asociadas a un cierto término evaluable según una definición anterior. En los lenguajes que admiten esta interpretación, existen dos formas de definiciones locales (ej.: *let* y *letrec*) para indicar expresamente cada caso. Aquí se toma la decisión de considerar un único tipo de definicion que es, en general, recursiva, con lo cual se impide que una constante sea redefinida usando una definición anterior, por considerarse que esta práctica no es consistente con el estilo puramente aplicativo.

Para conseguir el objetivo mencionado debe considerarse a *c* definida como punto fijo de una abstracción de la forma λc.e. Esta ultima expresión no está bien formada, sin embargo, puesto que el argumento de una abstracción debe ser una variable y no una constante. La función CTOV se usa para solucionar este problema. Su especificación es:

$$\text{CTOV}(c,e) = [xc/c]e$$
donde $c \in$ CNST, $e \in$ E-TERM, $xc \in$ VAR, xc no ocurre en e.

es decir, las ocurrencias de la constante *c* en *e* son sustituidas por una variable no usada *xc*.

Usando CTOV puede expresarse la definición de la constante *c* como punto fijo de λxc.xce donde *xc* = CTOV(c,c) y *xce* = CTOV(c,e).

La construcción *fix x e* es usada con el propósito de representar el punto fijo de λx.e. No pertenece a E-TERM desde que

no puede ser usado directamente en las expresiones del lenguaje, de donde surge la necesidad de definir FIX aparte de E-TERM, como se hizo en la especificación.

Finalmente, aparece la regla para evaluar el destructor rep de un tipo abstracto T. Este debe estar definido en el contexto de tipos, lo cual se impone como condición para aplicar la regla. El operando de rep(T) debe ser un objeto de tipo T construido por abs(T) aplicado a una cierta representación f. La evaluación del término original es la evaluación de f.

Esto completa la explicación de las reglas de evaluación de términos evaluables. Deben considerarse también las definiciones, las cuales afectan los contextos de evaluación. Como ya se dijo, esto se expresa dando semántica al concepto de sesión, por medio de la especificación SessEval:

SessEval

Se definen predicados

$$\rho \vdash s \longrightarrow v$$

donde ρ es un contexto de constantes y tipos,
$s \in$ SESS
$v \in$ SESS

[*Sesión Vacía*]

$$\rho \vdash [\,] \longrightarrow [\,]$$

[*Definición de Constante Global*]

$$\frac{\langle \gamma \cup \langle\langle c,\ \mathbf{fix}\ xc\ xce\rangle\rangle, \tau\rangle \vdash s \longrightarrow s'}{\langle \gamma, \tau\rangle \vdash [\mathbf{define}\ c\ e\ ;s] \longrightarrow s'}$$

$$(xc = \text{CTOV}\ \langle c, c\rangle,\ xce = \text{CTOV}\ \langle c, e\rangle)$$

[*Definición de Tipo Abstracto*]

$$\frac{\langle \gamma,\ \tau \cup \langle T\rangle\rangle \vdash cdeflist \circ s \longrightarrow s'}{\langle \gamma, \tau\rangle \vdash [\mathbf{abstype}\ T\ (Xlist) = Tterm\ \mathbf{with}\ cdeflist\ ;\ s] \longrightarrow s'}$$

[*Término Evaluable*]

$$\frac{\rho \overset{\text{TermEval}}{\vdash\!\!\!-\!\!\!-\!\!\!-} e \longrightarrow v}{\rho \vdash [e;s] \longrightarrow [v;s]}$$

■

La primera regla es el único axioma de esta especificación y corresponde a la sesión vacía que es la única forma canónica (y valor posible) de sesión.

La siguiente regla se aplica a una sesión cuyo primer mensaje es una definición de constante global. El tratamiento es análogo al explicado antes para las constantes locales: se agrega al contexto de constantes un par que define como expresión asociada a la constante al punto fijo del término evaluable indicado en la definición. Luego, por un tratamiento usual para secuencias, se evalúa el resto de la sesión en el nuevo contexto.

La regla para las definiciones de tipos abstractos tiene una interpretación similar, aunque parezca más compleja. La constante de tipo definida es incorporada al contexto de tipos y la evaluación continúa sobre el resto de la sesión, prefijada con la lista de definiciones de operaciones para el tipo introducido. Esto se nota por $\circ$.

Finalmente, cuando el primer mensaje de la sesión a evaluar

es un término evaluable, se intenta dar valor a éste usando la especificación **TermEval** para proseguir con el resto de la sesión.

Con esta especificación puede interpretarse la evaluación de una sesión s como la búsqueda de una demostración para el secuente:

$$\langle\langle\rangle,\langle\rangle\rangle \vdash s \Longrightarrow v, \text{ para algún } v$$

es decir, iniciada con contextos vacíos.

Nótese que no cualquier predicado $\rho \vdash E \Longrightarrow v$ podrá ser probado, ya sea en una u otra especificación. De hecho, las alternativas son tres: la prueba puede tener éxito (dando lugar a la evaluación de la expresión) o bien fallar o ser infinita. Ejemplos de las dos últimas alternativas son las evaluaciones de los términos evaluables

$$\textbf{if } \lambda x.x \; d \; e$$

y

$$\textbf{local } c \; c \textbf{ in } c$$

Las evaluaciones fallidas son las de términos evaluables bien formados pero sin significado. Las evaluaciones infinitas corresponden a programas que no terminan. De la especificación **SessEval** puede verse que la evaluación de una sesión falla o es infinita sii falla o es infinita la de algún término evaluable de la sesión.

Precisamente, para impedir la evaluación de términos sin significado se introduce la noción de tipo. El sistema de tipos que se definirá en el capítulo siguiente caracteriza expresiones del lenguaje con significado.

3. Sistema de Tipos.

3.1 Especificación.

La intención en este capítulo es definir sistemas de deducción análogos a los del capítulo anterior para asignar tipos a las expresiones del lenguaje.

Primeramente se definen los tipos del lenguaje.

[Tipos]

$t \in \mathbb{TYPE}$ sii

$t ::=$ **1**	¦	*[Tipo de un elemento]*
bool	¦	*[Booleano]*
number	¦	*[Numero]*
$A \longrightarrow B$	¦	*[Función]*
$A \times B$	¦	*[Producto]*
$A \bigcup B$	¦	*[Unión Disjunta]*
$T(Tlist)$		*[Abstracto]*

donde $A, B \in \mathbb{TYPE}$, $T \in \mathbb{TYPE}$-CNST, $Tlist \in \mathbb{TYPE}*$

■

La asignación de tipo al cuerpo de un término evaluable que introduce variables ligadas dependerá de hipótesis sobre los tipos de estas variables.

Por este motivo se da la siguiente definición:

[Contexto de Variables]

Es una función parcial

$$\varphi : \mathbb{VAR} \rightarrow \mathbb{TYPE}$$

dada por extensión.

■

También se modifica la noción de contexto de tipos, asociando a cada constante de tipo introducida el término de tipo que especifica su representación y la lista de variables de tipo de la que éste depende.

[Contexto de Tipos]
Es una función parcial
τ : TYPE-CNST $\rightarrow\!\!\!\!\!+$ (TYPE-VAR* × TYPE-TERM)
dada por extensión.
■

[Contexto de Inferencia de Tipos]
Es un par $\Gamma = \langle \rho, \varphi \rangle$ donde
ρ es un contexto de constantes y tipos
φ es un contexto de variables
■

Sobre los nuevos contextos definidos se usan las mismas operaciones del capítulo anterior. Los pares miembros de un contexto de variables son notados $x{:}A$.

Ahora puede darse la especificación de la asignación de tipos a términos evaluables, llamada **TermType**:

TermType
Se definen secuentes de la forma
$$\Gamma \vdash e{:}A$$
con $e \in$ E-TERM, $A \in$ TYPE
En lo que sigue $c \in$ CNST, $d,e,f \in$ E-TERM, $x,y \in$ VAR
$A,B,C \in$ TYPE, $T \in$ TYPE-CNST,
$Tterm \in$ TYPE-TERM, $Xlist \in$ TYPE-VAR*

[Un Elemento]
$$\Gamma \vdash ():1$$

[Booleanos]
$$\Gamma \vdash \textbf{true}:\textbf{bool}$$
$$\Gamma \vdash \textbf{false}:\textbf{bool}$$

[Números]
$$\Gamma \vdash \textbf{num}\ n:\textbf{number}$$

[Abstracción Funcional]
$$\frac{\langle \rho, \varphi \cup \langle x{:}A \rangle \rangle \vdash d:B}{\langle \rho, \varphi \rangle \vdash \lambda x.d : A \rightarrow B}$$

[Constructor de Pares]
$$\frac{\Gamma \vdash d:A \qquad \Gamma \vdash e:B}{\Gamma \vdash \textbf{pair}\ d\ e : A \times B}$$

[Constructores de Uniones Disjuntas]
$$\frac{\Gamma \vdash d:A}{\Gamma \vdash \textbf{inl}\ d : A \cup B}$$
$$\frac{\Gamma \vdash d:B}{\Gamma \vdash \textbf{inr}\ d : A \cup B}$$

[Constructor de Tipos Abstractos]
$$\frac{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash d : [\sigma]Tterm \quad \| \quad \tau(T) = \langle Xlist, Tterm \rangle}{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash \textbf{abs}(T)\ d : T([\sigma]Xlist)}$$
$(\sigma$: TYPE-VAR $\rightarrow$ TYPE)

.....

[Variable]

$$\frac{\|\varphi(x) = A}{\langle \rho, \varphi \rangle \vdash x : A}$$

[Constante]

$$\frac{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash \gamma(c) : A}{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash c : A}$$

[Aplicación]

$$\frac{\Gamma \vdash d : A \rightarrow B \qquad \Gamma \vdash e : A}{\Gamma \vdash (d\ e) : B}$$

[Aritméticas]

$$\frac{\Gamma \vdash d : \mathbf{number} \qquad \Gamma \vdash e : \mathbf{number}}{\Gamma \vdash +\ d\ e : \mathbf{number}}$$

[Igualdad]

$$\frac{\Gamma \vdash d : \mathbf{number} \qquad \Gamma \vdash e : \mathbf{number}}{\Gamma \vdash =\ d\ e : \mathbf{bool}}$$

[Lógicas]

$$\frac{\Gamma \vdash d : \mathbf{bool} \qquad \Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash \mathbf{and}\ d\ e : \mathbf{bool}}$$

[...]

[Discriminador de Uniones Disjuntas]

$$\frac{\langle \rho, \varphi \rangle \vdash d : A \cup B \quad \langle \rho, \varphi \cup \langle x : A \rangle \rangle \vdash e : C \quad \langle \rho, \varphi \cup \langle y : B \rangle \rangle \vdash f : C}{\langle \rho, \varphi \rangle \vdash \mathbf{case}\ d\ x.e\ y.f : C}$$

[Selectores de Pares]

$$\frac{\Gamma \vdash d : A \times B}{\Gamma \vdash \mathbf{fst}\ d : A}$$

$$\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathbf{snd}\ e : B}$$

[Condicional]

$$\frac{\Gamma \vdash d : \mathbf{bool} \qquad \Gamma \vdash e : A \qquad \Gamma \vdash f : A}{\Gamma \vdash \mathbf{if}\ d\ e\ f : A}$$

[Definicion de Constante Local]

$$\frac{\langle\langle \gamma \cup \langle\langle c, \mathbf{fix}\ xc\ xce \rangle\rangle, \tau \rangle, \varphi \rangle \vdash d : A}{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash \mathbf{local}\ c\ e\ \mathbf{in}\ d : A}$$

$(xc = \mathbf{CTOV}\ \langle c, c \rangle,\ xce = \mathbf{CTOV}\ \langle c, e \rangle)$

[Operador de Punto Fijo]

$$\frac{\langle \rho, \varphi \cup \langle x : A \rangle \rangle \vdash d : A}{\langle \rho, \varphi \rangle \vdash \mathbf{fix}\ x\ d : A}$$

[Destructor de Tipos Abstractos]

$$\frac{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash d : T[\sigma]Xlist \quad \| \tau(T) = \langle Xlist, Tterm \rangle}{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash \mathbf{rep}(T)\ d : [\sigma]Tterm}$$

$(\sigma : \mathbb{TYPE}\text{-}\mathbb{VAR} \rightarrow \mathbb{TYPE})$

■

Las primeras nueve reglas corresponden a los tipos de las formas canónicas (valores de términos evaluables):

El tipo 1 tiene un único valor **()**, **true** y **false** son los valores de bool, y los valores de **number** son precisamente de la forma num *n*. Esto establecen los primeros axiomas.

Enseguida se tratan los valores de los tipos construidos mediante los constructores primitivos de tipos. El primero de ellos es la abstracción funcional, que representa la forma de los valores del tipo de las funciones del tipo *A* en el tipo *B*. Puede probarse que una abstracción tiene tal tipo si, asumiendo que su argumento tiene tipo *A*, es posible probar que el cuerpo tiene tipo *B*. La hipótesis sobre el tipo del argumento se registra en el contexto de variables.

Las reglas correspondientes a los constructores de pares ordenados y uniones disjuntas son bastante intuitivas. Ellas definen la forma de los valores de dichos tipos.

La asignación de tipo a un término de la forma abs(*T*) *d* merece una explicación más detallada.

En primer lugar, la constante de tipo *T* debe haber sido definida, lo que equivale a establecer, como condición de la regla, que tiene una imagen en τ. Precisamente, $\tau(T)$ denota el par formado por el término de tipo que especifica la representación de *T* -*Tterm*- y la lista de variables de tipo -*Xlist*- que ocurren en él. El término *d* debe ser entonces de un tipo obtenido de *Tterm* por instanciación de sus variables en tipos cualesquiera. Esta instanciación está representada por la función σ que es, de hecho, una sustitución de variables por tipos. La aplicación de σ a *Tterm* se nota por $[\sigma]Tterm$. Finalmente, si el tipo requerido para *d* ha sido probado puede concluirse que el tipo del término original es, precisamente, la instancia de *T* definida por σ, es decir $T([\sigma]Xlist)$.

Los tipos de las variables pueden probarse si han sido introducidos en el correspondiente contexto (al pretender asignar tipo a la expresión que introdujo la variable). Análogamente puede entenderse la regla para las constantes. Existe, con todo, una diferencia: en el caso de la constante, no se registra ésta en el contexto asociada a un tipo sino ál (punto fijo del) término evaluable que ella representa. Las consecuencias de esta decisión son comentadas en 3.2.

Del resto de las reglas, las siguientes son comentadas en detalle, siendo las otras muy intuitivas:

La regla de la aplicación fuerza que el operador tenga el tipo de una función y que el operando tenga tipo del dominio de la misma para dar al resultado el tipo del codominio.

La del discriminador case introduce variables ligadas, lo cual es natural a partir de su semántica.

La regla de las definiciones locales introduce constantes en el contexto correspondiente. Véase que se asocia a la constante la aplicación del operador de punto fijo al término indicado en la definición y no el tipo de éste.

La del operador de punto fijo fix, también introduce una variable ligada y su cuerpo (naturalmente) debe ser del tipo supuesto para la referida variable.

Finalmente, la regla del destructor de tipos abstractos opera en forma simétrica a la del constructor: si la constante de tipo involucrada está definida y el término al que se aplica el destructor tiene por tipo una instancia del tipo abstracto obtenida por una sustitución de las variables de tipo, entonces el término original tiene por tipo una instancia de la representación del tipo abstracto obtenida por idéntica sustitución.

Lo que resta es extender la asignación de tipos a sesiones:

SessType

Se definen secuentes de la forma

$$\rho \vdash s$$

donde $s \in$ SESS.
En lo que sigue $c \in$ CNST, $e \in$ E-TERM,
$A \in$ TYPE, $T \in$ TYPE-CNST,
$Tterm \in$ TYPE-TERM, $Xlist \in$ TYPE-VAR*
$ldef \in$ CNST-DEF*

[*Definición de Constante Global*]

$$\frac{\langle \gamma \cup \langle\langle c, \mathbf{fix}\ xc\ xce\rangle\rangle, \tau\rangle \vdash s}{\langle \gamma, \tau\rangle \vdash [\mathbf{define}\ c\ e\ ; s]}$$

$$(xc = \text{CTOV}\ (c, c),\ xce = \text{CTOV}\ (c, e))$$

[*Definición de Tipo Abstracto*]

$$\frac{\tau \overset{\text{Type}}{\vdash\!\!\!-\!\!\!-} \langle T, Xlist, Tterm\rangle \qquad \langle \gamma, \tau \cup \langle\langle T, \langle Xlist, Tterm\rangle\rangle\rangle \vdash ldef \circ s}{\langle \gamma, \tau\rangle \vdash [\mathbf{abstype}\ T\ (Xlist)\ Tterm\ \mathbf{with}\ ldef\ ; s]}$$

[*Término Evaluable*]

$$\frac{\langle \rho, \langle\rangle\rangle \overset{\text{TermType}}{\vdash\!\!\!-\!\!\!-} e : A \qquad \rho \vdash s}{\rho \vdash [e ; s]}$$

[*Sesión Vacía*]

$$\rho \vdash [\,]$$

■

Las reglas deben interpretarse como un mecanismo de verificación de la buena formación de los mensajes de las sesiones en relación al sistema de tipos. De hecho, la sesión s está "*bien tipada*" si puede probarse en SessType:

$$\langle\langle\rangle, \langle\rangle\rangle \vdash s$$

La regla para la sesión cuyo primer mensaje es una definición de constante global opera introduciendo la constante asociada al punto fijo del término evaluable correspondiente en el contexto de constantes, para proseguir con el resto. La regla para las definiciones de tipos abstractos es similar. Se verifica la buena formación del término de tipo con un sistema que impone tipos a tales términos, llamado **Type**. No se incluye la especificación de este sistema para ahorrar espacio. Aquí puede forzarse encapsulamiento sustituyendo la segunda premisa por:

$$\langle \gamma, \tau \cup \langle\langle T, \langle Xlist, Tterm\rangle\rangle\rangle \vdash ldef \qquad \langle \gamma, \tau\rangle \vdash s$$

haciendo que la definición de T sólo pueda usarse en el alcance de **with**. De hecho, dicha definición sólo se usa para asignar tipos a términos en que se apliquen el constructor o el destructor de T, lo cual, si hay encapsulamiento, sólo puede ocurrir en el alcance mencionado.

La regla para términos evaluables deriva la decisión sobre el tipado de la sesión a la asignación de algún tipo al término evaluable que es su primer mensaje. Finalmente, la sesión vacía está bien tipada.

3.2 Variables, constantes y polimorfismo.

La fuente original de polimorfismo del sistema está constituída por las reglas correspondientes a la abstracción funcional, aplicación y variable, es decir, las de asignación de tipos al cálculo λ puro (HinSel 86). Estas reglas imponen sobre las aplicaciones la restricción de que su operando sea del tipo del dominio del operador y se definen con abstracción de cualquier conjunto de constantes de tipos que se pueda introducir en el lenguaje, siendo por esto inherentemente polimórficas.

Para una expresión como:

$$\lambda x.\, x$$

más de un tipo puede ser asignado, en particular, cualquiera de la forma $A \rightarrow A$. En las derivaciones correspondientes a una tal asignación, la regla de la abstracción funcional introduce la hipótesis $x:A$ sobre el tipo de su argumento. El éxito de la derivación dependerá de la forma en que el cuerpo de la abstracción restrinja el tipo del argumento. En el ejemplo, el cuerpo es la propia variable. La regla de la variable no impone ninguna restricción a su propio tipo: puede probarse que una variable tiene cualquier tipo que le haya sido asociado en una hipótesis. Esto muestra que cualquier deducción de la forma indicada será exitosa y también ilustra la naturaleza polimórfica de la regla de la abstracción funcional.

En el lenguaje presentado se extiende el cálculo λ con construcciones adicionales para términos evaluables. Algunas de estas construcciones imponen restricciones al tipo de sus operandos, comportándose del mismo modo que la aplicación en relación a la abstracción funcional. Véase, por ejemplo:

λx. **pair** (x) (**+** x (**num** 1))

La primera ocurrencia de x en el cuerpo de la abstracción no recibe restricciones sobre su tipo, puesto que **pair** es polimórfico en sus argumentos. Sin embargo la presencia de x como argumento de **+** restringe su tipo a **number**. Como consecuencia, una derivación para el término completo sólo tendrá éxito si se hace la hipótesis x:**number**, resultando en la única asignación de tipo posible para el término:

λx. **pair** (x) (**+** x (**num** 1)) : **number** × **number**

Esto muestra que el tipo de cualquier ocurrencia de una variable en el cuerpo de la abstracción que la introduce es restringido al tipo de la ocurrencia de tipo menos general. Por supuesto, el razonamiento hecho para las variables de las abstracciones funcionales es válido para cualquier construcción que introduzca variables ligadas.

En principio, puede pensarse que el comportamiento de las constantes en relación a los tipos admite una caracterización análoga a las de las variables. La idea proviene de que la semántica clásica de las declaraciones locales **local** *x e* **in** *d* es usualmente dada como equivalente a la de (λx.*d e*), lo cual podría extenderse de modo similar a las declaraciones globales.

Sin embargo, debe considerarse que las constantes son nombres para términos complejos y eventualmente polimórficos. Si el tratamiento en la asignación de tipos a constantes fuera idéntico al de las variables, distintas ocurrencias de la constante x en el cuerpo *d* de la declaración local serían todas restringidas a un mismo tipo. Considérese:

local id λx.x **in** **if** (id **true**) (id **num** 1) 0

La constante id representa un término polimórfico. Tiene sentido, entonces, que las dos ocurrencias de id en el cuerpo de la declaración puedan ser restringidas a tipos diferentes, en particular: **bool** → **bool** y **number** → **number**.

PEDECIBA
INFORMATICA

Si se tratara la asignación de tipos de la definición de constante como en la aplicación correspondiente:

(λid.if (id true) (id num 1) 0 λx.x)

se tendría un término al que no puede asignarse tipo, pues no existiría ninguna hipótesis sobre el tipo de id compatible con los de sus dos ocurrencias en el cuerpo de la abstracción.

Esto justifica la distinción entre constantes y variables como formas sintácticas y del punto de vista de la asignación de tipos. Las reglas de definiciones locales y globales añaden al contexto la constante introducida asociada al término que ella representa. Luego, la regla de la constante convierte la derivación de un tipo cualquiera para ella en la derivación del mismo tipo para su término asociado. Si el término es polimórfico, distintos tipos para distintas ocurrencias de la constante podrán ser deducidos.

4. Desarrollos complementarios y conclusiones.

4.1 Propiedades del sistema de tipos.

La culminación natural del desarrollo es la demostración de que el sistema de tipos propuesto es sólido (en el sentido de que la evaluación de una sesión bien tipada no puede ser fallida) y la proposición de un algoritmo de inferencia que implemente las reglas del sistema. En relación al primer punto, según lo discutido en 3.1 y 2.3, es suficiente probar que si puede asignarse un tipo a un término evaluable, entonces su evaluación no es fallida.

Esto obliga a caracterizar las demostraciones fallidas de secuentes $\rho \vdash e \Longrightarrow v$ en TermEval. Tal caracterización es engorrosa y conduce a una prueba cuya complejidad es análoga a la de describir un interpretador para el lenguaje, basado en la semantica dada. En contrapartida, la verificación puede hacerse de una forma bastante directa si la semántica se da en la forma de un cálculo de reducciones, extendiendo la de [Holm 89].

Obviamente, el sistema de tipos no es completo, e.e. existen términos evaluables con valor a los que no puede asignarse tipo. Si el sistema fuera completo podría decidirse si un término evaluable tiene o no una evaluación fallida. En el sistema presentado pueden darse casos como:

if true (num 1) (+ true 0)

donde el tercer operando no puede recibir tipo y, por lo tanto, tampoco el término completo, pero éste puede ser evaluado a num 1.

Finalmente, el sistema de tipos es decidible, lo cual es obvio a partir de la forma de sus reglas. Pueden especificarse, entonces, algoritmos de inferencia que implementen este sistema. En [Holm 89] y [CLéDDK 86] se discute este problema para casos más simples que el aquí presentado. El primero presenta un algoritmo en estilo funcional y el segundo prueba que el sistema puede interpretarse como un programa Prolog. Extender estos resultados para el caso de este trabajo no es difícil aunque sí laborioso.

4.2 Conclusiones.

Se ha definido un sistema de tipos polimórfico para un lenguaje funcional, en presencia de definiciones de constantes globales y locales y de tipos abstractos. Esto extiende los resultados de [Milner 78] y [Holm 89] y formaliza construcciones similares a las de otros lenguajes funcionales.

El uso de sistemas de deducciones para especificar la semántica del lenguaje provee un tratamiento nítido de los contextos de constantes y tipos de los que dependen las evaluaciones de las expresiones. En este aspecto, el planteo es preferible al de semántica de reducciones. Sin embargo, con este último enfoque la propiedad de solidez del sistema de tipos puede probarse fácilmente.

Un algoritmo de inferencia de tipos para este lenguaje ha sido implementado usando el lenguaje Scheme. Otras implementaciones en Prolog son realizables con facilidad a partir de la especificación dada.

Referencias.

[CarWeg 85] **On Understanding Types, Data Abstraction and Polymorphism.** Cardelli L., Wegner P. *ACM Computing Surveys Vol 17, No 4, December 1985.*

[ClémDDK 86] **A Simple Applicative Language: Mini-ML.** Clément D., Despeyroux J., Despeyroux T., Kahn G. *INRIA - Rapport de Recherche 529, Mai 1986.*

[CouHP 85] **The ML Handbook.** Cousineau G., Huet G., Paulson L. *En Méthodes et Langages de l'Intelligence Artificielle. INRIA, 1985.*

[HinSel 86] **An introduction to combinators and the lambda-calculus.** Hindley J., Seldin J. *Cambridge University Press, 1986.*

[Holm 89] **Polymorphic Type Systems - A Proof Theoretic Approach.** Holmström S. *Programming Methodology Group, University of Goteborg, Report 6, September 1983.*

[Kahn 87] **Natural Semantics.** Kahn G. *INRIA - Rapport de Recherche 601, Février 1987.*

[Milner 78] **A Theory of Type Polymorphism in Programming.** Milner R. *Journal of Computer and System Sciences, Vol. 17, No. 3, 1978.*

[M-Löf 82] **Constructive Mathematics and Computer Programming.** Martin-Löf P. *In Proceedings of the 6th. International Congress for Logic, Methodology and Philosophy of Science, Hannover 1979, North Holland 1982.*

[NorPet 89] **Types and Specifications.** Nordström B., Peterson K. *Programming Methodology Group, University of Goteborg, Report 10, September 1983.*

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

UN SISTEMA DE TIPOS PARA UN LENGUAJE FUNCIONAL

DAIQUI SYLVIA
GASPES VERONICA
KESNER DELIA
LAVATELLI CAROLINA

ESLAI
Escuela Superior Latinoamericana de Informática

1 - RESUMEN

Se define un lenguaje de programacion funcional basado en el calculo lambda con un sistema de tipos con polimorfismo. El lenguaje es fuertemente tipado y la asignacion de tipos a sus expresiones se resuelve por inferencia a partir de las declaraciones y el contexto.

Los fundamentos teoricos del trabajo se encuentran en "A theory of type polymorphism in programming ", Milner 77.

Se implemento un precompilador que consiste en un analizador sintactico, un modulo de chequeo e inferencia de tipos y un generador de codigo Lisp puro.

2 - INTRODUCCION

2.1 - ACERCA DE LOS SISTEMAS DE TIPOS EN PROGRAMACION

La nocion de tipo surge informalmente en cualquier dominio. Categorizar objetos de acuerdo a su uso y comportamiento, da lugar a sistemas de tipos mas o menos bien definidos, de modo que los universos no tipados pueden pensarse como tipados.

En matematica y computacion, los tipos imponen restricciones sobre la interaccion entre objetos que previenen que estos interactuen en forma inconsistente con otros objetos.

Las siguientes son algunas caracteristicas que pueden tener los sistemas de tipos de los lenguages de programacion.

Se dice que un lenguage con un sistema de tipos es **estatico**, cuando el tipo de todas las expresiones puede determinarse mediante un analisis del texto. Esto hace que no haya que realizar ningun chequeo de tipos durante la ejecucion del programa. El requerimiento de que los tipos de todas las variables y expresiones este determinado estaticamente es muy restrictivo, y limita el poder de expresion al restringir prematuramente el comportamiento de los objetos.

Estos sistemas excluyen procedimientos y funciones genericos.

Se dice que un lenguage con un sistema de tipos es **fuertemente tipado** si mediante un analisis del texto puede decirse si las expresiones son consistentes, a pesar de que el tipo de las expresiones puede no conocerse. Esta propiedad garantiza las mismas cosas que la anterior y es menos restrictiva.

Dos lenguages que adoptan esta filosofia son ML [ML 85] y Miranda [Tur 86].

A continuacion se intenta caracterizar la idea de polimorfismo y las distintas formas en que se puede presentar.

Una funcion es **monomorfica** si sus argumentos deben tener un unico tipo. Por el contrario, si se permite que una funcion se aplique a valores de distintos tipos esta es **polimorfica**. Por extension si un lenguaje soporta funciones polimorficas es polimorfico.

En principio se podria hablar de dos tipos de polimorfismo :

1- aparente : cuando una funcion se aplica (o parece que se aplica) a argumentos de distintos tipos puede comportarse en forma distinta para cada caso . En general puede pensarse que esa funcion es en realidad un conjunto finito de funciones monomorficas , que ejecuta un codigo diferente para cada tipo. Casi todos los lenguages presentan este tipo de polimorfismo en la forma de sobrecarga de operadores y coercion . Un ejemplo es el operador " + " de PASCAL.

2- universal : cuando una funcion se aplica normalmente sobre un numero infinito de tipos todos de estructura similar. Se puede afirmar que estas funciones realmente tienen muchos tipos , y que se ejecuta el mismo codigo para argumentos de cualquier tipo admisible. Puede resultar util pensar en este tipo de polimorfismo de alguna de las dos formas siguientes, sin que esto imponga una clasificacion

_ parametrico : toda funcion polimorfica tiene parametros (explicitos o implicitos), que determinan el tipo del argumento para cada aplicacion de la funcion (ML fue construido enteramente alrededor de este estilo).

_ por inclusion : un objeto puede verse como perteneciente a muchas clases diferentes no necesariamente disjuntas.

2.2 - INFERENCIA DE TIPOS EN LENGUAJES FUNCIONALES

En este apartado se introduce la metodologia de Milner, mediante algunos ejemplos. El objetivo es conseguir un sistema de tipos flexible y seguro.

Por seguro se entiende que en un programa bien tipado es imposible por ejemplo tratar un entero como una lista, o tratar a true como una funcion. De cualquier forma, el sistema de tipos no puede proteger de todos los errores de tiempo de ejecucion; por ejemplo tomar el primer elemento de una lista vacia o dividir por cero.

La flexibilidad se consigue usando polimorfismo.

El fundamento teorico que garantiza la existencia de un algoritmo capaz de asignar tipos de forma que se consiga la seguridad mencionada incluye los siguientes teoremas:

1-Solidez semantica: las expresiones bien tipadas no pueden andar mal. Esto es, si se las evalua con una asignacion de valores que respeta el contexto entonces el resultado tiene el tipo previsto.

2-Solidez sintactica: si el algoritmo propuesto termina asignando un tipo a una expresion, esta es bien tipada.

Algunas caracteristicas importantes del trabajo de Milner son:

- Todo lo relacionado con tipos es resuelto estaticamente, de modo que pueda ejecutarse el programa sin controlar los tipos.

- El tipo de los terminos puede ser inferido del contexto. No es obligacion del programador declarar el tipo de todos los terminos de su programa. En caso que lo haga se verifica que sea correcto.

- El polimorfismo juega un rol muy importante. El polimorfismo presente en un programa se considera una extension natural de los operadores polimorficos primitivos presentes en todos los lenguages de programacion (aplicacion funcional,formacion de pares ordenados, operadores de procesamiento de listas,etc.)

Los ejemplos se describen en ML ya que este es un lenguaje que sigue la metodologia de tipos propuesta.

Los tipos seran denotados con las letras R, S, T, ...

Se hara distincion entre:

- MONOTIPOS: tipos construidos a partir de un conjunto de tipos basicos (integer, boolean, etc.) por los operadores

  \# (producto cartesiano)

  \+ (union de tipos)

  ---> (tipo de una funcion)

  LIST (lista de objetos de un mismo tipo)

- POLITIPOS: tipos obtenidos admitiendo variables que denotan tipos (VT) en su definicion. A las VT se las representara por A, B,C, ...

EJEMPLO 1: Inferencia del tipo de la funcion map.

```
letrec MAP(f,m) = if null(m) then nil
                  else cons(f(hd(m)),MAP(f,tl(m)))
```

El contexto esta dado por el tipo de las primitivas de manejo de listas

- null : A LIST ---> boolean (dice si una lista es vacia)
- nil : A LIST (lista vacia)
- hd : A LIST ---> A (selector del primer elemento de la lista)
- tl : A LIST ---> A LIST (selector de la cola de la lista)
- cons : (A # A LIST) ---> A LIST (agrega un elemento al comienzo de la lista)

Estos operadores son polimorficos , ya que en sus tipos aparece por lo menos una VT.

Intuitivamente se vera que tipo asignar a MAP.

La definicion de MAP menciona hd(m) y tl(m) de lo que se deduce que m debe ser una lista de algo, es decir,

m : B LIST

y ademas menciona que f se aplica a un elemento de m, de lo que se deduce que f debe ser una funcion, es decir,

f : C ---> D

El resultado de MAP es nil, o se obtiene usando el operador cons, de lo que se deduce que es una lista.

Ahora bien, f se aplica a elementos de m, por lo tanto el dominio de f debe ser del mismo tipo que los elementos de m, con esto se puede decir que el dominio de MAP es

(B ---> D) # B LIST

-------- --------

^el tipo de f ^el tipo de m

Los elementos de la lista que MAP da como resultado, se obtienen aplicando f a elementos de m, por lo tanto tendran el tipo del rango de f, y entonces el tipo de MAP seria:

MAP : ((B ---> D) # B LIST) ---> D LIST

Todo esto se podria formalizar un poco. Se denota con T(id) al tipo del identificador id. De la definicion de Map y del contexto se pueden plantear las siguientes ecuaciones:

T(MAP) = T(f) # T(m) ---> R1

T(null) = T(m) ---> boolean

T(hd) = T(m) ---> R2

T(tl) = T(m) ---> R3

T(f) = R2 ---> R4

T(MAP) = T(f) # R3 ---> R5

T(cons) = R4 # R5 ---> R6

T(nil) = R1 = R6

Cada una de estas ecuaciones surge de algun subtermino de la definicion de MAP, salvo la ultima que surge de que las dos subexpresiones de un condicional tienen el mismo tipo y de que definiens y definiendum deben tener el mismo tipo . El tipo que se asigne a MAP debera satisfacer estas ecuaciones y las del contexto , y puede obtenerse usando el algoritmo de unificacion de Robinson [ROB 65] .

La aparicion de variables en el tipo de un identificador hace que cada ocurrencia del identificador sea tipada con una instancia de substitucion de este tipo generico (sustituyendo las VT por tipos). El tipo asignado al identificador en sus distintas ocurrencias no tiene por que ser el mismo. Un ejemplo de esto podria ser el siguiente:

Si se supone que string es un tipo basico, estructurado como lista de caracteres y que se dispone de la funcion

long : string ---> int

Para el tipo int se dispone de la funcion

sqrt : int ---> real

Entonces, si para tipar el termino

MAP(sqrt,MAP(long,cad:string LIST))

las dos ocurrencias de MAP tendran los tipos

((string --->int) # string LIST) ---> int LIST (la ocurrencia interna)
((int ---> real) # int LIST) ---> real LIST (la ocurrencia externa)

Esto tiene sentido siempre que no se trate de las distintas ocurrencias de un parametro formal o las del identificador que se esta definiendo recursivamente, ya que en estos casos se desea que todas tengan el mismo tipo.

EJEMPLO 2: Interaccion entre el let local y el lambda (una definicion util: variables genericas).

Se desea construir una funcion que produzca cosas de la forma

(b , c) |---> ((a , b) , (a , c))

Esto se puede describir en terminos de las dos funciones siguientes (a pesar de que puede ser descripta mucho mas facilmente de otro modo , pero que no mostraria lo que se pretende)

SUSTI tal que

SUSTI (f , g) (a , b) = (f(a) , g(b))

PAR tal que

PAR(a) (b) = (a , b)

con estas dos, la funcion que se pretende definir puede describirse :

GRAN_PAR ::= lmb x . lmb y z . let tag = PAR(x) in SUSTI(tag , tag)

Los tipos de SUSTI y PAR y el tipo esperado para GRAN_PAR son :

T(SUSTI) = (A ---> B) # (C ---> D) ---> ((A # C) ---> (B # D))

T(PAR) = A ---> (B ---> (A # B))

T(GRAN_PAR) = A' ---> [(B' # C') ---> ((A' # B') # (A' # C'))]

Si en GRAN_PAR se le da a x el tipo A', el tipo local de PAR(x) es D' ---> A' # D' y si se permite que tanto A' como D' sean instanciados en forma diferente en las dos apariciones ligadas de tag, se tendra para GRAN_PAR un tipo mas general que el esperado. El problema se debe a que tanto tag como su tipo generico dependen de la variable x ligada por un lambda (decimos que x esta lmb_ligada) y de su tipo , y no se permitira que distintas ocurrencias de una variable lmb_ligada tengan tipos diferentes. La forma de hacer esto es sencilla ya que se esta hablando del argumento de la funcion; se decreta que al instanciar el tipo de las variables ligadas por let solamente se podra instanciar las variables tipo que no aparezcan en el tipo de variables lmb_ligadas por lambdas que encierren al let . A estas variables instanciables se las llama genericas , y son las que dan el polimorfismo local del identificador. En el ejemplo anterior D' es generica , pero A' no .

La regla que se acaba de introducir hace que dos esquemas de expresiones lambda cuya semantica es la misma esten sometidas a reglas de tipado diferentes . El caso es el siguiente :

- los esquemas

(lmb x.e') e

let x = e in e'

tienen ambas la misma semantica: e' [x /e] . Sin embargo cuando se intenta tipar los siguientes casos particulares de estos esquemas, pueden resultar tipos diferentes:

- las expresiones

let PEPE = lmb x.x in PEPE(PEPE)

(lmb PEPE . PEPE(PEPE)) (lmb x.x)

son casos de los esquemas anteriores y la semantica de ambas es (lmb x.x) (lmb x.x) . En la primera expresion, PEPE no es argumento de una funcion ni es un identificador definido recursivamente, por lo tanto sus dos ocurrencias pueden tiparse con instancias distintas de algun tipo mas general.

En la segunda expresion, PEPE es el argumento de la definicion de una funcion por lo cual las dos ocurrencias deben instanciarse igual. Es por esto que la expresion no puede tiparse pues unifican los tipos de las dos ocurrencias de PEPE.

Poder resolver la interaccion entre variables ligadas por lmb y por let estaticamente es de fundamental importancia : permite tratar la declaracion no global de una funcion que contiene como variable libre un parametro formal de una funcion que la engloba, sin postergar los chequeos .

3 - DEFINICION DEL LENGUAJE

Teniendo en cuenta el objetivo del trabajo, no era esencial dar un conjunto muy amplio de primitivas, sino uno suficiente para poder especificar funcionalmente y permitir declaraciones de tipos.

Una primera aproximacion fue el lenguaje EXP [Mil 77] .

<EXP> ::=X (X denota un identificador valido)
(<EXP> <EXP>)
if <EXP> then <EXP> else <EXP>
lambda X . <EXP>
let X = <EXP> in <EXP>

Esto bastaba para incluir el calculo lambda, pero era muy limitado en su poder de expresion, no tenia ninguna primitiva para el manejo de listas y no permitia DECLARAR tipos explicitamente.

Por ello se definio un lenguaje con sintaxis similar a la de LISP [LSP 84] (que incluye al lenguaje EXP) y con una estructura de tipos que es un subconjunto de la de ML [ML 85]

Este nuevo lenguaje provee las siguientes primitivas:

- operadores aritmeticos:

+, *, /, -

- operadores logicos:

and, not, or

- operadores relacionales:

less, greater, eqnum (operador numerico), eqchar (operador sobre caracteres)

- operadores sobre listas:

cons, null, car, cdr, atom

- operador condicional :

if

todos con la semantica usual.

- operadores sobre producto cartesiano prim, seg, par, donde

prim ([a , b]) = a

seg ([a , b]) = b

par (a , b) = [a , b]

con a y b expresiones validas del lenguaje.

- let con evaluacion simultanea.

- lmb permite definir una funcion anonima, con cantidad arbitraria de parametros. La instanciacion de unode ellos define una funcion con un argumento menos y el argumento es sustituido por la instancia en toda la definicion de la funcion. Por ejemplo, lmb (x y) x+y (no es la sintaxis correcta) instanciado en 8 es la funcion lmb (y) 8+y Se puede instanciar mas de un argumento por vez.

Nota: Solo se proveen primitivas para comparar enteros y caracteres La funcion para comparar objetos de cualquier otro tipo debera ser implementada por el programador.

Con respecto a los tipos, estos podran ser monotipos o politipos donde:

- number, boolean y char son monotipos.
- si A y B son monotipos entonces A > B, A # B y list A tambien son monotipos.
- $x , con x identificador valido, es un politipo (una variable tipo).
- si A es un politipo, list A lo es.
- si A o B son politipos, A > B y A # B lo son.
- si A es tipo y @iden = A entonces @iden es tipo. (abreviatura de un tipo)

Un programa en este lenguaje constara de una declaracion de tipos opcional y de un cuerpo en que se definen funciones. Se da la opcion al programador de que asigne tipo a los argumentos de las funciones, a los de las expresiones lambda y let , y al resultado de las funciones.

3.1 - GRAMATICA

<PROGRAMA> ::= <DECLARACION> <ESPECIFICACION>

<DECLARACION> ::= μ |
type <LISTA_DECLARACION> ;

<LISTA_DECLARACION> ::= <IDEN_TIPO>= <TIPO> |
<IDEN_TIPO>= <TIPO> , <LISTA_DECLARACION>

<TIPO> ::= <TIPO_BASICO> |
<IDEN_TIPO>|
(<TIPO> > <TIPO>)|
(<TIPO> # <TIPO>)|
(list (<TIPO>))

<ESPECIFICACION> ::= <DEF>|
<DEF> <ESPECIFICACION>

<DEF> ::= (defun iden (<ARG_LISTA>) <EXP>) <TIPO_FUN>

<TIPO_FUN> ::= μ |
: <TIPO>

<ARG_LISTA> ::= μ|
iden |
iden : <TIPO> |
iden <ARG_LISTA> |
iden : <TIPO> <ARG_LISTA>

```
<EXP> ::= (if <EXP> <EXP> <EXP>) |          condicional

          (+ <EXP> <EXP> ) |                operadores aritmeticos
          (- <EXP> <EXP> ) |
          (* <EXP> <EXP> ) |
          (/ <EXP> <EXP> ) |

          (eqnum <EXP> <EXP>) |             operadores relacionales
          (eqchar <EXP> <EXP>) |
          (less <EXP> <EXP>) |
          (greater <EXP> <EXP>) |

          (and <EXP> <EXP> ) |              operadores logicos
          (or <EXP> <EXP> ) |
          (not <EXP> ) |

          (atom <EXP>) |                    operadores sobre listas
          (cdr <EXP> ) |
          (car <EXP> ) |
          (null <EXP> ) |
          (cons <EXP> <EXP> ) |

          (prim <EXP> ) |                   operadores sobre producto cartesiano
          (seg <EXP> ) |
          (par <EXP> <EXP> )|

          (let (<LETLIS>) <EXP> ) |

          (iden <LISTA_EXP>) |

          (lambda (<ARG_LAMBDA>) <EXP>)

          '<EXP> |
```

```
        true |                                  constantes
        false|
        num |
        nil |
        iden

<LISTA_EXP>::= <EXP>
               <EXP> <LISTA_EXP>

<ARG_LAMBDA> ::= iden  <COLA_ARG_LISTA>

<LETLIS> ::=  <PAR> |
              <PAR> <LETLIS>

<PAR> ::= ( iden  <EXP> )
```

4 - CHEQUEO E INFERENCIA DE TIPOS

El chequeo e inferencia de tipos debe hacerse sobre un programa sintacticamente correcto. El cuerpo de un programa en este lenguaje puede verse como una lista de definiciones de funciones. De esta forma lo trata el algoritmo de chequeo e inferencia.
Lo que hace dicho algoritmo es determinar el tipo de cada una de las funciones de la lista, ya sea , mediante inferencia y/o respetando la asignacion de tipo del programador y chequeando los tipos para las funciones propias del lenguaje.

El tipo de una expresion se obtiene a partir del tipo de sus subexpresiones. El tipo que se obtiene es el llamado " tipo mas general ", T, tal que para cualquier tipo H que pueda tener esa expresion existe una sustitucion S de variables tipo que cumple S.T=H (H=<T) [Hindley] [Milner].
Finalmente, lo que calcula el algoritmo es una lista de la forma

((nombre-funcion-1 tipo1) . . . (nombre-funcion-k tipok))

Para tipar cada una de las funciones de la lista , introduciremos primero los siguientes conceptos:

Un prefijo tipado P es una secuencia finita cuyos miembros son de la forma 'let x :m' o 'lmb x :m ', donde x es una variable y m es un tipo.
Una expresion prefijada tipada tiene la forma P | E , **donde toda variable libre de la** expresion E aparece en al menos un miembro del prefijo tipado P.

Para la gramatica dada anteriormente las subexpresiones prefijadas tipadas (sub-pes) quedan definidas del siguiente modo:

1- P | x [: tipo] no tiene mas sub-pes que ella misma.

2- P | (e e') tiene sub-pes P | e y P | e'.

3- P | (if b then a else c) tiene sub-pes P | b , P | a , P | c .

4- P | (lmb x1 [:t1] ...xk[:tk] . e) tiene sub-pe P.lmb x1:m1 ... lmb xk :mk | e .

5- P | (let x1 [:t1] = e1 ... xk[:tk] = ek . e) tiene sub-pes P | (e1, ... , ek) y P :-let x1:m1 ...-let xk:mk | e.

6- P | (e1, ... ,ek) tiene sub-pes P | e1 a P | ek.

7- P | (defun F (x1 [:t1], ... , xk[:tk]) . e) [: tipo] tiene sub-pe P | (lmb x1[: t1]... xk[:tk] . e) .

donde m es t si x esta declarada de la forma x : t , y una variable generica si no.
Por ejemplo,

(1) lmb y : number | let f= lmb x . (x y) in (f y)

tiene como sub-pes tipadas

(2) lmb y : number | lmb x . (x y)
(3) lmb y : number . let f : $A | (f y)

donde (2) tiene como sub-pes tipadas a

(4) lmb y : number . lmb x : $B | (x y) ,que a su vez tiene como sub-pes tipadas a

(5) lmb y :number . lmb x : $B | x

(6) lmb y :number . lmb x : $B | y

y donde (3) tiene como sub-pes tipadas a

(7) lmb y : number . let f : $A | f

(8) lmb y : number . let f : $A | y

4.1 - ALGORITMO DE CHEQUEO E INFERENCIA

La forma general de determinar el tipo de una expresion es:
Dada una expresion E a tipar y un prefijo tipado P, inicialmente vacio, lo que se hace es:

* determinar las sub-pes de P|E
* tiparlas
* segun la forma de E y con los tipos de las subexpresiones construir el tipo de E.

Lo que hace el algoritmo que asigna tipos a las funciones de un programa es:
Tipar cada una de las funciones (defun ...), y mantener una lista que contiene el nombre de cada funcion y el tipo correspondiente (el mas general).

Ya que un programa es sintacticamente correcto aunque haya aplicaciones que involucren funciones aun no definidas (el analizador sintactico garantiza que su correspondiente definicion este mas adelante en el texto del programa), con el siguiente ejemplo se puede ver lo que el algoritmo debe hacer.
Sea F la funcion a tipar, cuya definicion es

(defun F (x) (H x))

El calculo del tipo de esta expresion, dara el tipo mas general de F.
Durante el calculo deben considerarse dos casos,

1- Que la funcion H ya este definida , es decir que se conozca su tipo mas general.
Sea r >= q el tipo mas general de H, con r y q politipos o monotipos .
En este caso debe chequearse que el tipo de x sea "menor o igual" que r .
Si es asi, la aplicacion (H x) es correcta. Si no, se levanta un error y el programa de inferencia termina .

2- Que no se conozca aun el tipo mas general de H.
En cuyo caso se infiere el tipo del argumento x, t_x, y se mantiene $t_x > B$ (con B variable generica nueva) junto con los tipos (provisorios) con que fue usada la funcion H . En la determinacion del tipo de F se usa B como tipo de la aplicacion (H x).

Despues de obtener el tipo de F hay dos nuevos casos a considerar:

1- Que F haya aparecido antes en el programa (es decir en alguna otra funcion ocurrio la aplicacion de F a una expresion).
En este caso, cada vez que se la uso se le dio un tipo provisorio que no necesariamente era el mas general, y ahora es el momento de chequear que cada uno de esos sea menor o igual que el mas general (ya obtenido). Si esto no ocurrio en algun caso, es decir, en ese caso se aplico F a argumentos con tipos incompatibles con los inferidos de su definicion, el programa tal como esta no admite asignacion de tipo. Se levanta un error y el programa de inferencia termina.
Si por el contrario se pudo unificar cada tipo provisorio con el mas general, entonces la funcion F se uso correctamente en todos los casos hasta el momento tratados y el programa de inferencia continua con las demas funciones.

2- Que F no haya aparecido antes en el programa . En tal caso el tipo mas general inferido se asocia a F en la lista resultado, para luego comparar con el los tipos que se obtengan al tipar las aplicaciones de F.

El tipado de las expresiones y los chequeos de consistencia mencionados se resuelven basicamente con un algoritmo de unificacion llamado Unify . Este algoritmo tiene como argumentos dos expresiones y calcula la sustitucion que debera aplicarse para unificar dichas expresiones.

Una especificacion del algoritmo seria :

A(P , E) = T, donde

1- si E es x entonces

si lmb x:m pertenece a P , T=m

si let x :m pertenece a P , T=[Bi \ Ci] S(m) , donde Ci son las variables genericas de S(m), Bi variables genericas nuevas y S la sustitucion del ambiente.

si no T= B (variable generica nueva)

2- si E es (d e) , entonces

a1=A (P , d) ;

a2 = A(sust(P) , e), sust es la sustitucion del ambiente.

Si d tiene tipo definido t , K=Unify(((a2 > a1), t)

T= tipo que resulta de aplicar la sustitucion del ambiente a a1

Si no, agregar (a2 > a1) a los tipos posibles de d , y T=a1 .

3- si E es (if b then e1 else e2) , entonces

a1=A (P , b) ; z=Unify(a1,bool) ;

a2=A (P , e1) ; a3=A (P, e2) ;

T=Unify(a2,a3) ;

4- si E es (lmb x1[:t1] ...xk[:tk] . e) , entonces

a1=A(P : lmb x1:m1...lmb xk:mk , e) ;

n = sustitucion aplicada a (m1 # m2 # .... #mk)

T = n > a1

5- si E es (let x1[:t1]=e1 ... xk[:tk]=ek . e) , entonces

a1=A(P , (e1,...,ek)) ;

(n1,...,nk)=Unify(a1,(m1,...,mk)) , donde mi es ti si esta declarado y variable generica si no

T=A(P. let x1:n1...let xk:nk , e)

6- si E es (e1,..,ek) entonces

T=(A(P,e1) , A(P, (e2,...,ek)))

7- si E es (defun F (x1[:t1] ... xk[:tk]) . e) [:tipo-F] entonces

a1=A(P, (lmb x1[:t1] ... xk[:tk] . e));

T= Unify(a1, (B > m)) , donde m es tipo-F si esta declarado, y C (variable generica nueva) si no.

5 - CONCLUSIONES Y TEMAS ABIERTOS

Uno de los objetivos del trabajo era definir un lenguaje funcional con un sistema de tipos. Nuestro lenguaje es bastante sencillo y de limitado poder de expresion, pero esta es una primera aproximacion que constituye el nucleo de futuras extensiones, tanto del sistema de tipos como del conjunto de primitivas, que podrian resultar en lenguajes aptos para programar.

Con una extension del sistema de tipos se podria permitir que se definan nuevos tipos mediante constructores introducidos por el programador. Esta extension requiere de teoria adicional que permita decir en que condiciones se pueden garantizar programas bien tipados

Un tema interesante (abierto) es analizar que parte de esta teoria puede ser util para desarrollar un sistema de tipos con inferencia para algun otro paradigma de programacion.

Ambas propuestas estan siendo desarrolladas en la ESLAI como trabajos de pasantia.

6 - BIBLIOGRAFIA

[Hin 86] Hindley, J. Roger, Seldin, J.P. "An introduction to Combinators and the Lambda Calculus". Cambridge University Press. 1986.

[Lsp 84] Mc Carthy, Abrahamd, Edwards, Hart Levin. Lisp 1.5 Programer's manual. MIT Press.

[Ml 85] Instituto National de Recherche en Informatique et en Automatique. ML Handbook.

[Mil 77] Milner, R. "A theory of Type Polimorfism in Programing " Departament of computer Sciense. University of Edimburg.

[Rob 65] J.A.Robinson, "A machine-oriented logic based en the resolution principle".JACM 12, 1965.

[Tur 86] Turner, D. "An Overview of Miranda". Sigplan Notices, Dic 86.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Implementación de una Máquina de Multi-Combinadores Categóricos

Martin A.Musicante* & Rafael D.Lins†
Dept. Informática - Univ. Federal de Pernambuco - Recife - Brasil
ESLAI - Parque Pereyra Iraola - Berasategui - Bs.As. - Argentina.

Resumen

Los Multi-Combinadores Categóricos forman un sistema de reescritura desarrollado para proveer implementaciones eficientes de lenguajes funcionales con evaluación lazy. El núcleo del sistema contiene sólo dos reglas de reescritura, con muy baja complejidad de reconocimiento. El sistema realiza el equivalente a varias β-reducciones en un solo paso, evitando generar expresiones trivialmente reducibles.

Se presenta aqui la implementación de una máquina de reducción de grafos basada en este formalismo. Se dan también medidas de tiempo y espacio consumidos por la máquina para algunos casos de prueba, y se realizan comparaciones con otras implementaciones existentes.

El presente trabajo forma parte de la implementación de un intérprete de SASL usando Multi-Combinadores Categóricos.

Palabras clave: Multi-Combinadores Categóricos, lambda cálculo, programación funcional, evaluación lazy, reducción de grafos.

Abstract

Categorical Multi-Combinators form a rewriting system developed with the aim of providing efficient implementation of lazy functional languages. The core of the system consists of only two rewriting laws with a very low pattern-matching complexity. This system allows the equivalent of several β-reductions to be performed at once, and avoids the generation of trivially reducible sub-expressions.

Herein we present implementation details of a Categorical Multi-Combinator graph reduction machine. Performance figures of time complexity are also presented. These figures are compared with implementations using a different formalism.

This work is part of an implementation for SASL using Categorical Multi-Combinators.

Keywords: Categorical Multi-Combinators, lambda calculus, functional programming, lazy evaluation, graph reduction.

*ESLAI - CC 3193 - cp 1000 - Bs.As. - Argentina
†Dept.de Informática - U.F.PE. - 50.739 - Recife - PE - Brasil

Introducción

Los lenguajes de la programación funcional proveen una forma clara y fácil de especificación de soluciones a problemas computacionales. Ellos están siendo foco de intensa atención [15], y su aceptación, en general era postergada a causa de la ineficiencia de las implementaciones disponibles.

Esta situación esta siendo revertida. Existen hoy implementaciones de este tipo de lenguajes que compiten con los compiladores de los lenguajes tradicionales [6].

Podemos ver a un programa funcional como un conjunto de definiciones de funciones y otros objetos. La ejecución de un programa funcional consiste en la evaluación de una expresión; esto puede hacerse reescribiendo sucesivamente la expresión hasta que alcance una forma "imprimible" (o forma normal). Diremos que una expresión está en forma normal cuando ella no puede reescribirse con las reglas del sistema. Por ejemplo, si tenemos

```
fac n = n * fac(n-1)   , n>0          (*)
      = 1              , otherwise
```

entonces, la expresión

```
fac 2
```

será evaluada de la siguiente forma:

$$
\begin{aligned}
\texttt{fac2} &\rightarrow 2 * \mathit{fac}\ (2-1) \qquad \dagger \\
&\rightarrow 2 * \mathit{fac}\ 1 \\
&\rightarrow 2 * (1 * \mathit{fac}\ (1-1)) \\
&\rightarrow 2 * (1 * \mathit{fac}\ 0) \\
&\rightarrow 2 * (1 * 1) \\
&\rightarrow 2 * 1 \\
&\rightarrow 2
\end{aligned}
$$

La definición (*) se usó en la linea (†), substituyendo el valor 2 en la variable *n*. Este proceso de pasaje de parámetros es la mayor fuente de "overhead" en las implementaciones de lenguajes funcionales como sistemas de reescritura. En el método de compilación elaborado primeramente por Turner en [18] se evita este problema al substituir las variables del programa por una combinación aplicativa de funciones constantes llamadas

combinadores (En general, un combinador será una expresión funcional sin variables libres).

Turner usó un sistema de combinadores basados en la Lógica Combinatoria de Curry [1]. Puede obtenerse otra teoría formal de funciones basandose en la Teoría de Categorías. Este es el caso de los **Combinadores Categóricos**, que forman un sistema similar al de la lógica combinatoria. El sistema original es debido a Curien [4] y está inspirado en la equivalencia entre el λ-cálculo con tipos y las Categorías Cartesianas Cerradas (ésto último puede verse en Lambek [7] y Scott [16]).

En [3] se da una máquina de pila para la ejecución de Combinadores Categóricos; pero, como lo muestra Turner en [18], este tipo de máquinas resulta sumamente ineficiente para la implementación de lenguajes funcionales lazy.

El sistema llamado **Combinadores Categóricos Simplificados** fue desarrollado por Lins [8] para proveer implementaciones eficientes de lenguajes funcionales lazy. Es un sistema basado en el original de Curien, pero con las ventajas de obtener una relación lineal entre el tamaño de una λ-expresión y su equivalente en el sistema (en el original, esta relación es cuadrática), y de poseer pocas y muy simples reglas de reescritura de términos. Existen simulaciones [9] que muestran que una implementación basada en este sistema puede ser al menos, un orden de magnitud mas rápida que una basada en el original, y presenta complejidad en espacio y tiempo del mismo orden que los combinadores de Turner en [9].

En [9 y 10] son introducidas dos nuevas modificaciones al sistema, formando los llamados **Combinadores Categóricos Lineales**. Estas modificaciones reducen el conjunto de reglas de reescritura, incrementando la eficiencia en la evaluación al disminuir también el número de reescrituras necesarias para llevar un término a su forma normal. Estas reglas no alteran la complejidad del algoritmo de "pattern matching".

Multi-Combinadores Categóricos son una generalización de los Combinadores Categóricos Lineales. El núcleo del sistema consiste sólo de dos reglas de reescritura, con una muy baja complejidad de reconocimiento, y evitando la generación de sub-expresiones trivialmente reducibles.

Existen otros trabajos en el mismo sentido. Un ejemplo de ello son los **Supercombinadores** de Hughes [5]. En [11] se muestran las similitudes y diferencias entre estas máquinas.

Se presenta aquí la especificación e implementación de una máquina de Multi-Combinadores Categóricos, haciendo un muy sencillo estudio comparativo con otras implementaciones.

1 Multi-Combinadores Categóricos

Multi-Combinadores Categóricos forman un sistema de combinadores (formas funcionales sin variables libres) basados en la teoría de categorías.

Multi-Combinadores Categóricos utilizan la notación del λ-cálculo de DeBruijn para variables. En dicho λ-cálculo, cada variable está representada

por un número natural, que es la "cantidad de lambdas" entre dica variable y el lambda que la liga. Así, por ejemplo, el λ-término

$$\lambda x \lambda y \lambda z.xz$$

se traduce en el λ-cálculo de DeBruijn como

$$\lambda\lambda\lambda.20$$

Una forma de presentar el sistema, es dando su traducción desde el λ-cálculo. En Multi-Combinadores Categóricos, la aplicación de funciones se denota por yuxtaposición - tomándose esta operación como asociativa a izquierda.

El algoritmo de compilación [10] para traducir λ-expresiones hacia Multi-Combinadores Categóricos es el siguiente:

(T .1) $[\underbrace{\lambda x_i \dots \lambda x_j}_{n}.a] = L^{n-1}(R^{x_i,\dots,x_j}_{n-1,\dots,0}a)$

(T .2) $[a \dots b] = [a] \dots [b]$

(T .3) $[c] = c$, donde c es una constante

Donde $R^{x_i \dots x_j}_{n_i \dots n_j}$ es la función de *substitución* que reemplaza recursivamente las ocurrencias de cada variable x_i por el entero que la representará en el sistema. Esta función se define como sigue:

(T .4) $R^{x_i \dots x_j}_{n_i \dots n_j} \underbrace{\lambda x_k \dots \lambda x_l}_{m}.a = L^{m-1}(R^{x_i,\dots,x_j,x_k,\dots,x_l}_{n_i+m,\dots,n_j+m,m-1,\dots,0}a)$

(T .5) $R^{x_i \dots x_j}_{n_i \dots n_j}(a \dots b) = (R^{x_i \dots x_j}_{n_i \dots n_j}a) \dots (R^{x_i \dots x_j}_{n_i \dots n_j}b)$

(T .6) $R^{x_i \dots x_j}_{n_i \dots n_j}b = \begin{cases} b \text{ , si } b \text{ es una constante} \\ n_k \text{ , si } b = x_k \end{cases}$

Si al aplicar la regla T.6 a una variable b, ésta puede asociarse a más de un x_k, entonces se elije el mínimo n_k correspondiente para hacer la substitución. Esto tiene por objeto preservar la localidad en el "binding" de las variables (notemos que, en realidad, la función mantiene una pila, y a cada variable it b se le asocia la mínima distancia al tope).

Damos a continuación un ejemplo de traducción de una λ-expresión usando el algoritmo anterior:

$$\begin{aligned}
[\lambda x \lambda y (y \lambda z (zz))] &\overset{T.1}{=} (L^1(R^{x,y}_{1,0}\, y\ \lambda z(zz))) \\
&\overset{T.5}{=} (L^1((R^{x,y}_{1,0}\, y)\ (R^{x,y}_{1,0}\lambda z(zz)))) \\
&\overset{T.6}{=} (L^1(0\ (R^{x,y}_{1,0}\lambda z(zz))) \\
&\overset{T.4}{=} (L^1(0\ L^0(R^{x,y,z}_{2,1,0}(zz)))) \\
&\overset{T.5}{=} (L^1(0\ L^0((R^{x,y,z}_{2,1,0}z)\ (R^{x,y,z}_{2,1,0}z))) \\
&\overset{T.6}{=} (L^1(0\ L^0(0\ (R^{x,y,z}_{2,1,0}z))) \\
&\overset{T.6}{=} (L^1(0\ L^0(0\ 0)))
\end{aligned}$$

Con esto queda definida la sintaxis del núcleo del sistema. Ahora bien, es necesario proveerlo (con fines practicos) de estructuras de datos. Para ello, en [11] se propone la adición de dos nuevos combinadores: el combinador de Bloques de Datos (**D**) y los selectores (S^i); lo que implica la adición de una nueva regla de reescritura al núcleo del sistema, como se verá mas adelante.

Luego, deben agregarse las siguientes reglas para compilar listas y operadores sobre ellas

(1.1) $[\![[\,]]\!] \Rightarrow [\,]$

(1.2) $[\![Hd]\!] \Rightarrow S^0$

(1.3) $[\![Tl]\!] \Rightarrow S^1$

(1.4) $[\![a : b]\!] \Rightarrow D\ [\![a]\!]\ [\![b]\!]$

(1.5) $R^{x_i \cdots x_j}_{n_i \ldots n_j}[\,] \Rightarrow [\,]$

(1.6) $R^{x_i \cdots x_j}_{n_i \ldots n_j} Hd \Rightarrow S^0$

(1.7) $R^{x_i \cdots x_j}_{n_i \ldots n_j} Tl \Rightarrow S^1$

(1.8) $R^{x_i \cdots x_j}_{n_i \ldots n_j}(a : b) \Rightarrow D\ (R^{x_i \cdots x_j}_{n_i \ldots n_j} a)(R^{x_i \cdots x_j}_{n_i \ldots n_j} b)$

donde ":" denota el operador de construcción de listas y [ ] es la lista vacía. La compilación de operadores sobre listas se realiza reemplazando estos operadores por el codigo compilado de sus definiciones. Por ejemplo, para la función ++ de SASL, que concatena dos listas se tiene:

(1.9) $[\![a \ ++ \ b]\!] \Rightarrow concat\ a\ b$, *donde*

$$concat \equiv L^1((Cond\ ([\,] = 1)\ 0)\ (D\ (S^0 1)\ (concat\ (S^1 1)\ 0)))$$

Se da ahora un ejemplo de compilación de una función usando listas. Definimos una función que cambia el orden de los elementos de una lista.

```
ref n = if n=[ ] then [ ]
                 else (ref (Tl n)) : (Hd n);
```

O con la sintaxis de SASL

```
ref n = (n=[ ]) -> [ ]; (ref (Tl n)) : (Hd n)
```

Esta función se traduce a Multi-Combinadores Categoricos como sigue:

$$\begin{aligned}
& [\![ref\ n = (n = [\,]) \rightarrow [\,];\ (ref\ (Tl\ n)) : (Hd\ n)]\!] \\
\Rightarrow\ & ref \equiv L^0(R^n_0((n = [\,]) \rightarrow [\,];\ (ref\ (Tl\ n)) : (Hd\ n))) \\
\Rightarrow\ & ref \equiv L^0(Cond\ (R^n_0(n = [\,]))\ (R^n_0[\,])\ (R^n_0(ref\ (Tl\ n)) : (Hd\ n))) \\
\Rightarrow\ & ref \equiv L^0(Cond\ (= (R^n_0 n)\ (R^n_0[\,]))\ (R^n_0[\,])\ (R^n_0(ref\ (Tl\ n)) : (Hd\ n))) \\
\Rightarrow\ & ref \equiv L^0(Cond\ (= 0\ (R^n_0[\,]))\ (R^n_0[\,])\ (R^n_0(ref\ (Tl\ n)) : (Hd\ n)))
\end{aligned}$$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ (R_0^n[\,])\ (R_0^n(ref\ (Tl\ n)) : (Hd\ n)))$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (R_0^n(ref\ (Tl\ n)) : (Hd\ n)))$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (R_0^n(ref(Tl\ n)))(Hd\ n))))$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ ((R_0^n ref)\ (R_0^n((Tl\ n)))(R_0^n(Hd\ n))))$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (R_0^n(Tl\ n)))\ (R_0^n(Hd\ n))))$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ ((R_0^n Tl)\ (R_0^n n)))\ (R_0^n(Hd\ n))))$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (S^1\ (R_0^n n)))\ (R_0^n(Hd\ n))))$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (S^1\ 0))\ (R_0^n(Hd\ n))))$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (S^1\ 0))\ ((R_0^n Hd)\ (R_0^n n))))$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (S^1\ 0))\ (S^0\ (R_0^n n))))$

$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (S^1\ 0))\ (S^0\ 0))))$

El conjunto de reglas de reescritura del sistema (enriquecido con bloques de datos y operaciones aritméticas es el siguiente (para una explicación más detallada, el lector puede recurrir a [11]).

(M.1) $\circ(x_c x_0 x_1 x_2 \ldots x_n)\ (P\ y_m \ldots y_1 y_0) \Rightarrow (x_c x_0' x_1' \ldots x_n')$,

$$donde \begin{cases} x_i' & = x_i\ if\ x_i\ es\ una\ constante\ o\ de\ tipo\ L^a(b) \\ & = y_k\ if\ x_i\ es\ la\ variable\ k \\ & = \circ x_i\ (P\ y_m \ldots y_1 y_0), en\ otro\ caso \end{cases}$$

(M.2) $L^n(y)\ x_0 x_1 \cdots x_n x_{n+1} \cdots x_z \begin{cases} \Rightarrow y\ x_{n+1} \ldots x_z\ si\ y\ es\ una\ constante \\ \Rightarrow x_{n-y} x_{n+1} \ldots x_z\ si\ y\ is\ una\ variable \\ \Rightarrow (\circ y\ (P\ x_0 \cdots x_n)) x_{n+1} \cdots x_z, en\ otro\ caso \end{cases}$

(S) $S^n(D\ x_i \ldots x_1 x_0) \Rightarrow x_n$

(+) $+xy \Rightarrow x + y$

(=) $Cond\ x\ m\ n \begin{cases} \Rightarrow m,\ si\ x = True \\ \Rightarrow n, en\ otro\ caso \end{cases}$

Puede verse ahora un ejemplo de ejecución de un "programa" usando Multi-Combinadores Categóricos. La λ-expresión

$$(\lambda x \lambda y.y(\lambda z.z\ x)\ x)\ 2\ +$$

se traduce para Categorical Multi-Combinators usando el algoritmo de compilación como

$$L^1(0(L^0(0)1)1)\ 2\ +$$

y usando las leyes dadas más arriba, puede reescribirse sucesivamente

$$\overset{M.2}{\Rightarrow} \quad \circ(0(L^0(0)1)1)\ (P\ 2\ +)$$

$$\overset{M.1}{\Rightarrow} + (\circ (L^0(0)1)\ (P\ 2\ +))\ 2$$
$$\overset{M.1}{\Rightarrow} + (L^0(0)\ 2)\ 2$$
$$\overset{M.2}{\Rightarrow} + 2\ 2$$
$$\overset{+}{\Rightarrow} 4$$

Como se observa, en la secuencia de reducciones, el código de Multi-Combinadores sufre una "metamorfosis". La aplicación de la regla (M.2) cambia la estructura del código y genera un "ambiente", en el cual, a cada variable se asocia un valor. Este "ambiente de evaluación" se distribuye en el cuerpo de la Multi- abstracción (L^n) por medio de sucesivas aplicaciones de la regla (M.1).

2 La Máquina de Reducción de Grafos

La idea de este tipo de máquinas es presentada por Turner en [18]. Se basa en la construcción (y transformación) de un grafo, cuyos nodos son celdas donde se almacena información, y los arcos están representados por punteros a dichas celdas.

Por razones de tiempo y objetivos del trabajo, no utilizamos un método formal de especificación de la máquina de Multi-Combinadores Categóricos. La especificación en el método VDM puede verse en [14]

Antes de considerar detalles de implementación damos una explicación, con una estructura funcional, de la máquina. La misma tiene por objeto fijar el comportamiento y mostrar la "filosofía" de la solución adoptada.

Esta "especificación funcional" se usa como paso intermedio en la construcción de la implementación final.

Debe tenerse en cuenta que serán necesarios ajustes de implementación.

2.1 Nuevos Combinadores

Observando la regla (M.2) abajo,

$$\textbf{(M.2)}\ L^n(y)\ x_0 x_1 \cdots x_n x_{n+1} \cdots x_z \begin{cases} \Rightarrow y\ x_{n+1} \ldots x_z \textit{ si } y \textit{ es una constante} \\ \Rightarrow x_{n-y} x_{n+1} \ldots x_z \textit{ si } y \textit{ es una variable} \\ \Rightarrow (\circ y\ (P\ x_0 \cdots x_n)) x_{n+1} \cdots x_z, \textit{en otro caso} \end{cases}$$

Las funciones constantes y variables pueden ser identificadas en tiempo de compilación; posibilitando darles un tratamiento más sencillo que el de las funciones más complejas.

En el caso de las funciones constantes, tenemos que la regla anterior expresa:

$$L^n(c)\ x_0 x_1 \cdots x_n x_{n+1} \cdots x_z \Rightarrow c\ x_{n+1} \ldots x_z \textit{ si } c \textit{ es una constante}$$

De esta forma, puede agregarse al sistema un nuevo combinador (K_c^n), para funciones constantes; y con regla de reescritura

(K) $K^n(c)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow c\ x_{n+1}\ldots x_z$

Análogamente, para las funciones Variable, en el sistema original la regla (M.2) se comporta

$L^n(i)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow x_{n-i}x_{n+1}\ldots x_z$ *si* i *es una variable*

Pudoendo entonces dar una nueva regla

(V) $V^n(i)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow x_{n-i}x_{n+1}\ldots x_z$

Y para expresiones más complejas podemos dar la regla (L) siguiente:

(L) $L^n(y)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow (\circ y\ (P\ x_0\cdots x_n))x_{n+1}\cdots x_z$

Con lo cual, eliminando la regla (M.2), el nuevo sistema queda:

(M.1) $\circ(x_cx_0x_1x_2\ldots x_n)\ (P\ y_m\ldots y_1y_0) \Rightarrow (x_cx'_0x'_1\ldots x'_n)$,

$$\text{donde} \begin{cases} x'_i & = x_i \text{ if } x_i \text{ es una constante o de tipo } L^a(b) \\ & = y_k \text{ if } x_i \text{ es la variable } k \\ & = \circ x_i\ (P\ y_m\ldots y_1y_0), \text{ en otro caso} \end{cases}$$

(K) $K^n(c)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow c\ x_{n+1}\ldots x_z$

(V) $V^n(i)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow x_{n-i}x_{n+1}\ldots x_z$

(L) $L^n(y)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow (\circ y\ (P\ x_0\cdots x_n))x_{n+1}\cdots x_z$

(S) $S^n(D\ x_i\ldots x_1x_0) \Rightarrow x_n$

(+) $+xy \Rightarrow x+y$

(=) $Cond\ x\ m\ n \begin{cases} \Rightarrow m, \text{ si } x = True \\ \Rightarrow n, \text{ en otro caso} \end{cases}$

2.2 Estructuras de Datos

El primer paso en la "especificación" de la máquina es la identificación de las estructuras de datos necesarias.

Tanto en [11] como en [15] se concluye que una representación con grafos "de gran profundidad" lleva a implementaciones ineficientes. Por tanto, en esta implementación se utilizan celdas de tamaño variable, lo que permite una representación más compacta de los términos.

Las celdas son representadas por secuencias acotadas, lo que da la ventaja de poder acceder en tiempo constante a cada componente de las mismas.

Cada término de Multi-Combinadores Categóricos se denota por una lista

$$[x_0, x_1, \ldots x_n]$$

En la definición de los elementos de estas listas debe tenerse en cuenta la representación de

- Constantes.
- Variables.
- Operadores del lenguaje (cond, mas,...).
- Multi-abstracción.
- Constructor de Bloque de Datos.
- Selectores de Datos.

Además, cualquier término puede contener sub-términos complejos.

Son **términos** de la Máquina de Reducción sólo aquellas listas que respondan a alguno de los siguientes esquemas:

- $[cte(c), x_0, \ldots, x_z]$ con c una constante
- $[var(i), x_0, \ldots, x_z]$ con $i \in N$
- $[D(n), a_0, \ldots, a_n]$ con $n \in N$
- $[mas, a_0, a_1]$
- $[cond, a_0, a_1, a_2]$
- $[L(n), a_0, x_1, \ldots, x_z]$ con $n \in N$
- $[V(n,k), x_0, \ldots, x_z]$ con $0 <= k <= n$
- $[K(n,m), x_0, \ldots, x_z]$ con $n \in N$ y m constante
- $[alfa, x_0, \ldots, x_z]$
- $[P(n), b_0, \ldots, b_n]$

donde α es un término,
los a_i son constantes, variables o términos, y deben figurar en el término
los b_i son constantes o términos, y deben figurar en el término
los x_i son constantes, variables o términos y pueden omitirse.

Esta forma de presentar las celdas tiene el inconveniente de no expresar explicitamente la posibilidad de compartir términos. Esta posibilidad es tenida en cuenta en la implementación.

2.3 Distribución del ambiente de evaluación

Observando la regla (M.1), puede verse que el sistema de Multi-Combinadores categóricos permite dos formas de substituir valores en variables:

- Retardando la substitución hasta que sea necesaria para la evaluación.

- Substituyendo todas las variables del cuerpo de la función aplicada, en un solo paso.

Esta segunda forma (llamada "eager environment distribution") es considerada la más conveniente, pues no interfiere en el proceso de evaluación propiamente dicho, - dado básicamente por las reglas (K), (V) y (L), responsables por las Multi-β-reducciones.

El proceso de "eager environment distribution" presenta la ventaja de no necesitar de la genereción dinámica de celdas para el ambiente; pero puede caerse en el caso de substituir términos que durante la ejecución serían descartados.

La "función" *EnvDi* dada en la próxima sección realiza esta tarea.

2.4 Dinámica de la reducción

Con esto, es posible dar la máquina de reducción de grafos, que toma un término y devuelve su forma normal equivalente (si es que existe).

La idea básica de la función de evaluación es tomar un nodo del grafo y retornarlo si ya está en forma normal; si el nodo no está evaluado (es un nodo de Multi-aplicación), se realiza la operación indicada en él, componiendo un nuevo término, que será evaluado recursivamente.

Así, la función de evaluación queda

$$eval[cte(c), x_0, \ldots, x_n] = [cte(c), x_0, \ldots, x_n]$$

$$eval[undefined, x_0, \ldots, x_n] = [undefined]$$

$$eval[var(i), x_0, \ldots, x_n] = [var(i), x_0, \ldots, x_n]$$

$$eval[L(m), x_0, \ldots, x_n] = [L(m), x_0, \ldots, x_n]$$

$$eval[D(m), x_0, \ldots, x_n] = [D(m), eval\ x_0, \ldots, eval\ x_n]$$

$$eval[S^i, [D, x_0, \ldots, x_n]] = [x_i]$$

$$\begin{array}{l} eval[mas, a_0, a_1] = \\ \quad lett_1 = eval[a_0] \\ \quad in \\ \quad if t_1 = [cte(i)] \\ \quad then\ lett_2 = eval[a_2] \\ \quad in \\ \quad if t_2 = [cte(j)] \\ \quad then[cte(i+j)] \\ \quad else[undefined] \\ \quad else[undefined] \end{array}$$

Respectivamente para los otros operadores aritméticos y lógicos

$$eval[cond, a_0, a_1, a_2] =$$
$$\quad lett = eval[a_0]$$
$$\quad in$$
$$\quad ift = [cte(truc)]$$
$$\quad theneval[a_1]$$
$$\quad elseeval[a_2]$$

$$eval[V(n,k), x_0, \ldots, x_n, \ldots, x_z] = eval[x_{n-k}, x_{n+1}, \ldots, x_z]$$

$$CMC - eval[K(n,i), x_0, \ldots, x_n, \ldots, x_z] = [cte(i), x_{n+1}, \ldots, x_z]$$

$$eval[[L(n), y_0, \ldots, y_m], x_0, \ldots, x_n, \ldots, x_z] =$$
$$\quad eval(appendEnvDi([y_0, \ldots, y_m], [P(n), x_0, \ldots, x_n])[x_{n+1} \ldots x_z])$$

$$eval[alfa, x_0, \ldots, x_n] =$$
$$\quad eval(append(evalalfa), [x_0, \ldots, x_n])$$

$$EnvDi([x_0 \ldots x_n], [y_m \ldots y_0]) = [x'_0 \ldots x'_n]$$

$$donde \begin{cases} x'_i & = x_i \text{ si } x_i \text{ es una constante o de tipo } [L^a(b)\ldots] \\ & = y_k \text{ si } x_i \text{ es la variable } k \\ & = EnvDi([x_i], [y_m \ldots y_0]), \text{ sino} \end{cases}$$

Puede verse que esta máquina realiza las reglas de reescritura de Multi-Combinadores Categóricos, ya que ella es, básicamente, otra forma sintáctica para expresar dichas reglas.

3 Implementación de la Máquina de Reducción de Grafos

La implementación final de la máquina de reducción de Multi-Combinadores Categóricos está realizada en el lenguaje C, bajo el Sistema Operativo VMS, y sobre un equipo VAX/750.

La máquina consta de cuatro partes principales: lector, impresor, gestor de la memoria y evaluador.

El lector y el impresor son rutinas muy simples de estrada-salida. El módulo lector es el menos importante de esta parte del proyecto. Realiza la lectura de un "programa" en una codificación numérica del lenguaje de la máquina. No se puso mucho énfasis en permitir una gran facilidad de uso del lector, ya que en el futuro será substituido por un analizador sintáctico para la gramática de SASL que efectúe la carga de la memoria.

La rutina de impresión recorre recursivamente la lista resultado, imprimiendo de forma comprensible su contenido.

3.1 El gestor de la Memoria

En la especificación dada no se tienen en cuenta las limitaciones e inconvenientes provocados por la necesidad de trabajar con máquinas de memo-

ria finita. Por esto fue necesario usar un administrador de la memoria disponible. El mismo consta de dos partes:

- Un administrador de la memoria en uso, que gestiona por demanda, devolviendo la dirección de la primera de las células pedidas (que deben ser contiguas).
- Un *Garbage Collector*, que es invocado por el anterior.

Se trata de un "*Copying Garbage Collector*". Usa una técnica extremadamente simple, que consiste en tener dos heaps gemelos que se alternan en el uso. Cuando el gestor no posee en la memoria en uso la cantidad de celdas requeridas, se invoca al Garbage Collector, que copia (compactando) las celdas conocidas por el evaluador hacia la otra memoria. Luego de ésta operación, queda en uso la memoria recién asignada.

En [12] puede verse un algoritmo de Garbage Collection capaz de tratar el tipo de celdas usadas aquí (de longitud variable y con estructura cíclica). Este algoritmo no se implementó por razones de tiempo y prioridad en los objetivos del proyecto.

3.2 El Intérprete de Multi-Combinadores Categóricos

Es básicamente una traducción de las funciones dadas en la especificación; con la diferencia básica de tener en cuenta em manejo de la memoria.

Esto se hace mediante el mantenimiento de una estructura de datos que aquí es vista como una pila de referencias a direcciones de la memoria. En esta pila se almacenan las direcciones de los sub-términos alcanzables desde el intérprete. El Garbage Collector ve a ésta estructura de datos como un arreglo, y obtiene de allí las direcciones de comienzo de las celdas a copiar.

En realidad, se extiende el de esta "pila", utilizándola también para el pasaje de parámetros en las llamadas del evaluador (recordemos que se trata de un programa recursivo).

4 Consideraciones sobre la performance del sistema

En esta sección se hace un análisis comparativo entre el desempeño de la presente implementación de la máquina de Multi-Combinadores Categóricos y el de una implementación de K.R.C [19], basada en la máquina de combinadores de Turner[18].

4.1 Programas de Prueba

Se eligieron como programas de prueba la función de Fibonacci, dada por el algoritmo

```
fib n = (n < 2) -> 1; fib (n - 1) + fib (n - 2)
```

Esta función (evaluada en 20) es un "brenchmark" standard para lenguajes funcionales, haciendo uso extensivo de la recursión, y siendo un caso clásico de función estricta.

Otro programa de test es

```
twice twice twice succ 1
```

donde

```
twice f x = f (f x)
succ n    = n + 1
```

que explota el uso de funciones de orden superior[1].

Con estos tests se obtuvieron los siguientes resultados:

	M-C.C	K.R.C
fib 20	81 s	71 s
twice	0.10 s	0.10 s

Estos números son tiempos de CPU de VAX/750.

4.2 Análisis de los resultados

Aunque la diferencia de tiempo entre los medidos de *fib* 20 son del orden del 15 % a favor de K.R.C, puede decirse que los resultados obtenidos son favorables a la máquina de Multi-Combinadores Categóricos, si se tiene en cuenta que:

- K.R.C usa aritmética entera, mientras que M-C.C está implementado con aritmética real.
- K.R.C posee muchas rutinas implementadas en VAX-Assembler, cosa que podría hacerse también en M-C.C.
- M-C.C no está concluida. Al momento de redactar este artículo están comenzando a efectuarse optimizaciones a la implementación, removiendo ineficiencias, al adoptar una representación más compacta para las celdas, y al evitar la generacion de grafos no necesarios.

Conclusiones

Multi-Combinadores Categoricos demuestran ser un formalismo sencillo y que permite implementaciones eficientes para lenguajes funcionales con lazy evaluation.

Dado el poco tiempo transcurrido desde su desarrollo, creemos que todavía puede ser refinado, posibilitando implementaciones mas rápidas que ésta, y que, como la presente traten de obtener eficiencia, sin dificultar la claridad de los programas.

[1] Funciones que toman y/o devuelven funciones.

Agradecimientos

Agradecemos el apoyo de Claudio Hermida, Alberto Pardo y Regina Motz.

Este trabajo es parte de un proyecto de Pasantía de la ESLAI, y está siendo financiado por el Programa Argrntino-Brasileño de Investigación y Estudios Avanzados en Informática - Proyecto ETHOS.

Referencias

1. H.P.Barendregt, The Lambda Calculus Its Syntax and Semantics, North Holland (1984).

2. N.G.DeBruijn, Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, Indag.Math. 34, 381-392 (1972).

3. G.Cousineau, P-L.Curien, & M.Mauny, The Categorical Abstract Machine, Functional Programming Languages and Computer Architecture, ed. J-P.Jouannaud, SLNCS 201.

4. P-L.Curien, Categorical Combinators, Sequential Algorithms and Functional Programming, Research Notes in Theoretical Computer Science, Pitman Publishing Ltd., (1986).

5. R.J.M.Hughes, The Design and Implementation of Programming Languages, Ph.D. Thesis, Oxford Univ. Comp.Lab., July 1983.

6. T.Johnsson, Compiling Lazy Functional Languages, PhD. Thesis, Chalmers University of Technology, Göteborg, 1987.

7. J.Lambek, From Lambda-calculus to Cartesian Closed Categories, in To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, ed J.P.Seldin and J.R.Hindley, Academic Press (1980).

8. R.D.Lins, A New Formula for The Execution of Categorical Combinators, Proceedings of 8th. International Conference on Automated Deduction, Oxford, England, July 1986, LNCS 230, p 89 - 98, Springer Verlag.

9. R.D.Lins, On the Efficiency of Categorical Combinators as a Rewriting System, Software Practice & Experience, Vol 17(8), 547-559, (August 1987).

10. R.D.Lins, Categorical Multi-Combinators, Proceedings of Third International Conference on Functional Programming and Computer Architecture, Portland, USA, September 1987, LNCS 274, p 60-79, Springer Verlag.

11. R.D.Lins & S.J.Thompson, Implementing SASL using Categorical Multi-Combinators, Computing Lab. Report N.47, The Univ. of Kent, at Canterbury, 1st. revision, June/88. (a ser publicado por Software Practice & Experience).

12. R.D.Lins & S.J.Thompson, Reference Counting Garbage Collection for Cyclic Structures, June/88 (a ser publicado como U.K.C. Lab. Report e submetido a Software Practice & Experience).

13. R.D.Lins, Compiling Categorical Multi-Combinators, en preparación.

14. M.A.Macedo, Uma Especificação VDM para uma máquina de Multi-Combinadores Categóricos, en preparación.

15. S.Peyton-Jones, The Implementation of Functional Languages, Prentice Hall, (1987).

16. D.Scott, Relating Theories of the Lambda-Calculus, in To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, ed. J.P.Seldin and J.R.Hindley, Academic Press (1980).

17. D.A.Turner, SASL Language Manual, Revised Version: Nov/83, Computing Lab., UNIKENT, (1983).

18. D.A.Turner, A New Implementation Technique for Applicative Languages, Software Practice and Experience, Vol 9, 31-49 (1979).

19. D.A.Turner, Recursion Equations as a Programming Language, Functional Programming and its Applications, J.Darlington, P.Henderson, and D.A.Turner eds., Cambridge University Press (1982).

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Un Esquema para la Programación Orientada a Objetos en PROLOG[1]

Javier Pinto
Depto. Ciencia de la Computación
Pontificia Universidad Católica de Chile
Casilla 6177, Santiago, CHILE
Correo Electrónico (UUCP): jpinto@juncal.puc.cl

Resumen. En este trabajo se presenta un esquema para incorporar la Programación Orientada a Objetos al lenguaje Prolog. Una ventaja importante de este esquema es el apoyo que ofrece para la *abstracción de datos*, permitiendo encapsular definiciones de Tipos Abstractos de Datos. Además de esta capacidad, ausente en Prolog, la Programación Orientada a Objetos incorpora el concepto de *herencia de atributos*, permitiendo la jerarquización de los Tipos Abstractos de Datos que se definan.

El esquema presentado expande Prolog, incorporando un conjunto de predicados para la definición de las clases y de sus atributos. Al definir una clase, se introduce su definición como "query" u objetivo, que al ser satisfecho produce el efecto lateral de instalar, vía asserts, las definiciones en la base de datos de Prolog.

Además de las características funcionales de la extensión propuesta, que permiten la definición y utilización de clases, se discute la integración semántica de la extensión al lenguaje Prolog y se presentan algunos aspectos relacionados con la implementación.

Introducción

Una desventaja importante de Programación en Lógica Clásica [Hogg84], ejemplificada por el lenguaje PROLOG [Cloc84], es que sólo permite colecciones de reglas y hechos absolutamente planas [Subr85]. Esta característica del lenguaje impide modularizar las reglas, así como también impide la definición de jerarquías entre conjuntos de reglas.

Por otra parte, se ha argüido que PROLOG es inadecuado para la construcción de aplicaciones de Inteligencia Artificial, por ejemplo Sistemas Expertos, debido a que propugna un solo estilo o paradigma de programación [Bobr85].

En este trabajo se presenta un esquema para mejorar la posición de PROLOG con respecto a estas deficiencias, construyendo sobre el lenguaje un conjunto de predicados, denominado CLASS, que permita incorporar "Programación Orientada a Objetos" [Stef86] al lenguaje. En primer término, se introducen conceptos de la Programación Orientada a Objetos en el marco general de PROLOG aumentado con CLASS. Se presenta el mecanismo de herencia de atributos y el manejo de excepciones en el contexto de un ejemplo. Posteriormente se analizan los problemas semánticos asociados a la integracion de CLASS con PROLOG, se discuten algunos aspectos de implementación, y finalmente se presentan las conclusiones y se discuten posibles líneas de trabajo futuro.

Aspectos Básicos.

El conjunto CLASS de predicados permite la definición de clases de objetos, de modo que todos los objetos que pertenecen a una misma clase comparten la definición de un conjunto de datos, o variables de

[1]Este trabajo ha sido financiado en parte por la Dirección de Investigacion de la Universidad Catolica, bajo el proyecto 13/87.

instancia y un conjunto de predicados. Los conjuntos de predicados definidos para una clase reciben el nombre de métodos de la clase. Así, es posible definir una clase usando:

```
?- defClass(persona,[class])[1].                                    (1)
?- defInstanceVariables(persona,[(nombre,[]),                       (2)
                                 (apellido,[]),
                                 (padre,[]),
                                 (madre,[]),
                                 (edad,0)
                                 (hijos,[])]).
```

El primer predicado, **defClass**, se satisface si su primer argumento, **persona**, puede definirse como una nueva clase; el segundo argumento será explicado más adelante. El segundo predicado, **defInstanceVariables**, define el conjunto de variables de instancia asociadas a los objetos de la clase especificada por el primer argumento, el segundo argumento es un conjunto de pares ordenados (<nombre de variable>, <valor inicial>). Para definir que un objeto pertenece a una determinada clase, es necesario especificar:

```
?- instanceOf(jorge, persona,[(nombre,"Jorge")]).                   (3)
```

mediante lo cual, el objeto **jorge** queda definido como una instancia de **persona** con sus variables de instancia inicializadas como en (2), exceptuando **nombre**, que queda definido como "Jorge".

Los métodos permiten asociar predicados a una clase, de modo que éstos se pueden invocar sólo con los objetos pertenecientes a dicha clase. Por ejemplo:

```
?- defMethod(persona,                                               (4)
        [(actualizarEdad:-  send(self,get(edad,Y)),
                            Z is Y+1,
                            send(self,set(edad,Z)))]).
```

define el método **actualizarEdad** para la clase **persona**, el cual modifica el valor asociado a la variable de instancia **edad** aumentándolo en uno. En la definición de este método se han introducido dos características importantes asociadas a la programación orientada a objetos (OOP), según se discute a continuación. En OOP, la interacción con los objetos se realiza a través del envío de mensajes, para lo cual se utiliza el predicado **send**, de modo que un objeto recibe mensajes que es capaz de interpretar solamente si se ha definido un método para ello. De este modo, el satisfacer predicados sobre determinados objetos es equivalente al envío de mensajes a ellos. Así, para invocar un método asociado a un objeto, se le envía un *mensaje* a éste último. Por ejemplo, para actualizar la edad de "Jorge", se especifica:

```
?- send(jorge,actualizarEdad).                                      (5)
```

que tiene el efecto descrito por el método definido en (4). Por otra parte, el símbolo distinguido **self** es utilizado en la definición de métodos para referirse al objeto que recibe el mensaje. Además, en la definición del método **actualizarEdad** se envían los mensajes **get** y **put** a **self**, los métodos que responden a estos mensajes quedan asociados a las clases en forma automática, cuando se realiza la definición de la clase correspondiente. El método **get** permite obtener el valor de una variable de instancia, mientras que el método **put** permite alterarlo, para lo cual se retractan los valores previos. Entonces, cuando **jorge** recibe el mensaje **actualizarEdad**, se invoca el predicado especificado en (4), el cual envía el mensaje **get**, con argumentos **edad** e **Y**, al mismo objeto **jorge**. Luego actualiza el valor

---

[1] El símbolo '?-', indica que los términos a la derecha del mismo se introducen como objetivos. La razón para introducir estos términos como objetivos, y no como aserciones simples, es que la información registrada en la base de datos debe ser actualizada cada vez que se defina una nueva clase, método u objeto. La alternativa de aceptar estos términos como aserciones, obligaría a introducir un "overhead" en el momento de utilizar los objetos definidos.

asociado a **edad**, cuyo valor original es retornado como binding de **Y**, al satisfacer **get(edad,Y)**, y, finalmente, se actualiza el "binding" de la variable de instancia **edad**.

Comúnmente, al declarar que un cierto objeto pertenece a una determinada clase, es necesario ejecutar algún código que permita inicializar apropiadamente las estructuras de datos asociadas a éste. Por ejemplo, suponga que se define el siguiente objeto:

```
?- instanceOf(david,persona,[ (nombre,"David López"),          (6)
                              (hijos, [benjamin,luis])]).
```

En esta situación, se ha declarado que **david** tiene como **hijos** a **benjamin** y **luis**, los cuales son también objetos que están siendo manipulados por el programa. Dada la forma en que se ha decidido estructurar la información, es necesario que al declarar el conocimiento respecto de los hijos de **david**, se actualice la información asociada a ellos. Esto es, se debe agregar como atributo de **benjamin** y **luis** el valor **david** para sus variables de instancia **padre**. Para realizar esta operación, es posible exigir que el usuario de las clases las utilice en forma disciplinada. Una mejor solución permite que se especifique un método para ser ejecutado cada vez que se cree una nueva instancia de una determinada clase, de manera que los datos mantengan su consistencia en forma automática. Así, la definición:

```
?- defMethod(persona,                                          (7)
        [(beforeMethod:- send(self,get(hijos,L)).
                         compatHijos(self,L))]).
```

especifica el método **beforeMethod**[1] para la clase persona. Para satisfacer el objetivo:

```
?- instanceOf(<objeto>, <clase>, <inicialización>).
```

se intenta satisfacer el método **beforeMethod** dado en (7). Si este método falla, entonces, la definición de la instancia también falla. En el ejemplo, **beforeMethod** falla cuando **compatHijos** falla, lo que ocurre si el objeto **self** (en este caso **david**), no puede ser **padre** de **benjamin** o **luis**, por ejemplo por tener éstos un padre diferente previamente definido.

Herencia de Atributos.

La Programación Orientada a Objetos, originalmente introducida por el Lenguaje Simula67 [Dahl70], ha hecho aportes muy importantes para el desarrollo de metodologías de diseño de software. El concepto más importante introducido por OOP es el de herencia de propiedades, concepto que permite jerarquizar los tipos de datos definidos en forma tradicional. La utilización adecuada de esta herramienta de conceptualización, entre otras, facilita la reutilización de código [Meye87].

Las características de CLASS presentadas hasta el momento posibilitan la definición de clases de objetos, de modo que el conjunto de métodos asociado a cada clase permite formar módulos de software. Cada uno de estos módulos puede interpretarse como la implementación de un Tipo de Dato Abstracto [Bish86]. Además de esto, en CLASS también se incorpora la idea de herencia, permitiendo definir jerarquías de clases. Por ejemplo, es posible definir que la clase **niño** es una subclase de **persona**:

```
?- defClass(niño,[persona]).                                   (8)
```

Esta definición crea una nueva clase de objetos (**niño**), la cual *hereda* las propiedades asociadas a la clase **persona**, esto es, su definición de variables de instancia, más su definición de métodos. Así, al declarar:

```
?- instanceOf(hugo,niño,[]).                                   (9)
```

[1] El nombre **beforeMethod** queda reservado para ser utilizado en el contexto indicado.

el objeto **hugo** hereda el conjunto de definiciones asociadas a **persona**, incluyendo todo aquello que es, a su vez, heredado por **persona**.

En la práctica, todos los objetos definidos para el universo de un programa tendrán en común un conjunto de primitivas básicas. Por ejemplo, todos los objetos son capaces de responder a los mensajes **get** y **put**, los cuales reciben como argumentos nombres y valores de variables de instancia asociadas a los objetos. Para lograr que todos los objetos definidos incorporen efectivamente el conjunto de primitivas básicas, sería posible agregar éstas como métodos de cada una de las clases que se definan. Una solución de este tipo es obviamente inadecuada, por el "overhead" extremo de producir código de similares características para cada una de las clases. Una mejor solución, adoptada por la mayoría de los lenguajes para la Programación Orientada a Objetos, es definir una clase elemental que incorpora un conjunto básico de métodos y de variables de instancia. En CLASS, esta clase básica es llamada **class** y todas las clases definidas por el usuario heredan sus atributos[1].

Un Ejemplo.

Este trabajo fue desarrollado en el contexto de un proyecto para la construcción de software para la simulación cualitativa de procesos [Pint88]. PROLOG surgió naturalmente como una de las opciones para la construcción del sistema. Sin embargo, inmediatamente se tropezó con la dificultad de organizar el conocimiento asociado al modelo que se deseaba construir.

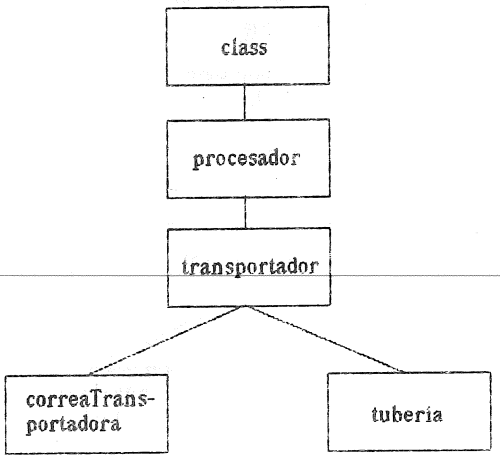


Figura 1. Una jerarquía de clases

Particularmente, en la aplicación mencionada interesa incorporar conocimiento relativo a ciertos dispositivos procesadores. Por ejemplo, existe un molino de mineral, correas transportadoras, tuberías, motores, etc. Todos estos dispositivos comparten entre sí un conjunto de características, lo que permite agruparlos en *clases* de objetos. Por ejemplo, las correas transportadoras y las tuberías pertenecen a la clase de dispositivos que permiten transportar material entre dos puntos. Sin embargo, correas transportadoras y tuberías no pueden aparecer como instancias de una misma clase, dado que estos dispositivos presentan diferencias importantes entre sí. En vez de esto, es posible definir independientemente las clases **tubería** y **correaTransportadora**, pero ambas como sub-clases de la clase superior **transportador**. Esta jerarquía de clases es ilustrada en la figura 1.

---

[1]En rigor, en CLASS es necesario especificar explícitamente cuando una clase será subclase de **class**. Por ejemplo, en (1), la clase **persona** fue declarada como subclase de **class** en forma explícita; en cambio, en (8), la clase **niño** simplemente hereda esta definición de su clase superior **persona**.

Para declarar este conocimiento en CLASS, se puede especificar:

```
?- defClass(procesador, [class]).                                    (10)
        % Se define que procesador es una clase de alto nivel (descendiente
        % directo de class).

?- defInstanceVariables(procesador,[ (vEndógena, []),                (11)
                                     (vExógena,[])]).
        % El estado del procesador es caracterizado por dos conjuntos de
        % variables, exógenas y endógenas, las que se introducen como
        % variables de instancia de la clase.

?- defClass(transportador,[procesador]).                             (12)
        % La clase transportador es descendiente de la clase procesador.
?- defInstanceVariables(transportador,[(material , [])]).            (13)
        % Los objetos transportadores transportan material específico
        % (mineral, pulpa, etc.).

?- defClass(molino,[procesador]).                                    (14)

?- defClass(correaTransportadora,[transportador]).                   (15)
?- defInstanceVariables(correaTransportadora,[(motor , [])]).        (16)
?- defClass(tuberia,[transportador]).
?- defInstanceVariables(tuberia,[(bomba , [])]).                     (17)
        % Tanto correaTransportadora como tuberia son subclases de
        % transportador. La clase correaTransportadora tiene una
        % variable de instancia motor. Análogamente, tuberia tiene asociada
        % una bomba.
```

En (16), al definir la variable de instancia **motor**, se tiene la intención de que ésta quede asociada a un objeto de clase **motor**[1], para lo cual es posible utilizar la siguiente secuencia:

```
?- chequearMotor(mot1), send(correa, set(motor, mot1)).              (18)
```

donde **chequearMotor**, es satisfecho sólo si **mot1** es una instancia de un motor y es un motor apropiado para mover una correa. Sin embargo esta solución es pobre, por cuanto se requeriría utilizar predicados de "chequeo" para todas las variables de instancia interesantes. En vez de esto, sería preferible utilizar:

```
?- send(correa, set(motor, mot1)).                                   (19)
```

siempre que se pueda especificar que cuando se realice un **set** de la variable de instancia **motor**, en forma automática se haga un chequeo previo del valor asociado a **mot1**. Para esto, es posible incluir un predicado *if-added* a las variables de instancia (solución común en lenguajes orientados a *frames* [Fike85]). En vez de esto, se ha preferido manejar este tipo de situaciones como *excepciones*. Una excepción se produce cuando en una jerarquía, un método de una clase superior no es adecuado para responder mensajes específicos de clases inferiores a ella. Típicamente, esta situación se produce cuando un método debe ser refinado o especializado. Para especializar un método, basta con redefinirlo para la clase inferior en cuestión. Este esquema funciona adecuadamente, pues al enviar un mensaje a un objeto, primero se revisa si existe un método para responder a él asociado a la clase más baja del objeto. Si dicho método no se encuentra, se busca un método asociado a una de sus clases superiores.

---

[1]Esta asociación no está relacionada con la asociación jerárquica entre clases. Más bien es una asociación entre instancias de diferentes clases, relación que debe ser manejada en forma explícita por el programador.

Entonces, en el ejemplo es posible redefinir el método **set** asociado a la clase **correaTransportadora** para la variable de instancia **motor**, de modo que además de asociar un valor a la variable de instancia, haga el chequeo correspondiente. Un primer intento, refinado más adelante, para realizar la redefinición es el siguiente:

```
?- defMethod(correaTransportadora,                                    (20)
     [ (set(motor,Value):-
          !,                                                         (21)
          chequearMotor(Value),
          send(self, set(motor, Value)))]).                          (22)
```

En la redefinición de este método, es necesario destacar dos detalles importantes. El primero es el *cut* de la línea (21), el cual asegura que el objetivo asociado al mensaje **set**, para **motor**, no se verá satisfecho por un método **set** asociado a una clase superior a **correaTransportadora**. Este tema se discute más adelante. En segundo término, en (22) se pretende modificar el valor de la variable de instancia **motor** asociada a la instancia de **correaTransportadora** que recibe el mensaje. Sin embargo, el efecto real es que (22) queda como un llamado recursivo, donde, para satisfacer el mensaje **set**, se envía un mensaje **set** al mismo objeto y con los mismos argumentos, produciéndose un ciclo sin término. Una versión correcta para la redefinición será introducida en la sección subsiguiente, donde se discute en mayor detalle el problema de la redefinición de métodos.

Herencia Múltiple.

En algunos lenguajes orientados a objetos, es posible definir clases con herencia múltiple, permitiendo crear jerarquías representadas por grafos dirigidos. En general, el mecanismo de herencia múltiple no es difícil de implementar, la dificultad asociada a este esquema se deriva de la complejidad semántica resultante. En particular, sería posible tener una jerarquía como la que ilustra la figura 2.

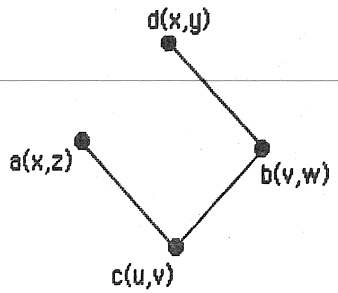


Figura 2. Una jerarquía con herencia múltiple.

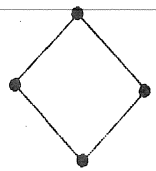


Figura 3. Un ciclo en el grafo no dirigido subyacente, en la jerarquía de clases.

Cada nodo en la jerarquía de la figura tiene una etiqueta **c(m,n)**, tal que **c** representa una clase con métodos definidos **m** y **n**. Cuando se envía un mensaje a un objeto perteneciente a la clase **c**, es necesario establecer un ordenamiento entre los predecesores de éste para decidir qué clase debe interpretarlo y responder. Por ejemplo, ante un requerimiento **send(c_1,X)**, con $c_1 \in$ **c**, es necesario determinar si c_1 debe comportarse como objeto de la clase **d**, o de la clase **a**. La situación se complica aún más cuando el grafo tiene ciclos en el grafo no dirigido subyacente, como ilustra la figura 3.

Una posible solución, [Moon86]), es recorrer el grafo *bottom-up* en *depth first* en algún orden, listando todas las clases con sus métodos, si alguna se repite (debido a ciclos) se eliminan todas sus ocurrencias, salvo la primera. Dicho esquema resuelve el problema, aunque en forma solamente parcial, puesto que el aporte semántico de la solución es bastante oscuro. ¿Qué diferencia tendría, desde

un punto de vista semántico, realizar un recorrido *breadth-first*, en vez de un *depth-first?*; en CLASS este problema aún no se ha abordado, únicamente se permite definir jerarquías con forma de árbol (sólo un ancestro por clase).

Manejo de Excepciones o Especialización de Métodos.

Típicamente, al recomponer métodos se requiere utilizar el método original, situación ejemplificada en (20). En este ejemplo, el refinamiento del método **set**, requiere del método original. Sin embargo, con las herramientas provistas hasta este momento, no es posible referirse a este método a través de un objeto de la clase **correaTransportadora**. Lo que se requiere es un mecanismo explícito para hacer referencia a la jerarquía de clases. En CLASS, al igual que en lenguajes con capacidades para OOP (ej. Flavors [Moon86] y Smalltalk [Gold83]), se provee el nombre distinguido **super**, el cual permite hacer referencias a clases superiores en el momento de la definición de métodos. Por ejemplo, el siguiente método redefine el método **set** definido en (20):

```
?- defMethod(correaTransportadora,                                    (23)
        [ (set(motor,Value):-
                !,
                chequearMotor(Value),
                send(super,set(motor,Value)))]).
```

Aquí se ha incluído una referencia al símbolo distinguido **super**, cuando una instancia de la clase **correaTransportadora** recibe un mensaje **set(motor,Value)**, el método refinado envía el mismo mensaje (**set**), pero utiliza el símbolo **super**. Esto significa que se debe buscar un método para responder al mensaje **set** a partir de las clases superiores a **correaTransportadora**, evitando así el fenómeno de recursión que se produce en (22). Así, a menos que se haya redefinido el método **set** para una clase superior a **correaTransportadora**, éste será atendido por el predicado original, definido para la clase **class**.

Consideraciones Semánticas.

En esta sección, se estudia la integración que existe entre la extensión CLASS que se propone para el lenguaje PROLOG, y la semántica asociada a este último. Para realizar el análisis es necesario considerar los aspectos declarativos y *procedurales* de PROLOG. Para una exposición más resumida del tema, se limitirá a considerar aquellos predicados que son más interesantes, desde el punto de vista del contenido semántico asociado a ellos.

La definición de clases, métodos y variables de instancia, implica necesariamente un efecto lateral que modifica el contenido de la base de datos de PROLOG. Por ejemplo, el requerimiento:

```
?- instanceOf(hugo, niño,[]).                                        (24)
```

establece que el objeto hugo pertenece a la clase **niño**. En CLASS, para satisfacer este requerimiento se modifica el contenido de la base de datos, realizando aserciones que declaran en forma explícita las clases de los objetos presentes en el sistema. De este modo, una interpretación *procedural* es simplemente que instanceOf produce un efecto lateral (instalando declaraciones en la base de datos). Desde un punto de vista declarativo, (24) puede interpretarse como: "¿Es cierto que **hugo** es **niño**?". Por otra parte, el requerimiento:

```
?- instanceOf(hugo,X,_).                                             (25)
```

podría interpretarse como:"¿a qué **clase** pertenece **hugo**?". Sin embargo, debido al efecto lateral introducido, indeseable desde un punto de vista puramente lógico, esta interpretación es inaceptable. El problema fundamental que se produce es que no es posible determinar cuáles son las intenciones del usuario (es decir, el requerimiento es ambiguo), por ejemplo, el requerimiento (25) podría también interpretarse como la aserción: "**hugo** es un objeto que pertenece a todas las clases". ¿cuál es la

interpretación correcta?, dependerá de la aplicación específica. En CLASS, este problema se evita exigiendo que todos los parámetros de instanceOf estén instanciados al momento de ser invocado. Problemas análogos al ilustrado con **instanceOf** se presentan con la definición de métodos y de variables de instancia.

El tratamiento de los mensajes tampoco está libre de problemas, pero, afortunadamente, la situación es más clara. Cuando se envía un mensaje utilizando **send**, como en:

?- send(<objeto>,<mensaje>). (26)

send no es interpretado como un predicado cuya verdad dependa estrictamente de sus argumentos. Para una interpretación adecuada, es necesario escaparse del formalismo de lógica de primer orden, por cuanto el predicado que se invoca en (26) no está determinado por la declaración, sino que éste debe identificarse a partir del **<objeto>** y del nombre del **<mensaje>** específico. De modo que la interpretación de (26) está dada por:

send(<objeto>,<mensaje>) → (27)
predicado(clase(<objeto>),nombre(<mensaje>))(<objeto>).

donde **clase** es una función que denota la clase a la que pertenece su argumento, **nombre** es una función que denota el nombre de **<mensaje>**, y **predicado** es una función que denota un predicado (obviamente la función no es de primer orden), el cual se aplicará a **<objeto>** para obtener el valor de verdad resultante.

Por otra parte (27) puede interpretarse como una regla sintáctica (i.e. una macro), que transforma el lado izquierdo en el derecho. Si ambos parámetros de **send** se encuentran instanciados (situación más típica), entonces los argumentos a la función **predicado** son constantes, lo que implica que el literal **predicado(_,_)** es una constante. Entonces, si se exige que ambos parámetros de la función **send** sean constantes, el problema de introducir características fuera de la lógica de primer orden desaparece.

No obstante lo anterior, se ha preferido no introducir la exigencia referida. Fundamentalmente, porque la aplicación del send considerando sus argumentos variables, tiene interpretaciones que son interesantes. Particularmente, existen diferentes interpretaciones del mensaje (26), dependiendo de los objetos que se encuentren instanciados. Por ejemplo, el mensaje:

?- send(Objeto,mensajeA). (28)

puede interpretarse como "¿Existe algún objeto que pueda responder a **mensajeA**?"[1]. Por otra parte, el mensaje:

?- send(objeto,X). (29)

debiera satisfacerse con cualquier mensaje al que **objeto** pueda responder. Sin embargo, el significado del resultado es muy poco claro (e.g., ¿cómo interpretar un *yes* como respuesta a (29)?). Por esta razón, se ha decidido restringir el segundo parámetro a una función de primer orden, cuyo nombre debe aparecer como literal.

Finalmente, es posible interpretar el contenido semántico agregado a PROLOG con extensión CLASS, en el marco de la lógica de primer orden para el caso general en el que se restrinja el uso del predicado send a aquél en el cual sus argumentos estén siempre instanciados. En otro caso, sería necesario recurrir a interpretaciones bajo un esquema lógico diferente.

---

[1] Esta interpretación es análoga a aquella de [Fuku86].-

Aspectos de Implementación.

CLASS se ha desarrollado utilizando una versión de PROLOG de acuerdo al estándar de la Universidad de Edimburgo [Cloc84]. Cada uno de los predicados **defClass**, **defInstanceVariables** e **instanceOf** son predicados simples que instalan información en la base de datos. La información que se almacena es la relativa a cada clase definida, junto con la definición de la jerarquía correspondiente.

Para realizar la definición de métodos, el predicado **defMethod** realiza una traducción del método especificado en el segundo parámetro. De modo que ante una especificación como:

?- **defMethod(clase,** (30)
[(método(Arg$_1$,Arg$_2$,...,Arg$_n$):- ***Código*****(self,super)]).**

donde ***Código*****(self, super)**, representa el código del método, el cual puede tener referencias a los símbolos distinguidos **self** y **super**, se realiza una traducción a:

método(clase, Self,Arg$_1$,Arg$_2$,...,Arg$_n$):- ***Código*****(Self, super(clase)).** (31)

Así, el método original es transformado a un predicado estándar de PROLOG. En forma análoga, los mensajes (e.g. (26)) son traducidos consistentemente con la traducción de los métodos.

En la actualidad, CLASS funciona en forma interpretada sobre el sistema PROLOG, en el cual se leen las definiciones de CLASS previamente a su utilización. La ejecución interpretada de CLASS presenta la ineficiencia derivada de la búsqueda de los métodos en tiempo de ejecución. Es posible incorporar CLASS a PROLOG, de modo que los envíos de mensajes sean traducidos a invocaciones de los predicados correspondientes. Este esquema permitiría una ejecución libre del *overhead* propio a la interpretación de los mensajes.

Conclusiones y Trabajo Futuro

La extensión que se propone para PROLOG cumple con el objetivo de permitir Programación Orientada a Objetos dentro del marco de la programación en lógica. El sistema resultante hereda características importantes de ambas clases de lenguajes, esto es, lenguajes para programación en lógica y lenguajes para programación orientada a objetos.

Dentro de las ventajas más notables se encuentra la facilidad para la organización de reglas permitiendo construir estructuras conceptuales como aquellas propias de *frames*. Esta característica tiene gran importancia desde un punto de vista de Ingeniería de Software, puesto que permite la definición de módulos con encapsulamiento de predicados. Por otra parte, la programación orientada a objetos se ha reconocido como muy importante para el desarrollo de aplicaciones de IA, puesto que permite implementar con facilidad esquemas de representación basados en *frames* y redes semánticas.

En el futuro se espera especificar formalmente un lenguaje que extienda PROLOG con las ideas aquí expuestas, esperando producir la implementación de un intérprete para el mismo.

Por otra parte, es necesario ganar experiencia con la utilización de esta extensión, de modo de poder analizar, en base a casos concretos, los beneficios de la programación orientada a objetos en PROLOG.

Referencias:

[Bish86] J. Bishop, "Data Abstraction in Programming Languages", Addison Wesley, 1986.
[Bobr85] D.G. Bobrow, "If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms", IEEE Trans. Software Eng., vol. SE-11, pp. 1401-1408.

[Cloc84] W.F. Clocksin y C.S. Mellish, "Programming in Prolog, Second Edition", Springer Verlag, 1984.

[Dahl70] O.J. Dahl, B. Myhrhang y K. Nygaard, "Simula67 Common Base Language", Norwegian Computer Center, S-22, 1970.

[Fike85] R. Fikes y T. Kehler, "The Role of Frame-Based Representation in Reasoning", Communications of the ACM, September 1985, pp. 904-920.

[Fuku86] K. Fukunaga, S. Hirose, "An Experience with a Prolog-based Object-Oriented Language". Proceedings de la 1986 Object Oriented Programming Systems, Languages and Applications, 1986, pp. 224-231

[Gold83] A. Goldberg, D. Robson, "Smalltalk-80: The Language and Its Implementation", Addison - Wesley, 1983.

[Hogg84] C.J. Hogger, "Introduction to Logic Programming", Academic Press, 1984.

[Meye87] B. Meyer, "Reusability: The Case for Object-Oriented Design" IEEE Software, March 1987, pp. 50-64.

[Moon86] D. Moon, "Object Oriented Programming with Flavors", Proceedings de la 1986 Object Oriented Programming Systems, Languages and Applications, 1986 pp. 1-8.

[Pint88] J. Pinto, "Arquitectura para Sistemas Expertos en Control de Procesos: Reporte Preliminar", Anales de la Conferencia Latinoamericana IEEE, Latincon '88, Buenos Aires Argentina, Abril 1988.

[Stef86] M. Stefik y D.G. Bobrow, "Object Oriented Programming: Themes and Variations", AI Magazine, Vol.6. No. 4, Winter 1986, pp. 40-62.

[Subr85] P.A. Subrahmanyam, "The 'Software Engineering' of Expert Systems: Is Prolog Appropriate?", IEEE Trans. Software Eng., vol. SE-11, pp. 1391-1400.

Capítulo 3

Especificaciones

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

Especificación de Operadores de Control y Desarrollo de Herramientas de Puesta a Punto para un Ambiente de Programación Lógica.*

Nora Szasz

ESLAI - Argentina
Escuela Superior LatinoAmericana de Informática
Casilla de Correo 3193
(1000) Buenos Aires-Argentina

Resumen. Se extiende el conjunto de operadores de control usuales de los lenguajes de Programación Lógica, especificando formalmente los operadores de disyunción, "snip" y corrutinaje, además de la conjunción y negación. La especificación se da en términos de árboles del estilo de los SLD. Se extiende la definición asociando acciones a los nodos de los árboles, lo cual permite especificar comandos de herramientas de puesta a punto de programas. Las herramientas se implementan sobre un meta-intérprete del lenguaje extendido.

*Trabajo realizado en la ESLAI en el marco del proyecto API-2 para la aprobación de la Pasantía del Quinto Semestre de la Licenciatura en Informática bajo la dirección del Dr. J. Vidart.

Capítulo 0.

Introducción.

En el ámbito de la Programación en Lógica se reconoce la carencia de ambientes integrados para el desarrollo de programas. En [CDKMV 88] se propone la construcción de un meta-ambiente para la Programación en Lógica (API-2) en el que un usuario pueda definir lenguajes junto con herramientas asociadas para distintos tipos de aplicaciones.

Un aspecto en que resulta interesante dotar de flexibilidad a los lenguajes de Programación en Lógica es el de los mecanismos de control. Una opción para implementar este objetivo es permitir la definición de lenguajes en que se combinen operadores de control seleccionados de un conjunto predefinido que incluya operadores alternativos además de los convencionales.

En particular, como una versión preliminar de API-2 se ha desarrollado un intérprete para una extensión de Prolog en la que se introducen los siguientes operadores de control:

disyunción (;)
"snip" (^)
conjunción (,)
corrutinaje (\)
negación (NOT)

La primera parte de este trabajo se concentra en la especificación de la semántica de los programas que usan estos operadores representando sus ejecuciones por árboles del tipo SLD [Lloyd 84]. Esto se desarrolla en el capítulo 1, donde la semántica de los operadores es también explicada informalmente.

Por otra parte, cualquier ambiente de programación requiere de un conjunto de herramientas de puesta a punto. Las herramientas de puesta a punto son aquellas piezas de software que permiten observar el comportamiento de un programa durante su ejecución proveyendo facilidades de depuración.

La segunda parte de este trabajo tiene que ver con la especificación e implementación de un conjunto de herramientas de puesta a punto para la extensión de Prolog mencionada arriba. Las herramientas definidas son:

Traza.

Provee un seguimiento paso a paso de la ejecución de un programa. Se exhibe la secuencia de invocaciones de objetivos, asociando a cada uno de éstos un indicador de nivel de profundidad en el *árbol and*, representado mediante numeración (lexicográfica) de las llamadas. La traza puede suspenderse durante cierto rango de la ejecución.

Puntos de espionaje.

Provee puntos de detención del programa definidos por símbolos de predicado. Para cada instancia a ser probada durante la ejecución de un predicado construido con uno de estos símbolos, la ejecución es detenida y la instancia mostrada. Luego la ejecución continúa.

Ejecuciones parciales.

Provee la facilidad de efectuar ejecuciones parciales, donde una falla se transforma en una consulta al usuario quien determinará si el objetivo que falló debe considerarse válido (y en ese caso cuáles son las instancias correspondientes de las variables).

"Debugger".

Provee una forma de detección semi-automática de errores por inconsistencia e incompletitud de programas.

En el capítulo 2 se especifican comandos que implementan las primeras dos facilidades, usando extensiones apropiadas de las definiciones del capítulo 1. Las implementaciones correspondientes a todas las facilidades mencionadas se describen en el capítulo 3, mostrando para las tres primeras cómo fueron montadas sobre el intérprete de API-2 mencionado arriba.

En el capítulo 4 se incluyen conclusiones y posibles extensiones del trabajo. Finalmente, en los apéndices figuran algunas definiciones formales que se ha preferido no incluir en el texto principal para no hacer demasiado tediosa su lectura.

Capítulo 1

Especificación de un meta-intérperte de Programas Lógicos Extendidos con Operadores de Control

Se parte de las nociones clásicas de lenguaje de primer orden, átomo, cláusulas de Horn, lenguaje Prolog, tal como están definidas en [Lloyd 84].

En este capítulo, se presentara una variación de Prolog que introduce nuevos operadores de control. Se presentará la sintaxis y una semántica formal del lenguaje extendido.

definición: *secuencia extendida*

(a) TRUE y FAIL son secuencias extendidas.
(b) todo átomo (predicado Prolog) es una secuencia extendida.
(c) Si S_1 y S_2 son secuencias extendidas, entonces:

$(S_1 \; ; \; S_2)$,
$(S_1 \; \hat{} \; S_2)$,
$(S_1 \; , \; S_2)$,
$(S_1 \; \backslash \; S_2)$ y
NOT S_1

son secuencias extendidas.

Los simbolos {; ^ , \ NOT} serán llamados *operadores*, y el conjunto formado por ellos será llamado Θ.
■

definición: *goal extendido*

Un goal extendido es

$\leftarrow S$

donde S es una secuencia extendida.
■

definición: *cláusula extendida*

Una cláusula extendida es

$A \leftarrow S.$

donde A es un átomo, llamado la *cabeza* de la cláusula, y S una secuencia extendida, llamada *resolvente* de la cláusula.
■

definición: *programa extendido*

Un programa extendido es una lista de cláusulas extendidas.
■

notación:

Los átomos se denotarán con las primeras letras del alfabeto en mayúscula (A, B, A_1, A_2, etc.).

Los goals extendidos se denotarán en la forma $\leftarrow S, \leftarrow S_1$, o bien directamente G, G_1.

Se notará *secuencia*, *goal*, *cláusula* y *programa* por sus correspondientes *extendidos*.

semántica informal de los programas:

; es el operador *or*. El significado de $A \leftarrow (S_1;S_2)$. es: *para satisfacer* A, *debe satisfacerse* S_1 *o* S_2. Es equivalente a:

$A \leftarrow S_1.$
$A \leftarrow S_2.$

^ es un operador que representa una variedad "débil" del operador *cut* (!) de Prolog. Su efecto es el de un *snip* [Arity] sobre el primer operando. Su efecto es no reintentar el primer operando durante el "backtracking" en caso de falla del segundo.

El equivalente Prolog de A^B es

A^B← P.
P← A, !, B.

, es el operador *and* de Prolog.

\ es un operador de *corrutinaje*, que causa la *ejecución paralela* de los dos goals que él conecta. Su efecto es similar al recorrido *a lo ancho* del árbol *and* generado por el goal a demostrar.

NOT es el operador *not* de Prolog.

semántica formal de los programas:

La formalización de la semántica de los programas se hará representando todas las posibles evaluaciones de cada goal mediante un árbol.

Más precisamente, para cada programa se dará una función de goals en árboles, de forma tal que toda derivación de un goal dado en el programa estará representada por una rama del árbol que corresponde a dicho goal.

definición: *árbol de secuencias*

(a) Si S es una secuencia, entonces
$\langle S,[] \rangle$
es un árbol de secuencias (donde [] denota la lista vacia).

(b) Si S es una secuencia, $t_1,\ldots,t_n$ son árboles de secuencias y $\theta_1,\ldots,\theta_n$ son sustituciones para variables de S tales que, para todo i, las variables de θ_i no ocurren en t_i entonces
$\langle S,[\langle t_1,\theta_1\rangle,\ldots,\langle t_n\theta_n\rangle] \rangle$
es un árbol de secuencias.

■

En esta definición de árbol los subárboles que componen a cada árbol están ordenados [Knuth 68]. Además figura una sustitución asociada a cada subárbol, cuya intención es representar una etiqueta en la arista.

El orden de los subárboles se debe a que en Prolog las cláusulas de un programa están ordenadas, lo que debe respetarse en la elección de las mismas. Por medio de listas es muy simple hacer explícito este hecho.

En adelante se notará $\mathbb{S}$ al conjunto de las secuencias, $\mathbb{G}$ al de los goals, $\mathbb{P}$ al de los programas y $\mathbb{T}$ al de los árboles.

La sustitución identidad se notará id.

definición: *semántica de programas*

Para cada programa P se define la función $\tau(P)$, que a cada goal asocia un *árbol de secuencias* o sea:

$$\tau : \mathbb{P} \longrightarrow (\mathbb{G} \longrightarrow \mathbb{T})$$

Sea P un programa, entonces:

(a) $\tau(P)(\leftarrow \text{TRUE}) = \langle \text{TRUE}, [] \rangle$

(b) $\tau(P)(\leftarrow \text{FAIL}) = \langle \text{FAIL}, [] \rangle$

(c) $\tau(P)(\leftarrow A) = \langle A, \mathscr{L} \rangle$
donde
$$\mathscr{L} = [\langle \tau(P)(\leftarrow S_1\theta_1), \theta_1 \rangle, \ldots, \langle \tau(P)(\leftarrow S_n\theta_n), \theta_n \rangle]$$
si la lista $[B_1 \leftarrow S_1, \ldots, B_n \leftarrow S_n]$ es la sublista no vacía de P tal que A y B_i unifican con mgu θ_i y
$$\mathscr{L} = [\langle \langle \text{FAIL}, [] \rangle, id \rangle]$$
si no hay cláusulas que cumplan esto.

(d) $\tau(P)(\leftarrow (A;B)) = \langle (A;B), [\langle \tau(P)(\leftarrow A), id \rangle, \langle \tau(P)(\leftarrow B), id \rangle] \rangle$

(e) $\tau(P)(\leftarrow (A \wedge B)) = \langle (A \wedge B), [\langle \text{SNP}(\tau(P)(\leftarrow A), B, P), id \rangle] \rangle$

(f) $\tau(P)(\leftarrow (A,B)) = \langle (A;B), [\langle \text{CNJ}(\tau(P)(\leftarrow A), B, P), id \rangle] \rangle$

(g) $\tau(P)(\leftarrow (A \setminus B)) = \text{COR}(\tau(P)(\leftarrow A), \tau(P)(\leftarrow B))$

(h) $\tau(P)(\leftarrow \text{NOT } A) = \text{NEG}(\tau(P)(\leftarrow A))$

■

Una demostración de $\leftarrow S$ en P corresponde a una *rama exitosa*, esto es una rama que termine con el átomo TRUE. Las ramas que terminan con el átomo FAIL son *ramas de falla finita*, y aquellas que son de longitud infinita son las correspondientes a *derivaciones infinitas*. A continuación se define la función booleana scs, la cual, aplicada a un árbol, devuelve *true* si haciendo un recorrido del mismo en profundidad se encuentra una rama exitosa.

$$scs : \mathbb{T} \longrightarrow (true, false)$$

$$scs(S,[]) = (S = \text{TRUE})$$

$$scs(S,[\langle t_1, \theta_1 \rangle | \mathscr{L}]) = scs(t_1) \text{ or } scs(\langle S, \mathscr{L} \rangle)$$

■

La interpretación de los casos (a) y (b) en la definición de la función τ, corresponde a la interpretación usual de los goals TRUE y FAIL: el goal $\leftarrow$ TRUE siempre es exitoso y el goal $\leftarrow$ FAIL siempre falla.

Para la evaluación de un átomo, se consideran todas las cláusulas del programa cuyas cabezas unifican con dicho átomo y para cada una de estas, el árbol correspondiente a su resolvente con la sustitución adecuada. En el caso en que no haya cláusulas que cumplan con esta propiedad, habrá una rama de fallo.

La sustitución asociada a un subárbol es la unificación de salida entre el goal y la cabeza de cla cláusula que se utiliza en el paso de inferencia.

Se han usado las funciones CNJ, SNP , COR y NEG para definir los árboles. Estas son funciones que "combinan" los árboles de los goals conectados por estos operadores de forma tal que el nuevo árbol refleja la semántica del goal resultante. Se definen, entonces, naturalmente, por inducción sobre los árboles.

Para evaluar la *conjunción* , de dos goals, es necesario evaluar inicialmente al primero y si esta evaluación fue exitosa, evaluar al segundo. En términos de Prolog, esto significa *por cada demostración del primer goal, intentar todas las posibles demostraciones del segundo con las unificaciones heredadas de la demostración del primero.*

La función CNJ hace exactamente eso: dado el árbol correspondiente al primer operando, *reemplaza* cada hoja TRUE por el árbol correspondiente al segundo operando, al que previamente se le han aplicado todas las substituciones que se hicieron en esa (rama->demostracion.

La definición de la función CNJ es la siguiente:

$$CNJ:\mathbb{T} \times \mathbb{S} \times \mathbb{P} \longrightarrow \mathbb{T}$$

$$CNJ(\langle TRUE,[]\rangle,S,P) = \tau(P)(\leftarrow S)$$

$$CNJ(\langle FAIL,[]\rangle,S,P) = \langle FAIL,[]\rangle$$

$$CNJ(\langle R,[\langle t_1,\theta_1\rangle,\ldots,\langle t_n,\theta_n\rangle]\rangle,S,P) = \langle R,[\langle CNJ(t_1,S\theta_1,P),\theta_1\rangle,\ldots,\langle CNJ(t_n,S\theta_n,P),\theta_n\rangle]\rangle$$

■

El operador ^ corresponde al operador ! en forma local entre sus dos operandos. En términos de Prolog, A^B significa *demuestre A y luego demuestre B, sin reintentar otra demostración alternativa de A*. Corresponde a la "fijación" de la primera demostración (con su correspondiente unificación asociada) e implica (en términos de árboles) la *poda* del primer árbol, una vez obtenido el primer éxito.

Para la definición del operador ^, deberá proveerse una función similar a CNJ, pero tal que, una vez obtenida la primera demostración de su primer operando, *pode* al árbol correspondiente a éste, *eliminando al resto de sus ramas*, y *reemplace* la hoja TRUE por el árbol correspondiente a su segundo operando, luego de la sustitución adecuada.

Formalmente

$$SNP:\mathbb{T} \times \mathbb{S} \times \mathbb{P} \longrightarrow \mathbb{T}$$

$$SNP(\langle R,[]\rangle,S,P) = \text{IF } (R = TRUE) \text{ THEN } \tau(P)(\leftarrow S) \text{ ELSE } \langle FAIL,[]\rangle$$

$$SNP(\langle R,[\langle t_1,\theta_1\rangle|\mathcal{L}]\rangle,S,P) = \text{IF } scs(t_1) \text{ THEN } \langle R,[\langle SNP(t_1,S\theta_1,P),\theta_1\rangle]\rangle \text{ ELSE } ADD(\langle t_1,\theta_1\rangle,SNP(\langle R,\mathcal{L}\rangle,S,P))$$

■

El operador de *corrutinaje* \ corresponde a la ejecución como corrutinas de los goals que él conecta. Esto es: para demostrar A\B se ejecutarán en forma simultánea pasos de derivación de A y B hasta que uno de ellos o bien se demuestre (en cuyo caso se continúa con el otro) o bien falle (en cuyo caso se reintenta con otra derivación).

La función COR *combina* los árboles de sus operandos de manera tal de proveer todas las posibles combinaciones de derivaciones simultáneas de ambos, de la siguiente forma:

$$COR:\mathbb{T} \times \mathbb{T} \rightarrow \mathbb{T}$$

$$COR(\langle FAIL,[]\rangle,\langle R,\mathscr{E}\rangle) = \langle FAIL \backslash R,[\langle\langle FAIL,[]\rangle,id\rangle]\rangle$$

$$COR(\langle R,\mathscr{E}\rangle,\langle FAIL,[]\rangle) = \langle R \backslash FAIL,[\langle\langle FAIL,[]\rangle,id\rangle]\rangle$$

$$COR(\langle TRUE,[]\rangle,\langle R,\mathscr{E}\rangle) = IF\ (R \neq FAIL)\ THEN\ \langle TRUE \backslash R,[\langle R,\mathscr{E}\rangle]\rangle$$

$$COR(\langle R,\mathscr{E}\rangle,\langle TRUE,[]\rangle) = IF\ (R \neq FAIL)\ THEN\ \langle R \backslash TRUE,[\langle R,\mathscr{E}\rangle]\rangle$$

$$COR(\langle A,\mathscr{E}_A\rangle,\langle B,\mathscr{E}_B\rangle) = IF\ UNIF(\mathscr{E}_A,\mathscr{E}_B) \neq [] \ THEN\ \langle A \backslash B,UNIF(\mathscr{E}_A,\mathscr{E}_B)\rangle \ ELSE\ \langle A \backslash B,[\langle\langle FAIL,[]\rangle,id\rangle]\rangle$$

donde se define UNIF de la siguiente forma:

$$UNIF([],\mathscr{E}) = []$$

$$UNIF([\langle t,\theta\rangle|\mathscr{E}_1],\mathscr{E}_2) = APPEND(UNIF_1(\langle t,\theta\rangle,\mathscr{E}_2),UNIF(\mathscr{E}_1,\mathscr{E}_2))$$

$$UNIF_1(\langle t,\theta\rangle,[]) = []$$

$$UNIF_1(\langle t,\theta\rangle,[\langle u,\sigma\rangle|\mathscr{E}]) = IF\ compatibles?(\theta,\sigma)\ THEN\ [\langle t\delta \backslash u\delta,\theta\delta\rangle,|UNIF_1(\langle t,\theta\rangle,\mathscr{E})] \ ELSE\ UNIF_1(\langle t,\theta\rangle,\mathscr{E})$$

donde la función *compatibles?* es booleana e indica si existe una única sustitución δ que tenga el efecto de sus dos argumentos.
■

Para la evaluación de la *negación* de un goal, es preciso tratar de demostrar el goal no negado y, sólo en el caso de no poder hacerlo se habrá demostrado la tal negación. En términos de Prolog, el resultado de la evaluación de NOT S es: *si puede demostrar* S, *entonces* NOT S *falla*, *si no* NOT S *tiene éxito*.

La función NEG, entonces deberá buscar la primera hoja TRUE en el árbol de su operando y, de encontrarla, *sustituir* esta por FAIL y *podar* el resto del árbol (no es necesario buscar demostraciones alternativas del goal que se negó), si tal hoja TRUE no existe en el árbol, se colgará directamente de la raiz del árbol dado, una hoja TRUE.

$$NEG:\mathbb{T} \rightarrow \mathbb{T}$$

$$NEG(\langle S,\mathscr{E}\rangle) = IF\ SCS(S)\ THEN\ \langle NOT\ S,[\langle CNJ(\langle S,\mathscr{E}\rangle,FAIL,P),id\rangle]\rangle \ ELSE\ \langle NOT\ S,[\langle TRUE,id\rangle]\rangle$$

■

La función ADD construye un árbol a partir de dos dados, colocando al primero como primera rama del segundo.

$$ADD:(\mathbb{T} \times \mathbb{S}) \rightarrow \mathbb{T} \rightarrow \mathbb{T}$$

$$ADD(\langle t,\theta\rangle,\langle S,\mathscr{E}\rangle) = \langle S,[\langle t,\theta\rangle|\mathscr{E}]\rangle$$

■

De esta forma, queda completa la definición de la semántica de un programa en términos de árboles *or*, donde cada rama de un árbol correspondiente a un goal es una derivación de dicho goal en el tal programa.

Además, se puede asociar una sustitución a cada rama del arbol, que será la composición de las sustituciones asociadas a todos los arcos que componen esa rama. Así cada *respuesta* exitosa del sistema al goal será la aplicación de la sustitución asociada a cada rama exitosa al goal dado.

El recorrido del arbol se hará *en profundidad*, de la misma forma que se hace en Prolog, con lo que, un programa y un goal que sólo usen los operadores originales de Prolog obtendrán el mismo comportamiento que tendrían si se usara un intérprete Prolog tradicional. Es más, puede demostrarse que el árbol $\tau(P)(G)$ para un tal programa P y un tal goal G es el mismo que el arbol *SLD* para P∪{G} (Lloyd 84), a menos del etiquetado de los arcos con sustituciones.

No se incluye hasta el momento la semántica del operador ! (*cut*) ya que la variacion con que se trabajo no lo proveía.

De todas formas su definición puede darse con este formalismo, reformulando la regla (c) de la definición de la función τ.

(c) $\tau(P)(\leftarrow A) = \langle A,\mathscr{E}\rangle$
donde
Sea $\mathbb{C} = [B_1 \leftarrow S_1,\ldots,B_n \leftarrow S_n]$ la sublista de P tal que A y B_i unifican con mgu θ_i.
entonces,
Si $\mathbb{C} = []$ entonces $\mathscr{E} = [\langle\langle \text{FAIL},[]\rangle,id\rangle]$.
Si $\mathbb{C} \neq []$ y $\exists k\ (1 \leq k \leq n)$ tal que S_k es el primero que contiene una secuencia extendida "S_{k1},!,S_{k2}" con $scs(S_{k1}) = true$,
entonces
$\mathscr{E} = [\langle\tau(P)(\leftarrow S_1\theta_1),\theta_1\rangle,\ldots,\langle snp(\tau(P)(\leftarrow S_{k1}\theta_k),S_{k2}\theta_k,P),\theta_k\rangle]$
En otro caso
$\mathscr{E} = [\langle\tau(P)(\leftarrow S_1\theta_1),\theta_1\rangle,\ldots,\langle\tau(P)(\leftarrow S_n\theta_n),\theta_n\rangle]$
■

capítulo 2.

Especificación de herramientas de puesta a punto para programas lógicos extendidos.

2.1 Introducción.

La especificación del capítulo anterior es suficiente para representar cualquier ejecución de un programa lógico extendido.

Las herramientas de puesta a punto esencialmente proveen información sobre el desarrollo de la ejecución de un programa. La información exhibida debe poder ser seleccionada por el usuario. Eventualmente, el usuario podrá también alterar el curso de la ejecución.

Para cumplir este propósito se debe asociar a la representación de la ejecución la información disponible para ser exhibida. Esto se hace en la sección 2.2 donde se extiende la definición del árbol de ejecución, dando una numeración de los goals que corresponde al orden en que serán invocados.

Por otro lado, deben definirse comandos cuyo efecto sea seleccionar la información a ser exhibida. Para ello, en la sección 2.3 se extiende nuevamente la definición del árbol de ejecución asociando acciones a los nodos. En 2.4 se definen comandos cuyo efecto es dado por funciones que modifican los árboles con acciones.

2.2 Arboles numerados.

definición: *secuencia numerada* y *p-número*

Una secuencia numerada es una secuencia donde cada componente de la misma está acompañado de una cadena de caracteres de la forma $n_1.n_2...n_m$, $(m \geq 1)$, donde los n_i son numeros naturales.

Tales cadenas seran llamadas *p-números*.

■

definición: *goal numerado*

Un goal numerado es aquel que se obtiene a partir de una secuencia numerada.

■

ejemplos: ← $A_{(a,c)-1.9.2}$
←$(A_{(y)-2.4}\&(\text{TRUE}\hat{}B_{(a,c)-9.1})$
son goals numerados.

convenciones:

A partir de aquí, a los goals numerados, se les agregará el sufijo num, quedando de la forma ← S_{num},← S_{num1}, o directamente G_{num}, G_{num1}, etc.

definición: *árbol de secuencias numeradas*

(a) Si Snum es un secuencia numerada, entonces
$\langle Snum,[]\rangle$
es un árbol de secuencias numeradas.

(b) Si Snum es un secuencia $,t_1,\ldots,t_n$ son árboles de secuencias numeradas y $\theta_1,\ldots,\theta_n$ son sustituciones para variables de Snum que no ocurren en t_i entonces
$\langle Snum,[\langle t_1,\theta_1\rangle,\ldots,\langle t_n\theta_n\rangle]\rangle$
es un árbol de secuencias numeradas.

■

notación:

Se notará *n-secuencia*, *n-goal* y *n-àrbol* por sus correspondientes *numerados*, y los conjuntos formados por ellos se notarán $\mathbb{S}_n$, $\mathbb{G}_n$ y $\mathbb{T}_n$ respectivamente.
El conjunto de los p-números será notado $\mathbb{N}_p$.

definición: *numeración de goals*

Hay dos funciones, NUMINIT y NUM, las cuales sirven para numerar secuencias y se definen de la siguiente forma:

NUMINIT: $\mathbb{S} \longrightarrow \mathbb{S}_n$

NUMINIT(TRUE) = TRUE-1
NUMINIT(FAIL) = FAIL-1
NUMINIT(A) = A-1
NUMINIT((S1;S2)) = S1-1;S2-2
NUMINIT((S1^S2)) = S1-1^S2-2
NUMINIT((S1,S2)) = S1-1,S2-2
NUMINIT((S1;S2)) = S1-1;S2-2
NUMINIT((S1\S2)) = S1-1\S2-2
NUMINIT(NOT S) = (NOT S)-1

■

NUM: $\mathbb{S} \times \mathbb{N}_p \longrightarrow \mathbb{S}_n$

NUM(TRUE,K) = TRUE-K.1
NUM(FAIL,K) = FAIL-K.1
NUMINIT(A,K) = A-K.1
NUMINIT((S1;S2),K) = S1-K.1;S2-K.2
NUM((S1^S2),K) = S1-K.1^S2-K.2
NUM((S1,S2),K) = S1-K.1,S2-K.2
NUM((S1;S2),K) = S1-K.1;S2-K.2
NUM((S1\S2),K) = S1-K.1\S2-K.2
NUM(NOT S,K) = (NOT S)-K.1

■

La numeración de los goals surge de la necesidad de poder representar la relación existente entre los átomos que van apareciendo para ser demostrados en la ejecución de un programa extendido.

La función NUMINIT se aplicará al goal a ser demostrado y, como puede verse, lo que hace es: si el goal es un átomo, se numera con el número 1, y si es un goal compuesto otros dos, éstos se numeran con los números 1 y 2.

La función NUM recibe como parámetros a un p-número K y a una secuencia y devuelve la secuencia numerada obtenida de numerar a los componentes de la primera con los p-números K.1 y K.2, si ésta es compuesta o K.1 si nó .

La relación que quiere representarse mediante esta numeración de los goals es el orden de invocación que surge en la demostración de un goal determinado. Naturalmente, el goal a ser demostrado estará numerado con el número 1 (o 1 y 2 si es compuesto), y cada vez que un átomo numerado A-K se derive en otros B⟨op⟩C, éstos se numerarán de la forma B-K.1⟨op⟩C-K.2.

Por otro lado, se quiere representar en forma explícita en los árboles de secuencias la relación inversa de la anterior, esto es: el átomo del que es derivado cada goal que se demuestra. Para esto, cada vez que se deriva un goal G a partir de un átomo A, se agrega al nodo correspondiente a esta derivación el átomo A-o.K al final de G. Así, cada nodo que comience con un átomo de la forma A-o.K, corresponderá a la finalización de la prueba de A.

Para formalizar estos conceptos, se define la función τ_n, que es una extensión de la función τ definida en el capítulo anterior.

definición: *semántica "numerada" de programas*

Para cada programa P se define la función $\tau_n(P)$, la cual aplicada a un goal devuelve un árbol de secuencias numerado de la siguiente forma:

$\tau_n : \mathbb{P} \longrightarrow (\mathbb{G}_n \longrightarrow \mathbb{T}_n)$

Sea P un programa, entonces:

(a) $\tau_n(P)(\leftarrow \text{TRUE-K}) = \langle \text{TRUE-K}, [] \rangle$

(b) $\tau_n(P)(\leftarrow \text{FAIL-K}) = \langle \text{FAIL-K}, [] \rangle$

(c) $\tau_n(P)(\leftarrow \text{A-K}) =$ IF K = 0.J
THEN $\langle \text{A-K}, [\langle\langle \text{TRUE-J}, [] \rangle, id \rangle] \rangle$
ELSE $\langle A, \mathcal{L} \rangle$

donde
$\mathcal{L} = [\langle \text{CNJn}(\tau_n(P)(\leftarrow \text{NUM}(S_1\theta_1, K)), A\text{-o.K}, P), \theta_1 \rangle, \ldots, \langle \text{CNJn}(\tau_n(P)(\leftarrow \text{NUM}(S_n\theta_n, K)), A\text{-o.K}, P), \theta_n \rangle]$
si la lista $[B_1 \leftarrow S_1, \ldots B_n \leftarrow S_n]$ es la sublista no vacía de P tal A y B_i unifican con mgu θ_i.
$\mathcal{L} = [\langle\langle \text{FAIL-K}, [] \rangle, id \rangle]$
en otro caso.

(d) $\tau_n(P)(\leftarrow (A;B)\text{-K}) = \langle (A;B)\text{-K}, [\langle \tau_n(P)(\leftarrow A\text{-K.1}), id \rangle, \langle \tau_n(P)(\leftarrow B\text{-K.2}), id \rangle] \rangle$

(e) $\tau_n(P)(\leftarrow (A \hat{} B)\text{-K}) = \langle (A \hat{} B)\text{-K}, [\text{SNP}(\tau_n(P)(\leftarrow A\text{-K.1}), B\text{-K.2}, P), id \rangle] \rangle$

(f) $\tau_n(P)(\leftarrow (A,B)\text{-K}) = \langle (A,B)\text{-K}, [\langle \text{CNJn}(\tau_n(P)(\leftarrow A\text{-K.1}), B\text{-K.2}, P), id \rangle] \rangle$

(g) $\tau_n(P)(\leftarrow (A \backslash B)\text{-K}) = \text{CORn}(\tau_n(P)(\leftarrow A\text{-K.1}), \tau_n(P)(\leftarrow B\text{-K.2}), K)$

(h) $\tau_n(P)(\leftarrow (\text{NOT } A)\text{-K}) = \text{NEGn}(\tau_n(P)(\leftarrow A\text{-K}))$

■

Se define la función scsn del mismo modo que la función scs de la sección anterior, de la forma:

$scsn : \mathbb{T}_n \longrightarrow \langle true, false \rangle$

$scsn(S\text{-k}, []) = (S = \text{TRUE})$

$scsn(S\text{-k}, [\langle t_1, \theta_1 \rangle ! \mathcal{L}]) = scsn(t_1)$ *or* $scsn(\langle S\text{-K}, \mathcal{L} \rangle)$

Para construir el árbol numerado correspondiente a un goal ← S en un programa P dado, debe computarse la función $\tau_n(P)(\leftarrow \text{NUMINIT}(S))$.

■

Se notará que la definición de la función τ_n es muy similar a la de la función τ definida en el capítulo anterior. De hecho, hay sólo dos diferencias fundamentales entre los árboles generados por τ y, por τ_n, aparte del hecho obvio de que una trabaja con secuencias sin numerar y la otra con secuencias numeradas. Tales diferencias están en el caso (c), y corresponden a las relaciones que se quisieron hacer explícitas en el árbol de derivaciones de un goal en un programa.

La regla (c) es la encargada de las derivaciones de programa propiamente dicho, pues es éste el único caso en que se *consulta* al programa para extraer de allí los resolventes de las cláusulas que unifican con un átomo dado. Es aquí entonces donde se introduce la relación antecedente-consecuente que se quería hacer explícita en el árbol generado por un goal: se agrega al final de cada rama correspondiente a las derivaciones del tal átomo, el mismo átomo *cerodo* para indicar que éste es consecuencia de lo recién demostrado.

El caso atómico debe también considerar ahora que el átomo a derivar corresponda a la finalización de la demostración de un goal. Para ello, según se definió, sólo basta con ver si el p-número que acompaña al átomo comienza con o.

Las funciones que se utilizan para definir la conjunción, snip, corrutinaje y negación son muy parecidas a las definidas anteriormente para los mismos propósitos. Sus definiciones se dan a continuación.

$$\text{CNJn}:\mathbb{T}n \times \mathbb{S}n \times \mathbb{P} \longrightarrow \mathbb{T}n$$

$$\text{CNJn}(\langle \text{TRUE-K},[]\rangle,\text{Snum},P) = \tau(P)(\leftarrow \text{Snum})$$

$$\text{CNJn}(\langle \text{FAIL-K},[]\rangle,\text{Snum},P) = \langle \text{FAIL-K},[]\rangle$$

$$\text{CNJn}(\langle R\text{-K},[\langle t_1,\theta_1\rangle,\ldots,\langle t_n,\theta_n\rangle]\rangle,\text{Snum},P) = \langle R\text{-K},[\langle \text{CNJn}(t_1,\text{Snum}\theta_1,P),\theta_1\rangle,\ldots,\langle \text{CNJn}(t_n,\text{Snum}\theta_n,P),\theta_n\rangle]\rangle$$

■

$$\text{SNPn}:\mathbb{T}n \times \mathbb{T}n \longrightarrow \mathbb{T}n$$

$$\text{SNPn}(\langle R\text{-K},[]\rangle,\text{Snum},P) = \text{IF } (R = \text{TRUE}) \text{ THEN } \tau(P)(\leftarrow \text{Snum}) \text{ ELSE } \langle \text{FAIL-K},[]\rangle$$

$$\text{SNPn}(\langle R\text{-K},[\langle t_1,\theta_1\rangle|\mathcal{L}]\rangle,\text{Snum},P) = \text{IF SCSn}(t_1) \text{ THEN } \langle R\text{-K},[\langle \text{SNPn}(t_1,\text{Snum},P),\theta_1\rangle]\rangle \text{ ELSE ADD}(\langle t_1,\theta_1\rangle,\text{SNPn}(\langle R\text{-K},\mathcal{L}\rangle,\text{Snum},P))$$

■

$$\text{COR}_n : \mathbb{T}_n \times \mathbb{T}_n \longrightarrow \mathbb{T}_n$$

$$\text{COR}_n(\langle \text{FAIL-K}, [] \rangle, \langle R\text{-J}, \mathscr{L} \rangle) = \langle \text{FAIL-K} \backslash R\text{-J}, [\langle \langle \text{FAIL-K}, [] \rangle, id \rangle] \rangle$$

$$\text{COR}_n(\langle R\text{-K}, \mathscr{L} \rangle, \langle \text{FAIL-J}, [] \rangle) = \langle R\text{-K} \backslash \text{FAIL-J}, [\langle \langle \text{FAIL-J}, [] \rangle, id \rangle] \rangle$$

$$\text{COR}_n(\langle \text{TRUE-K}, [] \rangle, \langle R\text{-J}, \mathscr{L} \rangle) = \text{IF } (R \neq \text{FAIL}) \text{ THEN } \langle \text{TRUE-K} \backslash R\text{-J}, [\langle R\text{-J}, \mathscr{L} \rangle] \rangle$$

$$\text{COR}_n(\langle R\text{-K}, \mathscr{L} \rangle, \langle \text{TRUE-J}, [] \rangle) = \text{IF } (R \neq \text{FAIL}) \text{ THEN } \langle R\text{-K} \backslash \text{TRUE-J}, [\langle R\text{-K}, \mathscr{L} \rangle] \rangle$$

$$\text{COR}_n(\langle A, \mathscr{L}_A \rangle, \langle B, \mathscr{L}_B \rangle, K) = \text{IF } \text{UNIF}(\mathscr{L}_A, \mathscr{L}_B) \neq \emptyset \text{ THEN } \langle (A \backslash B)\text{-K}, \text{UNIF}(\mathscr{L}_1, \mathscr{L}_2) \rangle \text{ ELSE } \langle (A \backslash B)\text{-K}, [\langle \langle \text{FAIL-K}, [] \rangle, id \rangle] \rangle$$

■

Donde la función UNIF es la misma que la que se definió en el capítulo anterior, junto con la definición de COR.

$$\text{NEG}_n : \mathbb{T}_n \longrightarrow \mathbb{T}_n$$

$$\text{NEG}_n(\langle R\text{-K}, \mathscr{L} \rangle) = \text{IF } \text{SCS}_n(R\text{-K}) \text{ THEN } \langle \text{NOT } R\text{-K}, [\langle \text{SNP}_n(\langle R\text{-K}.1, \mathscr{L} \rangle, \text{FAIL-K}, P), id \rangle] \rangle \text{ ELSE } \langle \text{NOT } R\text{-K}, [\langle \text{TRUE-K}, id \rangle] \rangle$$

■

Así se completa la definición de la función τ_n, que de una forma casi idéntica a τ, define la semántica de los programas haciendo explícito el orden de invocación de los objetivos a demostrar en cada derivación de un goal dado en un programa. Es más, esta numeración de los goals que van apareciendo en el árbol corresponde exactamente a la representación del *árbol and* de una ejecución mediante una numeracion lexicográfica de sus nodos.

2.3 Arboles con acciones.

Se introducen en esta sección nuevas modificaciones a los árboles definidos anteriormente y a la función τ_n. Tales modificaciones corresponden a la introducción de acciones asociadas a los nodos de los árboles, con el fin de poder definir algunas de las herramientas de puesta a punto sobre el árbol generado por la función τ_n.

Las acciones que se asociarán a los nodos serán las encargadas de mostrar algún tipo de información sobre el estado de la ejecución en ese momento, la cual dependerá de la herramienta que se esté usando.

Inicialmente, todos los nodos de un árbol tendrán asociada la acción vacía (NOP), y los distintos comandos invocados durante la ejecución, irán modificando las tales acciones y generando nuevos árboles. Para poder formalizar estos conceptos, se introducen las siguientes definiciones:

definición: *acciones*

Se define por extensión el conjunto de acciones
$\mathbb{A} = \langle \text{NOP}, \text{EXH}, \text{EXH-FAIL}, \text{EXH-SLV}, \text{EXH-NOSLV} \rangle$

definición: *árbol con acciones*

(a) Si Snum es un secuencia numerada y acc $\in \mathbb{A}$, entonces
$\langle Snum,[],acc\rangle$
es un árbol de acciones.

(b) Si Snum es un secuencia $,t_1,\ldots,t_n$ son árboles de acciones, $\theta_1,\ldots,\theta_n$ son sustituciones para variables de Snum tales que, para todo i, las variables de θ_i no ocurren en t_i y acc $\in \mathbb{A}$, entonces
$\langle Snum,[\langle t_1,\theta_1\rangle,\ldots,\langle t_n\theta_n\rangle],acc\rangle$
es un árbol de acciones.

El conjunto de los árboles de acciones se notará $\mathbb{T}_a$.
■

Se define la función $\tau_{na}:\mathbb{P} \rightarrow (\mathbb{G} \rightarrow \mathbb{T}_a)$ en forma similar a la función τ_n asociando a cada nodo del árbol la acción NOP. La definición completa de la misma se halla en el Apéndice I.

2.4 Comandos de puesta a punto.

A continuación se da una lista con los comandos que se quieren proveer. Algunos de estos comandos se corresponden con los mencionados en la introducción del trabajo. Tales comandos son:

(a) TRACE
(b) SKIP(nc)
(c) JUMP(nc,p-num)
(d) GOPRED(nc,pred)
(e) DEMON(nc,cond)
(f) SPY(pred).

Cada posible comando será una función de árboles con acciones en árboles con acciones. Cada una de estas funciones modificará al árbol, asociando diferentes acciones a los nodos, que dependerán del comando elegido. Una sesión estará compuesta por un goal a demostrar y una lista de comandos. Las funciones asociadas a estos comandos se compondrán sucesivamente, y el árbol resultante de la sesión será el obtenido de aplicar esta composición al árbol generado por el goal a demostrar. Esto es, para un goal G y una lista de comandos [c1,..,cn] el árbol obtenido será $c_n(c_{n-1}(\ldots(c_1(\tau_{na}(G)))\ldots))$.

Los comandos no podrán ser invocados en cualquier orden: por ejemplo SKIP(n) (suspensión de la traza durante la demostración del goal asociado al nodo n) sólo podrá ser invocado después de la invocación de un TRACE. Para poder imponer esta noción de secuenciamiento cada función de comando tiene asociada una precondición, que impone requisitos sobre las acciones asociadas al árbol al cual la función se aplica.

Las especificaciones formales de las funciones asociadas a los comandos consisten básicamente en recorridos apropiados de los árboles de goals. A continuación, se describen los comandos y las funciones de manera informal, figurando las especificaciones formales en el Apéndice II.

TRACE: este comando provee un seguimiento paso a paso de la ejecución de un programa. Para poder representar tal facilidad en el árbol de la ejecución, debe asociarse a todos los nodos del mismo una acción que muestre el estado actual del programa. La información asociada no es la misma para todos los nodos: por ejemplo, hay nodos de falla, nodos de culminación de una prueba, nodos de reemplazo de un átomo por alguno de sus resolventes, etc. Todos estos casos son considerados en la especificación incluida en el Apéndice II, donde se explican en detalle.

SKIP(n): este comando provoca la suspensión de la traza durante la demostración del goal asociado al nodo identificado con el p-número n.

De acuerdo a la definición de τ_n, la demostración de un goal G-K comienza en el nodo que tiene tal goal asociado y termina con la primera aparición de un nodo cuyo goal sea G-0.K. Puede darse el hecho, sin embargo, de que no se pueda demostrar al tal goal, lo cual se pone en evidencia si no existe ningún nodo con goal asociado G-0.K en los subárboles del árbol cuya raíz es G-K.

Lo que debe hacerse entonces es un recorrido en profundidad del árbol al cual se aplica la función SKIP, buscar al primer nodo con p-número asociado n, el cual debe existir para que el comando tenga efecto, buscar en el árbol cuya raiz es ese nodo al primer nodo con p-número 0.n y asociar a todos los nodos que se recorren desde el nodo n al nodo 0.n la acción NOP. En el caso de no encontrar un nodo 0.n en ese árbol, se asociará a todos los nodos del árbol la acción NOP y se mostrará la falla de la demostración del nodo n.

JMP(n,m): provoca la suspensión de la traza entre el nodo n y el nodo con p-número m. Para que la invocación de este comando tenga efecto debe existir el nodo n en el árbol. Se asocia a los nodos entre n y m (en orden de profundidad primero) la acción NOP. Si no existe un nodo con p-número m, la operación descripta se aplica a todo nodo del árbol que siga a n.

GPRED(n,p): provoca la suspensión de la traza entre el nodo n y el comienzo de la demostración de alguna instancia del predicado p. Para que la invocación de este comando tenga efecto debe existir el nodo n en el árbol. Se asocia a los nodos entre n y el primero siguiente a éste con nombre de goal p en orden de profundidad primero la acción NOP. Si no existe un nodo con goal asociado p, la operación descripta se aplica a todo nodo del árbol que siga a n.

DEMON(n,cond): La activación de este comando provoca la suspensión de la traza entre el nodo n y el primer nodo cuyo goal asociado cumpla la condición cond. El nodo n debe existir en el árbol. Se asocia a los nodos entre n y el primero siguiente a éste cuyo goal cumpla cond (en orden de profundidad primero) la acción NOP. Si no existe un nodo cuyo goal cumpla cond, la operación descripta se aplica a todo nodo del árbol que siga a n.

SPY(p): se muestran todos los nodos del árbol que contengan al predicado p en su goal asociado o sean resolventes de una instancia del predicado p. Esto último equivale a que el nodo sea raiz de un subárbol de un árbol cuya raiz contiene a p.

Capitulo 3.

Montaje de herramientas de puesta a punto sobre un meta-intérprete de programas lógicos extendidos.

3.1 El meta-intérprete.

Un meta-intérprete para un lenguaje es un intérprete para dicho lenguaje escrito en ese lenguaje.

La forma más simple de un meta-intérprete para Prolog es, obviamente:

```
SOLVE(G) :- G.
```

La semántica de los operadores usados en el lenguaje puede hacerse explícita como en el caso de la conjunción, representada por

```
SOLVE(true).
SOLVE(G) :- CLAUSE(G,C),SOLVE(C).
SOLVE((A,B)) :- SOLVE(A),SOLVE(B).
```

donde se utiliza la primitiva CLAUSE para indicar que existe una cláusula cuya cabeza unifica con G, y cuyo resolvente es C.

Los dos primeros casos resuelven los goals atómicos mientras que el tercero se ocupa de definir al operador de conjunción dándole el significado usual.

Una forma mas general de resolver el caso anterior está basada en la reescritura del objetivo hasta llegar a la cláusula *true*, por ejemplo:

```
SOLVE(true).
SOLVE(G) :- REWRITE(G,NG),SOLVE(NG).

REWRITE(A,true) :- SYSTEM(A),A.
REWRITE(G,C) :- CLAUSE(G,C).
REWRITE((true,B),B).
REWRITE((A,B),(RA,B)) :- REWRITE(A,RA).
```

De esta manera pueden tratarse predicados del sistema, para los cuales no existen cláusulas que los tengan como cabeza.

Mediante este mecanismo pueden escribirse meta-intérpretes para lenguajes provistos con un conjunto más amplio de operadores. Precisamente esta idea es la seguida en la construcción del meta-intérprete de API-2. Por ejemplo, para el operador de corrutinaje se tienen las reglas:

```
REWRITE((A\B),(RA\RB)) :- REWRITE(A,RA),REWRITE(B,RB).
REWRITE((true\B),B).
REWRITE((A\true),A).
```

3.2 Montaje de acciones sobre el meta-intérprete.

En las reglas del meta-intérprete pueden introducirse acciones. De esta manera, tales acciones tendrán efecto a medida que se desarrolla cada paso de la ejecución del programa.

Por ejemplo, para mostrar la ejecución paso a paso de un programa, bastará con introducir en algunos pasos de la demostración una acción que exhiba el goal que se pretende demostrar. Por ejemplo:

```
SOLVE(true).
SOLVE(G) :- EXH(G),REWRITE(G,NG),SOLVE(NG).

REWRITE(A,true) :- SYSTEM(A),EXH-SYS(A),A.
REWRITE(G,C) :- CLAUSE(G,C),EXH-SLV(C).
REWRITE((true,B),B).
REWRITE((A,B),(RA,B)) :- REWRITE(A,RA).
```

De esta forma, cada goal a ser demostrado será exhibido antes de proceder a su reescritura. Cada reescritura posible de un goal, será exhibida si ésta es un resolvente de una cláusula. Si el predicado es del sistema, esto será indicado.

Esta idea fue la usada para implementar las herramientas de puesta a punto sobre el meta-intérprete de API-2.

3.3 Implementación de las herramientas de puesta a punto.

Para implementar las funciones descriptas en el capítulo anterior fue necesario definir primero la numeración dada a los goals. Cuando se da un goal para ser evaluado, se le aplica la función NUMINIT. Esta se implementó con la siguiente variante: no se considera a los goals como expresiones binarias, sino que se tiene en cuenta la asociatividad de los operadores numerándose los sub-goals en forma correlativa respecto del operador de menor jerarquía, lo cual resulta más grato a la intuición. Esta variante no causa conflictos con lo especificado.

Al goal numerado le es aplicado el ciclo del meta-interprete. Cada vez que un átomo se reescribe como un resolvente de una cláusula, se numera el resolvente con NUM, implementada con igual variante que NUMINIT.

Lo anterior se refleja en las cláusulas:

```
SOLVE(G) :- NUMINIT(G,Gn),SOLVEN(Gn).
SOLVEN(Gn) :- EXH-FST(Gn),REWRITE(Gn,RGn),SOLVEN(RGn).

REWRITE(A-n,RAn) :- L-CLAUSE(A,L),MEMBER(C,L),
                    EXH-SLV(C),NUM(C,n,RAn).
```

El predicado L-CLAUSE devuelve una lista con las cláusulas del programa que resuelven A. Cada reintento de demostración de A se reflejará en una re-invocación al predicado MEMBER. Asociando a esta re-invocación una acción adecuada se pueden exhibir los reintentos de demostraciones de A. Por otra parte, una falla en MEMBER implica que no hay más cláusulas que resuelvan A. Otra vez, una acción adecuada asociada a esta situación permite exhibir la falla de A.

Las cláusulas de reescritura correspondientes a los demás operadores son las del meta-intérprete, aplicadas a goals numerados.

Para implementar las herramientas que suspenden la traza en un intervalo de la ejecución (JMP, SKIP, GOPRED, DEMON), se modifica el predicado EXH-FST de manera tal que éste permita al usuario introducir los comandos correspondientes a tales herramientas. Los comandos introducidos modifican una variable de estado que es global al sistema y determina si debe mostrarse o no información en cada paso de demostración (la consulta sobre si debe o no mostrarse la información, la hace EXH-FST).
Mayor detalle puede encontrarse en el código Prolog que se anexa, y que no es excesivamente engorroso.

En cuanto a la facilidad de SPY, ésta se considera independiente de la traza, ya que no es necesario mostrar el contexto (numeración) asociado a cada goal en tal caso. De todas formas, la idea utilizada para su implementación es la misma que la anterior, usando los predicados EXH-FST y EXH-SLV en los casos adecuados.

3.4 Otras herramientas de puesta a punto implementadas.

3.4.1 Ejecuciones parciales ("Query the user").

La facilidad de "query the user" habilita al usuario a introducir instancias que se asumirán como válidas de predicados que han fallado. Usando la estructura del meta-intérprete de 3.1 esto puede escribirse de la siguiente forma:

```
REWRITE(A,RA) :- CLAUSE(A,RA).
REWRITE(A,RA) :- QUERY(A,RA).
```

es decir: la falla de CLAUSE provoca la consulta al usuario mediante el predicado QUERY, donde RA debe ser una instancia de A que será tomada como válida.
Esto se implementa usando listas de cláusulas como en la sección anterior. QUERY tiene lugar cuando se tienen listas vacías de resolventes de A.

En términos de la semántica formal definida en los capítulos 1 y 2, tales consultas provocarían una modificación del árbol de ejecución, pues en este caso la modificación no incumbiría solamente a la acciones asociadas a los nodos, sino que implicaría una variación en la estructura misma del árbol, esto es, el agregado de nuevos nodos correspondientes a las nuevas instancias válidas de los predicados introducidas y a partir de allí, nuevas ramas para la derivación de goals que los incluyan.

Este tema no fue abordado en la especificación de las herramientas del capítulo anterior, y queda como una posible extensión de este trabajo.

3.4.2 Debugger declarativo.

Un programa puede ser considerado como una colección de hipótesis arbitrariamente complejas y su conducta como una consecuencia de estas hipótesis. Debido a que en un momento determinado, no es posible conocer todas las consecuencias de un tal conjunto de supuestos, no se puodrán anticipar todas las posibles conductas de un programa dado. Es por esta razón que el problema de "debugging" debe ser considerado como de particular importancia en cualquier ambiente de programación [Shapiro 82].

Es importante la existencia de debuggers "declarativos", en el sentido de que puedan ser utilizados sin tener un conocimiento total del comportamiento del sistema. En [Lloyd 87] se propone un sistema declarativo de diagnóstico de errores para programas lógicos que usen sintaxis extendida y facilidades de control adicionales.

Siguiendo la propuesta recién mencionada, se implementó la parte del debugger declarativo que detecta los errores de inconsistencia de programas, sin considerar operadores de control alternativos ni errores de incompletitud en programas.

El código del debugger se anexa a la documentación entregada, y como podrá observarse, éste es muy sencillo y fácil de comprender.

Capítulo 4.

Conclusiones.

Se ha especificado formalmente un lenguaje de programación lógica extendido con un conjunto significativo de operadores de control. Aunque los aspectos de control de los lenguajes lógicos son de formalización frecuentemente engorrosa, la conseguida en este trabajo resulta clara aún cuando mantiene rigor formal.

El formalismo se ha revelado útil para especificar también acciones asociadas a cada paso de ejecución de un programa, lo cual se usó para definir también formalmente comandos para herramientas de puesta a punto. En este caso, la especificación es, sin embargo, algo tediosa por tratarse esencialmente de recorridos de árboles n-arios. Sin embargo la estructura subyacente fue útil para comprender el significado de cada comando. Igualmente, nuevas herramientas de puesta a punto pueden definirse con facilidad especificando su efecto sobre los árboles de ejecución con acciones.

La estructura de la especificación se refleja naturalmente en la de la implementación, lo cual hace a ésta también fácil de comprender.

Las extensiones del trabajo deberían contemplar la especificación e implementación de nuevas herramientas de puesta a punto con acciones que permitan modificar el árbol de ejecución del programa y de un mecanismo amigable que permita al usuario definir sus propias herramientas de puesta a punto.

Agradecimientos.

A Tato, que me ayudó como nadie con esto.

A Juan, por la cantidad de ideas que me dió y porque es lo más.

A Jorge Vidart, por su paternal dirección durante este semestre.

A Raúl Carnota.

Y un "gracias" muy especial para Daniel Pambrejo, ya sabe él por qué.

Apéndice I

Definición de la función τ_{na}

$\tau_{na}: \mathbb{P} \longrightarrow (\mathbb{G} \longrightarrow \mathbb{T}_a)$

(a) $\tau_{na}(P)(\leftarrow \text{TRUE-K}) = \langle \text{TRUE-K}, [] \rangle$

(b) $\tau_{na}(P)(\leftarrow \text{FAIL-K}) = \langle \text{FAIL-K}, [], \text{NOP} \rangle$

(c) $\tau_{na}(P)(\leftarrow \text{A-K}) =$ IF K = 0.J
THEN $\langle \text{TRUE-J}, [], \text{NOP} \rangle$
ELSE $\langle A, \mathcal{L}, \text{NOP} \rangle$

donde

$$\mathcal{L} = [\langle \text{CNJa}(\tau_{na}(P)(\leftarrow \text{NUM}(S_1\theta_1, K)), A\text{-0.K}, P), \theta_1 \rangle, \ldots, \langle \text{CNJa}(\tau_{na}(P)(\leftarrow \text{NUM}(S_n\theta_n, K)), A\text{-0.K}, P), \theta_n \rangle]$$

si el conjunto $\langle B_1 \leftarrow S_1, \ldots, B_n \leftarrow S_n \rangle$ (ordenado según su orden en P) de las cláusulas de P tales que A y B_i unifican con mgu θ_i es no vacío, y

$$\mathcal{L} = [\langle \langle \text{FAIL-K}, [], \text{NOP} \rangle, id \rangle]$$

en otro caso.

(d) $\tau_{na}(P)(\leftarrow (A;B)\text{-K}) =$
$\langle (A;B)\text{-K}, [\langle \tau_{na}(P)(\leftarrow A\text{-K.1}), id, \text{NOP} \rangle, \langle \tau_{na}(P)(\leftarrow B\text{-K.2}), id, \text{NOP} \rangle] \rangle$

(e) $\tau_{na}(P)(\leftarrow (A \hat{} B)\text{-K}) =$
$\langle (A \hat{} B)\text{-K}, [\langle \text{SNPa}(\tau_{na}(P)(\leftarrow A\text{-K.1}), B\text{-K.2}, P), id, \text{NOP} \rangle] \rangle$

(f) $\tau_{na}(P)(\leftarrow (A,B)\text{-K}) =$
$\langle (A,B)\text{-K}, [\langle \text{CNJa}(\tau_{na}(P)(\leftarrow A\text{-K.1}), B\text{-K.2}, P), id, \text{NOP} \rangle] \rangle$

(g) $\tau_{na}(P)(\leftarrow (A \backslash B)\text{-K}) =$
$\text{CORa}(\tau_{na}(P)(\leftarrow A\text{-K.1}), \tau_{na}(P)(\leftarrow B\text{-K.2}), K)$

(h) $\tau_{na}(P)(\leftarrow (\text{NOT } A)\text{-K}) = \text{NEGa}(\tau_{na}(P)(\leftarrow A\text{-K}))$

■

La definición de las funciones CNJa, SNPa, CORa y NEGa no se dará aquí por ser ésta sólo una simple extensión de las respectivas CNJn, SNPn, CORn y NEGn

Apéndice II

Definición de las funciones asociadas a los comandos de puesta a punto.

TRACE: $\mathbb{U}_\alpha \longrightarrow \mathbb{U}_\alpha$

TRACE(A)
precondición: todos los nodos del árbol deben tener asociada la función NOP.

(a) TRACE (<TRUE,[],acc>) = <TRUE,[],acc>

Los átomos TRUE se "filtrarán" durante la traza del programa, sin mostrar ningún tipo de información asociada en estos casos,ya que un TRUE implica la finalización de una demostración el cual se da a conocer al encontrar a un objetivo cerado (regla (c)).

(b) TRACE(FAIL,[],acc>) = <FAIL,[],acc>

Los átomos FAIL también se filtrarán durante la ejecución de la traza, ya que la falla en la prueba de un objetivo se muestra con la regla (d)

(c) TRACE(<A-o.K,$\mathcal{L}$,acc>) = <A-o.K,$\mathcal{L}$,(acc;EXH-SCS(A-K))>

Como se explicó en la sección anterior, un nodo de la forma A-o.K corresponde a la finalización de la prueba de A (ver punto (c) en la definición de τ_n)

(d) TRACE(<A-K,[<<FAIL,[]>,id>],acc>) =
<A-K,[<<FAIL,[]>,id>],(acc;EXH-NOSLV(A-K))>

Este es el caso en que el átomo A no tiene resolventes en el programa. Por lo tanto debe mostrarse que no hay cláusulas en el programa cuyas cabezas unifiquen con A (ver punto (c) en la definición de τ_n).

(e) TRACE(<R,[<<R_1,$\mathcal{L}_1$,acc$_1$>θ_1>,...,<<R_n,$\mathcal{L}_n$,acc$_n$>,θ_n>],acc> =
<R,[<TRACE(<R_1,$\mathcal{L}_1$,(EXH-SLV(R_1);acc$_1$)>),θ_1>,
...<TRACE(<R_n,$\mathcal{L}_n$,(EXH-FAIL(R_{n-1});EXH-SLV(R_n);acc$_n$)>),θ_n>],
(acc;EXH(R))>

Este es el caso mas complejo en la definición de la función TRACE: para cada subárbol se muestra su raiz como resolvente (de su padre) y la falla del anterior. Esto queda claro si se tiene en cuenta que los árboles con los que se está tratando son árboles or y por lo tanto cualquier subárbol de uno dado es una derivación de la raiz del mismo. entonces, si el recorrido del árbol se hace en profundidad, el inicio del recorrido de una rama significa la falla de la rama precedente.

SKIP : $\mathbb{T}_\alpha \times \mathbb{N}_p \longrightarrow \mathbb{T}_\alpha$

SKIP(A,n)
precondición: existe un nodo con p-número asociado n en el árbol y todos los nodos del árbol cuya raíz es ese nodo tienen asociada la acción EXH.

```
SKIP(<R-K,ℒ,acc>,n) =
        IF (K = n)
        THEN IF MEMBER (0.n, ℒ)
                THEN <R-K,IGN(ℒ,0.n),EXH(R)>
                ELSE <R-K,NOEXH(ℒ),(EXH(R);EXH-FAIL(R))>
        ELSE <R-K,SKP(ℒ,n),acc>
■
```

SKP: $\mathbb{L} \times \mathbb{N}_p \longrightarrow \mathbb{L}$

```
SKP([],n) = []
SKP([<R-K,ℒ,acc>,θ>!ℒ1],n) =
                        IF (K = n)
                        THEN [SKIP(<R-K,ℒ,EXH(R-K)>,θ>,n)!ℒ1]
                        ELSE [<R-K,ℒ,acc>,θ>!SKP(ℒ1,n)]
■
```

JMP:$\mathbb{T}_\alpha \times \mathbb{N}_p \times \mathbb{N}_p \longrightarrow \mathbb{T}_\alpha$

JMP(A,n1,n2)
precondición: existe un nodo con p-número asociado n1 en el árbol y todos los nodos del árbol a partir de tal nodo y el nodo con p-número n2 (si existe) tienen asociada la acción EXH, en caso de no existir un nodo con p-número n2 después del tal nodo, todos los nodos del árbol posteriores a n1 tienen asociada la acción EXH.

```
JMP(<R-K,[<t,θ>!ℒ],acc>,n1,n2) =
     IF (K = n1)
     THEN <R-K,IGN(ℒ,n2),EXH(R-K)>
     ELSE IF MEMBER(n1,t)
          THEN IF MEMBER(n2,t)
               THEN <R-K,[<JMP(t),θ>!ℒ],EXH(R-K)>
               ELSE ADD(<JMP(t),θ>,(OP-ELIM(<R-K,ℒ,EXH(R-K)>,n2)
          ELSE ADD(<t,θ>,JMP(<R-K,ℒ,acc>,n1,n2)
■
```

OP-ELIM:$\mathbb{T}_\alpha \times \mathbb{N}_p \longrightarrow \mathbb{T}_\alpha$

```
OP-ELIM(<R-K,[],acc>,n) = IF (K = n)
                              THEN <R-K,[],EXH(R-K)>
                              ELSE <R-K,[],NOP>

OP-ELIM(<R-K,[<t,θ>!ℒ],acc>,n) =
      IF (K = n)
      THEN (<R-K,[<t,θ>!ℒ],EXH(R-K)>
      ELSE ADD(<NOEXH(t),θ>,OP-ELIM(<R-K,ℒ,NOP>,n))
■
```

GOPRED:$\mathbb{T}a \times \mathbb{N}p \times \mathbb{S} \longrightarrow \mathbb{T}a$

GOPRED(A,n,p)
precondición: existe un nodo con p-número asociado n en el árbol y todos los nodos del árbol a partir de tal nodo y el nodo con goal asociado p (si existe) tienen asociada la acción EXH, en caso de no existir un nodo con goal p después del tal nodo, todos los nodos del árbol posteriores a n tienen asociada la acción EXH.

```
GOPRED(⟨R-K,[⟨t,θ⟩|ℰ],acc⟩,n,p) =
      IF (K = n)
      THEN ⟨R-K,IGNp(ℰ,p),EXH(R-K)⟩
      ELSE IF MEMBER(n,t)
            THEN IF MEMBERp(p,t)
                 THEN ⟨R-K,[⟨GOPRED(t,n,p),θ⟩|ℰ],EXH(R-K)⟩
                 ELSE
                   ADD(⟨GOPRED(t,n,p),θ⟩,⟨OP-ELIMp(⟨R-K,ℰ,EXH(R-K)⟩,p)
            ELSE ADD(⟨t,θ⟩,GOPRED(⟨R-K,ℰ,acc⟩,n,p)
```
■

OP-ELIMp:$\mathbb{T}a \times \mathbb{S} \longrightarrow \mathbb{T}a$

```
OP-ELIMp(⟨R-K,[],acc⟩,p) = IF (R = p)
                           THEN ⟨R-K,[],EXH(R-K)⟩
                           ELSE ⟨R-K,[],NOP⟩

OP-ELIMp(⟨R-K,[⟨t,θ⟩|ℰ],acc⟩,p) =
      IF (R = p)
      THEN (⟨R-K,[⟨t,θ⟩|ℰ],EXH(R-K)⟩
      ELSE ADD(⟨NOEXH(t),θ⟩,OP-ELIMp(⟨R-K,ℰ,NOP⟩,p))
```
■

DEMON:$\mathbb{T}a \times \mathbb{N}p \times \mathbb{C} \longrightarrow \mathbb{T}a$

DEMON(A,n,c)
precondición: existe un nodo con p-número asociado n en el árbol y todos los nodos del árbol a partir de tal nodo y el primer nodo tal que su goal asociado cumple c (si existe) tienen asociada la acción EXH, en caso de no existir un nodo cuyo goal cumpla c después del tal nodo, todos los nodos del árbol posteriores a ntienen asociada la acción EXH.

```
DEMON(⟨R-K,[⟨t,θ⟩|ℰ],acc⟩,n,p) =
      IF (K = n)
      THEN ⟨R-K,IGNC(ℰ,c),EXH(R-K)⟩
      ELSE IF MEMBER(n,t)
            THEN IF MEMBERC(c,t)
                 THEN ⟨R-K,[⟨DEMON(t,n,c),θ⟩|ℰ],EXH(R-K)⟩
                 ELSE
                   ADD(⟨DEMON(t,n,c),θ⟩,⟨OP-ELIMC(⟨R-K,ℰ,EXH(R-K)⟩,c)
            ELSE ADD(⟨t,θ⟩,DEMON(⟨R-K,ℰ,acc⟩,n,c)
```
■

```
OP-ELIMP:𝕋σ × ℂ ⟶ 𝕋σ

    OP-ELIMC(⟨R-κ,[],acc⟩,c) = IF c(R)
                               THEN ⟨R-κ,[],EXH(R-κ)⟩
                               ELSE ⟨R-κ,[],NOP⟩

    OP-ELIMC(⟨R-κ,[⟨t,θ⟩|ℒ],acc⟩,c) =
        IF c(R)
        THEN (⟨R-κ,[⟨t,θ⟩|ℒ],EXH(R-κ)⟩
        ELSE ADD(⟨NOEXH(t),θ⟩,OP-ELIMC(⟨R-κ,ℒ,NOP⟩,c))
■
```

```
SPY:𝕋σ × ℙ ⟶ 𝕋σ

SPY(A,p)

    SPY(⟨R-κ,[],acc⟩,p) = IF INST(R,p)
                          THEN ⟨R-κ,[],EXH(R-κ)⟩
                          ELSE ⟨R-κ,[],acc⟩

    SPY(⟨R-κ,[⟨⟨R1,ℒ1,acc1⟩θ1⟩,...,⟨⟨Rn,ℒn,accn⟩,θn⟩],acc⟩,p) =
        IF (INST(R,p) and κ ≠ 0.)
        THEN ⟨R-κ,[⟨SPY(⟨R1,ℒ1,(EXH-SLV(R1);acc1)⟩,p)⟩θ1⟩,
                ...,⟨SPY(⟨Rn,ℒn,(EXH-SLV(Rn);accn)⟩,p),θn⟩],
                (EXH(R);acc)⟩
■
```

donde INST es una función booleana que devuelve *true* si el primer parametro es instancia del segundo.

---

funciones auxiliares

```
IGN:𝕋σ × ℕp ⟶ 𝕋σ

    IGN  ([⟨⟨R-κ,ℒ,acc⟩,θ⟩|ℒ1],n)  =
            IF (κ = n)
            THEN [⟨⟨R-κ,ℒ,EXH-SCS(R-κ)⟩,θ⟩|ℒ1]
            ELSE IF MEMBER (n, ℒ)
                 THEN [⟨⟨R-κ,IGN(ℒ,n),NOP⟩,θ⟩|ℒ1]
                 ELSE [⟨⟨R-κ,NOEXH(ℒ),NOP⟩,θ⟩|IGN(ℒ1,n)]
■

IGNp:𝕋σ × ℙ ⟶ 𝕋σ

    IGN  ([⟨⟨R-κ,ℒ,acc⟩,θ⟩|ℒ1],p)  =
            IF (R = p)
            THEN [⟨⟨R-κ,ℒ,EXH-SCS(R-κ)⟩,θ⟩|ℒ1]
            ELSE IF MEMBERp (p, ℒ)
                 THEN [⟨⟨R-κ,IGNp(ℒ,p),NOP⟩,θ⟩|ℒ1]
                 ELSE [⟨⟨R-κ,NOEXH(ℒ),NOP⟩,θ⟩|IGNp(ℒ1,p)]
■

IGNC:𝕋σ × ℂ ⟶ 𝕋σ

    IGNC  ([⟨⟨R-κ,ℒ,acc⟩,θ⟩|ℒ1],c)  =
            IF c(R)
            THEN [⟨⟨R-κ,ℒ,EXH-SCS(R-κ)⟩,θ⟩|ℒ1]
            ELSE IF MEMBERC (c, ℒ)
                 THEN [⟨⟨R-κ,IGNC(ℒ,c),NOP⟩,θ⟩|ℒ1]
                 ELSE [⟨⟨R-κ,NOEXH(ℒ),NOP⟩,θ⟩|IGNC(ℒ1,c)]
■
```

```
MEMBER:ℕp × 𝕃 ⟶ 𝕃

      MEMBER(n,[]) = false

      MEMBER(n,[<<R-K,ℒ,acc>,θ>¦ℒ1]) = (n = K) or
                                        MEMBER (n,ℒ) or MEMBER(n,ℒ1)
■

MEMBERp:ℙ × 𝕃 ⟶ 𝕃

      MEMBER(p,[]) = false

      MEMBER(p,[<<R-K,ℒ,acc>,θ>¦ℒ1]) = (p = R) or
                                        MEMBERp (p,ℒ) or MEMBERp(p,ℒ1)
■

MEMBERC:ℂ × 𝕃 ⟶ 𝕃

      MEMBERC(c,[]) = false

      MEMBER(c,[<<R-K,ℒ,acc>,θ>¦ℒ1]) = c(R) or
                                        MEMBERC (c,ℒ) or MEMBERC(c,ℒ1)
■

NOEXH:𝕃 ⟶ 𝕃

      NOEXH([]) = []
      NOEXH([<<R,ℒ,acc>,θ>¦ℒ1])      =
                [<<R,NOEXH(ℒ),NOP>,θ>¦NOEXH(ℒ1)]
■
```

---

notación:

𝕃 denota el conjunto de las listas.
ℂ denota el conjunto de las condiciones.
ℙ denota el conjunto de los predicados.

Referencias.

[CDKMV 88]
Api-2: Un Meta-Ambiente para Programación Lógica.
R. Carnota, J. Diaz, A. Kvitca, A. Monteiro, J. Vidart.
aceptado para publicación, 1988.

[Knuth 68]
The Art of Computer Programming: Fundamental Algorithms.
D. Knuth
Addison Wesley, 1968.

[Lloyd 84]
Foundations of Logic Programming.
J. W. Lloyd.
Springer Verlag, 1984.

[Lloyd 87]
Declarative Error Diagnosis.
J.W. Lloyd.
New Generation Computing 5, 1987.

[Shapiro 82]
Algorithmic Program Debugging.
E. Shapiro.
PhD Thesis, Yale University, 1982.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Una implementación funcional a partir de una especificación algebraica

Fernando Carpani - Diana Cukierman

Instituto de Computación
Universidad de la República
Julio Herrera y Reissig 565
Montevideo. URUGUAY

Resumen.

Se describe una implementación realizada en un subconjunto puramente funcional de lenguaje Scheme a partir de una especificación algebraica.

De la experiencia en este proyecto de tamaño mediano (el prototipo de un intérprete de Prolog "puro"), son revisados los problemas hallados en la transformación "especificación formal - implementación".

Introducción.

Una etapa del proceso de construcción de programas es la elaboración de una especificación formal del problema. Esta cumple al menos dos funciones. La primera y principal, ser sometida a tratamiento matemático. La segunda, guiar el proceso de diseño e implementación.

Una forma difundida de implementar esta segunda función, es utilizar la especificación como medio de comunicación no ambiguo entre quien define el problema y quien diseña e implementa los programas.

El objeto de este trabajo es explorar esta visión a través del estudio de un caso: la construcción del prototipo de un intérprete de Prolog "puro" a partir de una especificación algebraica, en una porción puramente funcional de Scheme.

El trabajo se desarrolló bajo la dirección de Juan Echagüe (Instituto de Computación, Escuela Superior Latinoamericana de Informática).

En lo que sigue, se revisan las herramientas de trabajo y la Especificación Algebraica de partida, para luego pasar a puntos relevantes aparecidos en la tarea de implementación. Finalmente se discuten algunas conclusiones.

Especificación Algebraica y Scheme.

En principio, debido a lo diverso de sus objetivos, parece carecer de sentido comparar un lenguaje de especificación con uno de programación .

Sin embargo, en el proceso de desarrollo existe un momento en que deben elegirse ambas herramientas. Y muchas veces estas dos elecciones no son independientes: pesa en ellas (no exclusivamente), una noción informal de "distancia" entre los dos formalismos, que se relaciona con el costo de implementar en uno lo especificado en el otro.

La elección de Scheme [Rees 86] dada la Especificación Algebraica [Guttag 78] [Tasistro 86] fundó en la expectativa de reducir esa distancia, ya que ambos formalismos se apoyan en la noción de función. Además, la experiencia de [Zanni 87] apunta en esa dirección.

La *especificación algebraica* (EA) de *tipos abstractos de datos* (TADs) es un lenguaje de especificación formal especialmente adaptado a las técnicas de diseño basadas en la definición de clases de objetos cuyas propiedades son descriptas por su comportamiento frente a la aplicación de ciertas funciones. Provee mecanismos de representación abstracta que favorecen la separación entre la etapa de diseño y la de implementación.

Scheme es un dialecto de Lisp, que posee una semántica clara y uniforme. Posee elementos imperativos, que no fueron utilizados en este trabajo.

Entre las facilidades de la EA que aparecen en primera instancia no triviales de implementar en Scheme estan:

- semántica estricta de las funciones
- selección de axiomas basado en "filtrado"
- tipos estrictos.

En la implementación debe enfrentarse también el problema de los objetos "sin sentido" [Cornes 88] que aparecen en la EA.

Por su parte las, *funciones de orden superior* -aquellas que tienen funciones en su dominio o recorrido [Henderson 80]) -de Scheme no tiene correspondiente en EA.

Estas funciones son herramientas importantes de modularización [Vidart 88] propias de la programación funcional (sin equivalentes en programación imperativa [Echagüe 88]).

Un ejemplo clásico de función de orden superior es *composición*. Tiene como recorrido las funciones de una variable y como dominio las parejas de estas funciones. Como se espera, para una pareja de funciones f y g devuelve su composición, $f \circ g$. En Scheme se expresa

```
(define composición
     (lambda (f g)
          (lambda (x) (f (g x)))))
```

y definir, a partir de *sqr*, *padre* y *progenitor*

```
(define cuarta_potencia (composición sqr sqr))
(define 2+ (composición 1+ 1+))
(define abuelo (composición padre progenitor))
(define abuelo_paterno (composición padre padre))
```

La Especificación Algebraica de partida.

La EA de partida [Cornes 88] es un documento de 12 hojas, donde se definen 12 tipos. La técnica usada es la propuesta en [Guttag 78]. El estilo es extremadamente detallado, y para obtener completitud, especifica las operaciones selectoras a partir de las constructoras.

En [Cornes 88] se discuten extensamente los problemas de los objetos sin sentido, que aparecen en esta especificación, asi como interacciones con esta implementación en sus primeras etapas.

La tarea de implementación.

A continuación, se describen aspectos relevantes de los problemas encontrados durante la tarea de implementación.

El manejo de errores.

El objeto *indefinido* es usado en la especificación, para trabajar con funciones parciales. El simbolo *Error* en la implementación cumple el rol equivalente.

En la especificación, todas las funciones son estrictas respecto a *Error*. La implementación de esta propiedad en Scheme resulto sencilla a costa de muchas lineas de codigo para detectar la presencia de *Error* o de objetos sin sentido entre los argumentos de llamada.

Por su parte, podemos observar que una función (tanto en la especificación como en la implementación) debe devolver *Error* por una de dos situaciones:

- Alguno de los valores de su dominio es *Error* o un objeto sin sentido.
- Teniendo sentido todos los valores del dominio, la función no esta definida sobre ellos (por ejemplo, unificador más general de dos términos no unificables).

Sin embargo, será suficiente considerar solamente el segundo caso en una implementación en la que no se permita la construcción de objetos sin sentido.

Las funciones de alto orden.

Los tipos abstractos estan soportados en Scheme por enteros, símbolos y el constructor de pares ordenados (con él estan construídos productos cartesianos, listas y árboles).

Es natural definir por inducción las operaciones sobre estas estructuras, y se utiliza para ello funciones de alto orden.

Por ejemplo, las operaciones sobre listas fueron mayoritariamente definidas utilizando la siguiente función:

```
(define inducción
  (lambda ( @cabeza @resto
            constructor
            @caso_base base transformación )
    (lambda ( o )
      (if (@caso_base o)
          (base o)
          (constructor
           (transformación (@cabeza o))
           ((inducción @cabeza @resto
                       constructor
                       @caso_base base transformación)
            (@resto o)))))))
```

Cuando utilizamos *inducción* para computar una cierta función *Alfa*

- *@cabeza* y *@resto* son selectores sobre el dominio de *Alfa*.
- *constructor* es el constructor del tipo del recorrido.
- *@caso_base* detecta los casos base de la inducción, *base* devuelve la imagen de estos, y *transformación* la de los componentes.

Siendo *T1*, *T2* y *T3* tipos (no necesariamente distingos), para definir una función

$$Alfa : lista(T1) \longrightarrow T2$$

a partir de *inducción* necesitamos

$$@cabeza : lista(T1) \longrightarrow T1$$
$$@resto : lista(T1) \longrightarrow lista(T1)$$
$$constructor : T3 \times T2 \longrightarrow T2$$
$$@caso_base : lista(T1) \longrightarrow boolean$$
$$base : lista(T1) \longrightarrow T3$$
$$transformación : T1 \longrightarrow T3$$

Podemos decir que *@cabeza* y *@resto* aportan información sobre el tipo del dominio, y *constructor* sobre el tipo de recorrido.

Mediante *inducción* es posible definir operaciones sobre tipos distintos, por ejemplo:

$$Var_en_operandos? : Variable \times Operandos \longrightarrow Boolean$$

```
(define Var_en_operandos?
  (lambda ( v op )
    ((Inducción Primer_término Resto_operandos
                Funcion_or
                Operandos_vacio?
                  (lambda (x) #!false)
                  ((curry Var_en_término?) v)) op)))
```

Unir_objetivos : objetivo x objetivo ⟶ objetivo

```
(define Unir_objetivos
  (lambda (ob1 ob2)
    ((Inducción
       Primer_Atomo Resto_objetivo
       cons_objetivo
       objetivo_vacio?
         (lambda(x) ob2)
         (lambda(x) x)) ob1)))
```

El filtrado y los objetos sin sentido.

A continuación, se discuten problemas provenientes de la implementación del filtrado y el manejo de los objetos sin sentido a partir del caso particular del tipo *Término*.

Informalmente, *términos* son aquellos objetos sobre los que aplico *Símbolos de predicado* para obtener *Atomos* [Lloyd 84]. La especificación utilizada del TAD *Término* a partir de los TADs *Variable* y *Functor* es la siguiente:

/ Constructoras de Término */*

Aparición_variable : Variable ⟶ Término
Aparición_functor : Functor ⟶ Término
Cons_término : Término X Término ⟶ Término

Donde la intención de *Cons_término* es construir objetos que representan la aplicación de un functor a una serie de términos (la selección del conjunto de constructores de este TAD se discute extensamente en [Cornes 88]).

Lamentablemente esta definición permite la creación de objetos sin sentido: la aplicación de variables sobre otras. Por ejemplo:

Cons_término(Aparición_variable(a),Aparición_variable(b))

Estas construcciones no representan objeto alguno del problema original (los términos en el lenguaje Prolog).

En EA parece una buena forma de tratarlos el disponer una función que permita distinguir los objetos sin sentido de los demás, y que las restantes funciones, aplicadas sobre objetos sin sentido, devuelvan un valor distinguido *Error*.

El selector sobre *Término*, que permite distinguir los objetos sin sentido es *Término_bien_formado?*, cuya especificación es:

Término_bien_formado? : Término ⟶ Boolean

```
Término_bien_formado?(Aparición_variable(x)) = True
Término_bien_formado?(Aparición_functor(f)) = True
Término_bien_formado?(Cons_término(t1,t2)) =
        Término_bien_formado?(t1)
      and Término_bien_formado?(t2)
      and Not(Es_variable(t1))
```

Esta función se utiliza en la definición de las demás para garantizar la evaluación al valor distinguido.

Tomemos por ejemplo, el siguiente selector

Es_variable? : Término → boolean

Al definir sus axiomas, se utiliza *Término_bien_formado?* para que aplicado a objetos sin sentido, devuelva *Error*.

```
Es_variable?(Aparición_variable(x)) = True
Es_variable?(Aparición_functor(f)) = False
Es_variable?(Cons_término(t1,t2)) =
          if Término_bien_formado?(Cons_término(t1,t2))
             then False
             else Error
```

La intención es utilizar *Es_variable?* para determinar si un término fue construído mediante el constructor *Aparición_variable* y utilizar esto para implementar el filtrado.

Tenemos hasta aquí la especificación de dos funciones sobre *Término*, intuitivamente correctas, y decisiones de diseño:

- todos los selectores, aplicados sobre objetos sin sentido, devuelven el valor distinguido *Error*, con la única excepción de una función que sirve para distinguirlos de los demás objetos.
- el filtrado se implementará a partir de selectores que permiten detectar cual fue el "último" constructor utilizado para construir el objeto.
- la representación interna de los objetos será oculta para la mayoría de los selectores.

Una implementación directa de estas funciones produce, sorpresivamente, un resultado erróneo.

Dada la siguiente implementación de los constructores,

```
(define Aparición_variable (lambda(x) (cons 'variable x))
(define Aparición_functor (lambda(x) (cons 'functor x))
(define Cons_término cons)
```

es errónea una implementación inocente de los selectores de la forma siguiente (que parece natural):

```
(define Término_bien_formado?
  (lambda (t)
    (or (Es_constante? t)
        (Es_variable? t)
        (and (Término_bien_formado? (Primer_término t))
             (Término_bien_formado? (Segundo_término t))
             (not (Es_variable? (Primer_término t)))))))

(define Es_variable?
  (lambda (t)
    (if (Término_bien_formado? t)
        (igual_atomo (car t) 'variable)
        Error)))
```

Hay evaluaciones infinitas, inclusive ante términos bien formados, como

```
Es_variable?
  (Cons_término (Aparición_functor(f),Aparición_variable(v)))
```

Cuál es el problema? Las tres decisiones de diseño no pueden satisfacerse a la vez. Basta con levantar una de ellas para encontrar una solución.

Para salvar este problema, se optó por modificar la definición de *Es_variable?* (y algunos otros selectores) para que no devuelva *Error* cuando es aplicado a objetos sin sentido. Así llegó a modificarse la especificación a partir de problemas (no triviales) de implementación.

La especificación y la implementación en su versión final es:

```
Es_variable? : Término ⟶ Boolean

Es_variable? (Aparición_variable (x)) = True
Es_variable? (Aparición_functor (x)) = False
Es_variable? (C_término (t1,t2)) = False

Término_bien_formado? : Término ⟶ boolean

Término_bien_formado? (Aparición_variable (x)) = True
Término_bien_formado? (Aparición_functor (x)) = True
Término_bien_formado? (C_término (t1,t2)) =
      Término_bien_formado? (t1)
      and Término_bien_formado? (t2)
      and (not Es_variable? (t1))
```

y en Scheme

```
(define Es_variable?
  (lambda (te)
    (and (atom? te) (Simbolo_variable? te))))

(define Término_bien_formado?
  (lambda (te)
    (if (or (Es_variable? te)
            (Es_constante? te))
        True
        (and (and (Término_bien_formado? (Primer_término te))
                  (Término_bien_formado? (Segundo_término te)))
             (not (Es_variable? (Primer_término te)))))))
```

La modularización y la etapa de prueba.

Ni la EA ni el Scheme contienen la idea de módulo, en el sentido de proveer mecanismos adecuados para ocultar información.

La especificación entonces no esta dada por móodulos, sino por un conjunto de TADs, y también algunas operaciones que trabajan sobre estos, sin que quede claro que pertenezca a alguno de ellos en particular.

La implementación se construyó muy rapidamente, transcribiendo cada una de las funciones especificadas.

En este punto del trabajo, luego de una primera escritura del código Scheme aparece la necesidad de clasificar las funciones en grupos, que llamamos módulos, ordenados de alguna forma, con el fin de facilitar:

- La comprensión del código escrito.
- La prueba de las funciones, que podria realizarse por partes y en una secuencia razonable.

Esta clasificación realizó por sucesivos refinamientos.

En una primera etapa, se agrupan las funciones de acuerdo a criterios que contemplan solo los tipos de su dominio y codominio.

En una segunda se realiza una partición de estos módulos, separando donde es posible módulos que implementan clases de objetos y otros que implementan relaciones entre ellas. Aquellos se pueden asociar a los TADs de la especificación, y estos a agrupaciones de las funciones que relacionan TADs.

En esta etapa, se consideró no solamente los tipos de dominio y codominio, sino tambien los de las operaciones que se utilizaron para implementar cada función, aunque no aparezcan alli.

En la tercera y última etapa, se ordenan las funciones de cada módulo en "niveles", usando un criterio análogo al de la etapa anterior.

Como resultado se tiene una jerarquía de módulos y dentro de cada uno de éstos, una de funciones.

La prueba se realizó en forma ascendente. Para la construcción de los juegos de datos se consideraron dos criterios. El primero, revisar todos los casos límites y algunos casos usuales. Segundo, definirlos en forma ascendente, a fin de utilizar en cada nivel los datos que sirvieron para los inferiores.

Conclusiones

Se completó, dentro de márgenes de tiempo razonables, una implementación funcional a partir de la especificación algebraica del prototipo de un interprete de la porción "pura" de Prolog.

El único recurso de programación utilizado que puede considerarse fuera de la ortodoxia funcional es el *let*. Este se incluye solamente en lugares en que no en imprescindible, pero si ventajoso por razones de eficiencia. Puede eliminarse trivialmente sin poner en riesgo la corrección de la implementación.

La construcción de módulos por refinamiento para la prueba resulto ser adecuada. Se percibió la ausencia de herramientas conceptuales y de "software" para la construcción de un programa modular cuando se parte de una especificación de TADs.

Este trabajo alienta la conjetura de la corta distancia entre los lenguajes elegidos. Los problemas mayores surgen con el mecanismo de filtrado y el tratamiento de errores.

Las funciones de alto orden disponibles en Scheme son un mecanismo de alto nivel de abstracción con un gran impacto sobre la pragmática. Eso lleva a que el código escrito tenga, aproximadamente, el mismo tamaño que la especificación.

Esto no debe resultar extraño, ya que ambos documentos sirven a objetivos distintos, uno de ellos para razonar y otro para ser ejecutado.

Con dos salvedades, este trabajo alienta también a considerar positivo utilizar el criterio de minimizar esa distancia para la construcción de prototipos. La primera de la salvedades es la (in)eficiencia del programa obtenido.

La segunda es mas sustancial. Respecto a la posibilidad de razonar sobre la especificación, seria interesante verificar si es mas sencillo hacerlo dentro de EA o de la porción de Scheme que se utilizó, que posee una semántica clara.

Agradecimientos.

Deseamos agradecer los integrantes del Departamento de Programación por el apoyo prestado, especialmente a su director J. Cabezas, a C. Cornes, y a J. Echagüe por su dirección.

Tambien a J. Abrial las discusiones sostenidas en su estadia en ESLAI.

Bibliografía.

[Cornes 88] Cornes, C., Echagüe, J., Una especificación algebraica de Prolog. VIII Conferencia de la Sociedad Chilena de Computación, Santiago, 1988

[Echagüe 88] Echagüe, J., Algunas limitaciones de la "function" Pascal. Revista Escuela de Informatica, nro 3. Montevideo, 1988.

[Guttag 78] Guttag, J., Horning, J., The algebraic specification of abstract data types. Acta Informatica 10, 1978.

[Henderson 80] Henderson , P., Functional programing : Aplication and implementation. Prentice-Hall International, 1980.

[Kowalski 74] Kowalski R. A., Predicate Logic as a Programming Language, IFIP 1974.

[Lloyd 84] Lloyd, J. W., Foundations of Logic Programming, Springer-Verlag, Berlin. 1984.

[Rees 86] Rees et al, Revised^3 Report on the Algorithmic Language Scheme. ACM SIGPLAN Notes 21, 12, 1986.

[Tasistro 86] Tasistro, A., Viola, A.,Especificaciones algebraicas de tipos abstractos de datos para un curso medio de programación. XII Conferencia Latinoamericana de Informática, 1986.

[Vidart 88] Vidart, J., Tasistro, A.,Programación funcional y lógica. III Escuela Brasilero Argentina de Informática, 1988.

[Zannni 87] Zanni, C., Pedregal, C., Kesner, D., Cajias, I.. Un Editor Gráfico de Grafos. Sin publicar. 1987.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

DEFINICION DE FORMAS NORMALES DE TERMINOS CCS

A. Soriano
Grupo de Computación Paralela
Escuela de Computación
U.C.V.

RESUMEN

Se presenta la descripción de procedimientos para la construcción de formas normales para la congruencia fuerte, la equivalencia y la congruencia observacional de términos regulares de CCS. Las formas normales definidas se caracterizan por corresponder a comportamientos compuestos por el mínimo número de estados y de transiciones necesarios para su expresión. Dichos procedimientos se basan en transformaciones realizadas sobre los sistemas de transiciones que representan los comportamientos descritos por los términos.

INTRODUCCION

Actualmente, con el desarrollo de sistemas distribuidos y la concepción de lenguajes de programación capaces de expresar la concurrencia, es posible construir sistemas cada vez mas poderosos, sin embargo dichos sistemas tienden a ser cada vez mas complejos, de alli que hoy por hoy en la fase de desarrollo de un sistema es indispensable considerar una etapa para la verificación. Dicha verificación puede realizarse vía la construcción y comparación de formas normales. Estas formas normales resultan de suma utilidad en la etapa de depuración del sistema puesto que ayudan a su mejor comprensión al presentar un número minimal de estados y transiciones.

En este trabajo se describen procedimientos de construcción de formas normales de terminos regulares de CCS para la congruencia fuerte y la equivalencia y la congruencia observacional. Los procedimientos se basan en la aplicación de transformaciones apropiadas sobre relaciones de transición representando los comportamientos de los términos de CCS y en la noción de bisimulación.

El trabajo consiste de tres partes: en la primera se realiza una descripción del álgebra de CCS así como la definición y axiomatización de la equivalencia y la congruencia observacional, en la segunda se describen los procedimientos de construcción de las formas normales y por último se da una conclusión.

I. CCS: "CALCUL OF COMMUNICATING SYSTEMS"

CCS [Mi 80] es un cálculo para la descripción, la espécificación y la verificación de sistemas comunicantes asincronos. Este cálculo está basado en dos ideas principales: la observación y la comunicación sincrónica.

Informalmente, en CCS, puede describirse el comportamiento observable de un sistema (noción de observación). Es a partir de esta noción de observación que se definen los criterios de comparación de sistemas (equivalencia y congruencia observacional).

La descripción de un sistema puede hacerse también por composición paralela (operación que se escribe como ||) de procesos que se comunican entre ellos. La comunicación de procesos se considera como una acción indivisible del sistema (idea de comunicación sincronizada), esta acción se denomina comunicación interna y se denota por "τ".

En este trabajo, nos restringimos al álgebra de procesos que representa los comportamientos regulares y sin transmisión de valores. La presentación se divide en tres partes: en la primera se define el álgebra de CCS, en la segunda se da la semántica operacional y en la tercera parte se definen las tres principales relaciones de equivalencia de CCS, ésto es la congruencia fuerte, la equivalencia observacional y la congruencia observacional.

I.1.- Algebra para CCS.

Sea:

- D, un conjunto de nombres de *acciones* o *puertos de comunicación* cuyos elementos se identifican por letras minúsculas,
- ~D, un conjunto de nombres de puertos llamados *complementarios*, cuyos elementos son los elementos de D precedidos por el símbolo "~",
- una biyeccion:

  $\alpha <---> \sim\alpha$, donde $\alpha \in D$,

  $\sim\alpha \in \sim D$,

  se dice que "$\sim\alpha$" es el *puerto complementario* de "α",

  ésto nos permite definir la interacción entre procesos,
- τ, ($\tau \notin D$ y $\tau \notin \sim D$) una etiqueta que representa una *acción no observable* resultante de una comunicación interna entre α y $\sim\alpha$, $\alpha \in D$,

- P, el conjunto de etiquetas de puertos de comunicación, $P = D \cup \sim D$,
- P_τ $P_\tau = P \cup \{\tau\}$,
- F, el conjunto de funciones de redefinición f tales que:
  $f: L \subseteq P \dashrightarrow M \subseteq P$,
  f es una biyección,
  f respeta los complementos definidos por la biyección,
- V, el conjunto de variables de comportamiento,
- las firmas, $\Sigma_1 = (P_\tau, nil, +)$

  $\Sigma_2 = (P_\tau, nil, +, \|, [\,], \backslash)$

donde:

nil es una constante,
+ y || son operadores binarios,
$\forall \alpha \in P_\tau\ \alpha$, $\forall f \in F\ [f]$, $\forall \alpha \in P\ \backslash\alpha$, son operadores unarios.

Los operadores + y || son operadores infijos, α es un operador postfijo, [f] y $\backslash\alpha$ son operadores postfijos. El símbolo "." se utiliza como separador después del operador α, por ejemplo se escribe α.nil y no αnil.

La prioridad es creciente para los opeadores '+' y '||', '[f]' y '$\backslash\alpha$', 'α'. Los operadores '+', '||', '[f]' y '$\backslash\alpha$' son asociativos a izquierda y el operador 'α.' es asociativo a derecha.

Representamos por:

T_f, el álgebra de firma S_2. Los elementos de T_f se llaman *términos finitos*.
Se dice que un término finito está escrito en *forma de suma* si:
- sea t=nil,
- sea $t = \Sigma_{i \in I} \alpha_i.t_i$ y cada t_i es un término finito escritos en forma de suma.

T_S, es el subconjunto de términos finitos escritos en forma de suma. ($T_S \subseteq T_f$).

$T[\Sigma_1,V]$, el álbegra de firma $\Sigma_1 \cup V$. ($T_S \subseteq T[\Sigma_1,V]$).

$T[\Sigma_2,V]$, el álgebra de firma $\Sigma_2 \cup V$. ($T[\Sigma_1,V] \subseteq T[\Sigma_2,V]$ y $T_f \subseteq T[\Sigma_2,V]$).

Los elementos de $T[\Sigma_1,V]$ y los de $T[\Sigma_2,V]$ son expresiones a partir de las cuales se construyen los términos que definen comportamientos infinitos.

T'_r, el conjunto de pares de la forma: $u = (t,E)$

donde $t \in T[\Sigma_2,V]$,

E es un conjunto de ecuaciones de la forma: $\{v_i=t_i\}_{i=1,\ldots,n}$ donde $v_i \in V$ y $t_i \in T[\Sigma_1,V]$ y $\forall i,j\ i \neq j \Rightarrow v_i \neq v_j$.

Se dice que t_i es la *definición* de v_i.

Los elementos de T'_r definen los comportamientos infinitos. Los pares $(t,\emptyset) \in T'_r$ (*términos finitos*) se escriben simplemente t. ($T_f \subseteq T'_r$).

Para los términos de T'_r se introducen las nociones de *variable útil, variable inutil, variable libre, término cerrado, variable no guardada, término guardado y término bien guardado*. Estas nociones se utilizan para definir los términos regulares.

Para $t \in T[\Sigma_2,V]$, sea var(t) el conjunto de variables de t.

Para un término u=(t,E), se dice que una variable v es *útil* si y solamente si $v \in \bigcup_{j=0}^{card(E)} \rho^j(var(t))$, donde $\rho(X)$ se define como: $\rho(X)=\{z \mid v_i \in X \text{ y } v_i=t_i \text{ y } z \in var(t_i)\}$,

ρ^0 es la identidad

$\rho^{k+1}(X) = \rho(\rho^k(X))$, para $k>1$.

Se dice que una variable v est *inutil* en un término u=(t,E) si y solamente si $\exists t_i$ tal que $v=t_i$ pertenece a E y v no es una variable útil del término u.

Se dice que una variable v es *libre* en un término $(t,E) \in T'_r$ si y solamente si no existe j tal que $v=t_j \in E$.

Se dice que un término es *cerrado* si y solamente si no contiene variables libres.

Se dice que una variable v *no es guardada* en un término u=(t,E) si y solamente si v aparece en t o en la definición de una variable útil de u sin estar precedida por una acción.

Se dice que un término u=(t,E) es *guardado* si y solamente si toda variable no guardada del término puede reemplazarse por nil o por una expresion de la forma $\Sigma\, \alpha_i.t_i$, por substitución repetida de variables por sus definiciones.

Se dice que un término u=(t,E) es *bien guardado* si y solamente si él es guardado y además, toda variable guardada por τ, puede ser reemplazada por nil o por una expresión de la forma $\Sigma_{i=1,\ldots,n}\, \alpha_{i1}. \ldots . \alpha_{ik}.t_i, \alpha_{ik}=\tau$.

Sobre los términos de T'_r definimos los operadores α, +, ||, \, [f] de la manera siguiente:

$\alpha(t,E) = (\alpha t, E)$

$(t_1,E_1) + (t_2,E_2) = (t_1+t_2, E_1 \cup E_2)$, si $x_i \in V$ tal que $x_i = t_i \in E_1$ y $x_i = t'_i \in E_2$

$(t_1,E_1) \parallel (t_2,E_2) = (t_1 \parallel t_2, E_1 \cup E_2)$, si $x_i \in V$ tal que $x_i = t_i \in E_1$ y $x_i = t'_i \in E_2$

$(t,E)\backslash\alpha = (t\backslash\alpha, E)$

$(t,E)[f] = (t[f], E)$.

- T_r, es el subconjunto de términos cerrados y guardados de T'_r. Los elementos de T_r son los términos regulares de CCS.

1.2.- Semántica operacional de los operadores.

La semántica operacional [Pl 81] de CCS se da a través de relaciones de transición $\xrightarrow{\alpha}$, $\alpha \in P_\tau$. Estas relaciones son las más pequeñas relaciones binarias sobre T_r definidas por las reglas:

$$\frac{t \xrightarrow{\alpha} t'}{(t,E) \xrightarrow{\alpha} (t',E)}$$

$$\frac{y=t' \in E \text{ et } t[t'/_y] \xrightarrow{\alpha} t''}{(t,E) \xrightarrow{\alpha} (t'',E)}$$

(α): $\alpha.t \xrightarrow{\alpha} t$ (ACCION)

(+):

(1) $$\frac{t_1 \xrightarrow{\alpha} t'_1}{t_1+t_2 \xrightarrow{\alpha} t'_1}$$

(2) $$\frac{t_2 \xrightarrow{\alpha} t'_2}{t_1+t_2 \xrightarrow{\alpha} t'_2}$$

(SELECCION NO DETERMINISTA)

(||) (1) $\dfrac{t_1 \xrightarrow{\alpha} t'_1}{t_1 \| t_2 \xrightarrow{\alpha} t'_1 \| t_2}$

(2) $\dfrac{t_2 \xrightarrow{\alpha} t'_2}{t_1 \| t_2 \xrightarrow{\alpha} t_1 \| t'_2}$ (COMPOSICION PARALELA)

(3) $\dfrac{t_1 \xrightarrow{\alpha} t'_1, t_2 \xrightarrow{\sim\alpha} t'_2}{t_1 \| t_2 \xrightarrow{\tau} t'_1 \| t'_2}$

(\) $\dfrac{t \xrightarrow{\alpha} t', \alpha \neq \beta}{t\backslash\beta \xrightarrow{\alpha} t'}$ (OCULTADOR)

([]) $\dfrac{t \xrightarrow{\alpha} t'}{t[f] \xrightarrow{f(\alpha)} t'[f]}$ (REDEFINICION)

Notación:

- Sea $s = \alpha_1 \ldots \alpha_n$, $n>0$ y $\alpha_i \in P$
  entonces $p \left(\xrightarrow{\tau}\right)^* \xrightarrow{\alpha_1} \left(\xrightarrow{\tau}\right)^* \cdots \xrightarrow{\alpha_n} \left(\xrightarrow{\tau}\right)^* p'$
  se escribe como $p \overset{s}{\Longrightarrow} p'$.
- $t \left(\xrightarrow{\tau}\right)^* t'$ se escribe como $t \Rightarrow t'$.
- $t \left(\xrightarrow{\tau}\right)^+ t'$ se escribe como $t \overset{\tau}{\Longrightarrow} t'$.

I.3.- Relaciones de equivalencia y de congruencia.

R. Milner define relaciones de equivalencia y de congruencia sobre T'_r. Estas relaciones definen criterios de comparación entre términos. A continuación se presentan tres relaciones definidas para T'_r. Estas relaciones son:

- la *congruencia fuerte*
- la *equivalencia observacional*
- la *congruencia observacional*

I.3.1. *Congruencia Fuerte.*

Sean: $p, q \in T'_r$,

$\mathcal{R}$ una relación binaria sobre T'_r y F(R) la función:

$(p,q) \in F(R)$ ssi (i) $p \xrightarrow{\alpha} p'$ implica $\exists q'\ (q \xrightarrow{\alpha} q'$ y $(p',q') \in \mathcal{R})$ y
(ii) $q \xrightarrow{\alpha} q'$ implica $\exists p'\ (p \xrightarrow{\alpha} p'$ y $(p',q') \in \mathcal{R})$,

Definición 1.1.3.

Se dice que $\mathcal{R}$ es una *bisimulación* si y solamente si $\mathcal{R} \subseteq F(R)$.

Definición 2.1.3.

La *equivalencia fuerte* (escrita ~) se define como la más grande relación de bisimulación, es decir:

$$\sim = \cup \{\mathcal{R} \subseteq T'_r \times T'_r \mid \mathcal{R} \subseteq F(R)\}.$$

La relación ~ existe y es única porque F es monótona en el reticulado de las relaciones binarias con la inclusión de conjuntos ($\subseteq$).

Para los términos pertenecientes a T_f, la axiomatización de la *equivalencia fuerte* esta dada por: (A_1)-(A_{16}) de la Tabla 1. Esta relación de equivalencia es una relación de congruencia para los operadores a, + et ||. En lo sucesivo se denominará *congruencia fuerte*.

I.3.2.- *Equivalencia Observacional.*

La equivalencia observacional se define exactamente como la congruencia fuerte, excepto que se utilizan las relaciones $=\alpha\Rightarrow$ y $\Rightarrow$ en lugar de $\xrightarrow{\alpha}$ y $\xrightarrow{\tau}$. La equivalencia observacional se escribe como $\approx$.

Proposición 1.1.3.

Para dos términos $t_1, t_2 \in T_r$, $t_1 \approx t_2$ si y solamente si se verifican las condiciones siguientes:

(i) $t_1 =\alpha\Rightarrow t'_1$ implica $(\alpha=\tau$ y $t'_1 \approx t_2)$
o $\exists t'_2 (t_2 =\alpha\Rightarrow t'_2$ y $t'_1 \approx t'_2)$

(ii) $t_2 =\alpha\Rightarrow t'_2$ implica $(\alpha=\tau$ y $t'_2 \approx t_1)$
o $\exists t'_1 (t_1 =\alpha\Rightarrow t'_1$ y $t'_1 \approx t'_2)$.

Proposición 2.1.3.

p ~ q implica p ≈ q.

La relación de equivalencia observacional no es una relación de congruencia. Por ejemplo: a.nil ≈ t.a.nil pero, b.nil + a.nil *no es observacionalmente equivalente* a b.nil + t.a.nil.

Para términos pertenecientes a T_f, la equivalencia observacional esta axiomatizada por: (A_1)-(A_{20}) de la Tabla 1.

1.3.3.- *Congruencia Observacional.*

La *congruencia observacional* (escrita $\approx^c$) se define como la mayor relación de congruencia para los operadores α, + et || incluída en la equivalencia observacional.

Proposition 3.1.3.

Para dos términos $t_1, t_2 \in T_r$, $t_1 \approx^c t_2$ si y solamente si $\forall \alpha \in P_\tau$ se verifican la condiciones siguientes:

(1) $t_1 \xrightarrow{\alpha} t'_1$ implica $\exists t'_2 (t_2 = \alpha \Rightarrow t'_2 \text{ y } t'_1 \approx t'_2)$

(2) $t_2 \xrightarrow{\alpha} t'_2$ implica $\exists t'_1 (t_1 = \alpha \Rightarrow t'_1 \text{ y } t'_1 \approx t'_2)$

En la proposición siguiente, damos una nueva caracterización para la congruencia observacional. Esta caracterización se utilizará en la definición del procedimiento de decisión de la congruencia observacional.

Proposition 4.1.3.

Para dos términos $t_1, t_2 \in T_r$, $t_1 \approx^c t_2$ si y solamente si se verifican las condiciones siguientes:

- $t_1 \approx t_2$,
- $t_1 \xrightarrow{\tau} t'_1$ implica $\exists t'_2$ tal que $t_2 = \tau \Rightarrow t'_2$ y $t'_1 \approx t'_2$,
- $t_2 \xrightarrow{\tau} t'_2$ implica $\exists t'_1$ tal que $t_1 = \tau \Rightarrow t'_1$ y $t'_1 \approx t'_2$.

Para los términos pertenecientes a T_f, la axiomatización de la equivalencia observacional está dada por: (A_1)-(A_{19}) de la Tabla 1.

Milner a demostrado la completitud de una axiomatización para la congruencia observacional para los términos recursivos. Esta axiomatización está definida para términos recursivos escritos de

manera diferente a la nuestra, sin embargo, los axiomas y reglas de inferencia de nuestro formalismo pueden obtenerse de los suyos por transformaciones sintacticas. De esta manera, una axiomatización completa para la congruencia observacional de términos de T_r está dada por los axiomas (A_1)-(A_{19}) de la Tabla 1 y los axiomas y reglas de inferencia dados en la Tabla 2.

Cabe destacar que los axiomas (A_6), (A_7), (A_{12}), (A_{14}), (A_{16}) indicando las propiedades de asociatividad, commutatividad y distributividad de los operadores ||, \ y [ ] no pueden deducirse de los otros axiomas; pero todas sus instancias si pueden deducirse.

TABLA 1:

(+):	(A_1):	$t_1 + (t_2 + t_3) = (t_1 + t_2) + t_3$
	(A_2):	$t_1 + t_2 = t_2 + t_1$
	(A_3):	$t + t = t$
	(A_4):	$t + nil = t$
(||):	(A_5):	Si t_1 y t_2 son de la forma: $t_1 = \Sigma_{i=1,\ldots,n} \alpha_i . t'_i$ y $t_2 = \Sigma_{j=1,\ldots,n} \beta_j . t'_j$ entonces: $t_1 \| t_2 = \Sigma \alpha_i . (t'_i \| t_2) + \Sigma \beta_j . (t_1 \| t'_j) + \Sigma \tau . (t'_i \| t'_j)$ $\alpha_i = \sim\beta_j \wedge \sim\alpha_i = \beta_j \wedge i = 1,\ldots,n \wedge j = 1,\ldots,n$
	(A_6):	$t_1 \| t_2 = t_2 \| t_1$
	(A_7):	$t_1 \| (t_2 \| t_3) = (t_1 \| t_2) \| t_3$
	(A_8):	$t \| nil = t$
([]):	(A_9):	$(\alpha.t)[f] = f(\alpha).(t[f])$
	(A_{10}):	$(t_1 + t_2)[f] = t_1[f] + t_2[f]$
	(A_{11}):	$nil[f] = nil$
	(A_{12}):	$(t_1 \| t_2)[f] = t_1[f] \| t_2[f]$
(\):	(A_{13}):	$(\alpha.t)\backslash\beta = \begin{cases} nil & \text{si } \alpha = \beta \circ \alpha = \sim\beta \\ \alpha.(t\backslash\beta) & \text{si no} \end{cases}$
	(A_{14}):	$(t_1 + t_2)\backslash\alpha = t_1\backslash\alpha + t_2\backslash\alpha$
	(A_{15}):	$nil\backslash\alpha = nil$
	(A_{16}):	$t\backslash\alpha_1\backslash\alpha_2 = t\backslash\alpha_2\backslash\alpha_1$
(τ):	(A_{17}):	$t + \tau.t = \tau.t$
	(A_{18}):	$\alpha.(t_1 + \tau.t_2) = \alpha.(t_1 + \tau.t_2) + \alpha.t_2$
	(A_{19}):	$\alpha.\tau.t = \alpha.t$
	(A_{20}):	$\tau.t = t$

TABLA 2:

(1): $$\frac{t_1 = t_2}{(t, E \cup \{x_1 = t_1\}) = (t, E \cup \{x_2 = t_2\})}$$

(2): $$(t, E \cup \{x_1 = t_1, x_2 = t_2\}) = (t, E \cup \{x_1 = t_1, x_2 = t_2[t_1/x_1]\})$$

(3): $$\frac{(z_i, E) = (t_i[z_j/x_j]_{j=1,\ldots,n}, E)}{(z_i, E) = (x_i, \{x_j = t_j\}_{i=1,\ldots,n})} \quad var(t_i) \{x_1, \ldots, x_n\}$$

(4): $$\frac{x \notin \text{útil}(t, E)}{(t, E) = (t, E \cup \{x = t_1\})}$$

(5): $$(t, E \cup \{x = \tau.x + t_1\}) = (t, E \cup \{x = \tau.t_1\})$$

(6): $$(t, E \cup \{x = \tau.(x + t_1) + t_2\}) = (t, E \cup \{x = \tau.x + t_1 + t_2\})$$

II. Formas Normales de Términos Regulares.

En este trabajo, se presenta la construcción de formas normales de términos regulares para la congruencia fuerte, la equivalencia y la congruencia observacional. Estas formas normales corresponden a términos cuyos sistemas de transiciones asociados poseen un número mínimo de estados y de transiciones.

Para la construcción de las formas normales, se utilizan las transformaciones *τ-cadenas y τ-circuitos* definidas en [So 87a] y la noción de *bisimulación*.

Para la construcción de cada una de las formas normales, se procede en tres etapas:

1.- Construcción del sistema de transiciones asociado al término.
2.- Applicación de transformaciones adecuadas a dicho sistema de transiciones (reducción).
3.- Construcción del término de T_r asociado al sistema de

transiciones transformado.

La etapa 1 fue descrita en [So 87a], a continuación se describen las etapas 2 y 3 de los algoritmos de construcción de formas normales.

II.1. Transformaciones de los sistemas de transiciones.

En esta parte se definen la transformaciones *NCF*, *NEO* y *NCO* y se demuesta que estas trasformaciones permiten construir las formas normales de los sistemas de transiciones para cada una de las equivalencias.

II.1.1.- Transformación *NCF*.

Definición 1.II.1.

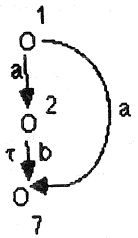
Para todo sistema de transiciones $S=(Q, \cup_{\alpha \in Pt} P_\alpha, q_0)$, *NCF*(S) es el sistema de transiciones cociente con respecto a la congruencia fuerte (las clases de equivalencia se obtienen por cálculo de la mas grande bisimulación contenida en QxQ).

Proposición 1.II.1.

Para todo sistema de transiciones S, *NCF*(S) es una forma normal para la congruencia fuerte.

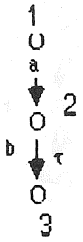
II.1.2.- Transformaciones *NEO* y *NCO*

En el caso de la equivalencia y de la congruencia observacional, el sistema de transiciones cociente posee un número minimal de estados, pero no siempre un número minimal de transiciones. Por ejemplo, el sistema de transiciones reducido:



posee un número minimal de estados, pero no es minimal respecto al número de transiciones. En efecto, este sistema es

observacionalmente equivalente al sistema:



Este sistema es la forma normal (sistema bien guardado y compuesto por un mínimo número de estados y de transiciones).

Definición 1.II.2.

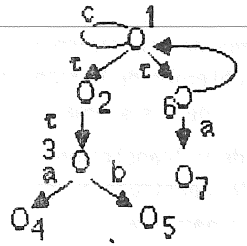
Para todo sistema de transiciones S, el sistema de transiciones *NEC*(S) se obtiene por aplicación de la siguiente secuencia de transformaciones:

1.- Transformaciones τ-cadenas y τ-circuitos.

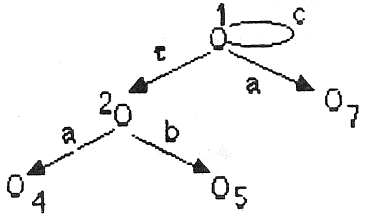
2.- Reducción respecto a la equivalencia observacional.

3.- Eliminación de toda transición $p \xrightarrow{\alpha} q$ en el caso en que $\exists s \in \{\tau^*\alpha\tau^+, \tau^+\alpha\tau^*\}$ tal que $p \xrightarrow{s} q$.

Por ejemplo, sea el siguiente sistema de transiciones:

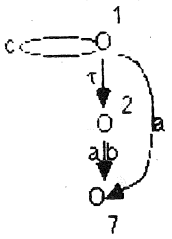


- Por aplicación de las transformaciones τ-cadenas y τ-circuitos, se obtiene el sistema de transiciones:

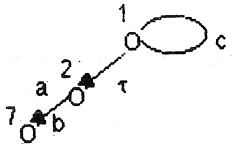


- El sistema de transiciones cociente respecto a la equivalencia

observacional es el siguiente:



- Por eliminación de transiciones superfluas (la transición $1 \xrightarrow{a} 3$ queda eliminada ya que $1 \xrightarrow{\tau a} 3$) se obtiene el sistema de transiciones *NEO*(S):



Proposición 1.II.2.

Para todo sistema de transiciones S, *NEO*(S) es una forma normal para la equivalencia observacional.

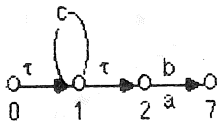
Definición 1.II.3.

Para todo sistema de transiciones S, *NCO*(S) es el sistema de transiciones obtenido de la manera siguiente:

1.- Construcción del sistema $S'=(Q, \cup_{\alpha \in P\tau} R_\alpha, q_0)$ por applicación de la transformación *NEO* sobre S.

2.- Si durante la construcción del sistema de transiciones *NEO*(S) eliminamos al menos una τ-transición partiendo del estado inicial, se agregará un nuevo estado inicial q'_0, y una τ-transición $q \xrightarrow{\tau} q'_0$.

Por ejemplo, para el sistema de transiciones S del ejemplo anterior, el sistema de transiciones *NCO*(S) es el siguiente:



Proposición 1.II.3.

Para todo sistema de transiciones S, *NCO* (S) es una forma normal para la congruencia observacional.

II.2.- Construcción del término asociado a un sistema de transiciones.

Definición 1.II.4.

El término regular asociado a un sistema de transiciones $S=(Q, \cup_{\alpha\in P_\tau} R_\alpha, q_0)$ es $u_S=(C(q_0),E)$ donde:

$E=\{x_p=D(p)\}_{p\in Q'}$, con $Q'\subseteq Q$, $D: Q \to T[\Sigma_1,V]$, $C: Q \to T[\Sigma_1,V]$ definidos por,

$q_0\in Q'$ ssi $\exists q\in Q, \exists\alpha\in P_\tau$ tales que: $q\xrightarrow{\alpha}q_0$,

$q\in Q'$, $q\neq q_0$ ssi $(\exists q_1,q_2,q_3\in Q, \exists\alpha,\beta,\gamma\in P_\tau$ tales que: $q_1\xrightarrow{\alpha}q$, $q_2\xrightarrow{\beta}q$ y $q\xrightarrow{\gamma}q_3)$

$D(p)=nil$,	si $\exists q\in Q$, $\exists\alpha\in P_\tau$ tales que	$p\xrightarrow{\alpha}q$
	$D(p)=\Sigma_{i\in I}\alpha_i.C(q_i)$,	ssi $p\xrightarrow{\alpha_i}q_i$ para $i\in I$ y para todo j tal que $p\xrightarrow{\alpha_j}q_j$, $j\in I$,
	$C(q)=x_q$,	si x_q es una variable de E.
	$C(q)=D(q)$,	si no lo es.

Proposición 1.II.4.

Si S es un sistema de transiciones en forma normal para la congruencia fuerte, o para la equivalencia o la congruencia observacional, entonces el término u=(t,E) associado a S definido anteriormente esta tambien en forma normal para le equivalencia fuerte o la equivalencia o la congruencia observacional.

CONCLUSION.

El objetivo del trabajo fué el de definir procedimientos de construcción de formas normales.

Se realizó un estudio del cálculo CCS sin transmisión de valor. Para términos regulares, se definieron procedimientos de

construcción de formas normales minimales para la congruencia fuerte y la equivalencia y la congruencia observacional. Estos procedimientos se basan en transformaciones sobre los sistemas de transiciones que representan los comportamientos descritos por los términos.

REFERENCIAS.

[Mi 80] Milner R.
A calculus of communicating systems.
Lecture Notes in Computer Science 92.
Springer-Verlag. 1980.

[Mi 83] Milner R.
Calculi for synchrony and asynchrony.
T.C.S. 25; pp. 267-310. 1983.

[Mi 85] Milner R.
A complete axiomatisation for observational congruence of finite state behaviors.
Manuscrito no publicado, diciembre 1985.

[Pl 81] Plotkin G.
A structural approach to operational semantics.
Aarhus University. Denmark.
DAIMI FN-19. Septembre 1981.

[So 87] Soriano A.
VENUS: UN OUTIL D'AIDE A LA VERIFICATION DES SYSTEMES COMMUNICANTES.
Tesis de doctorado. INPG. Francia. 1987.

[So 87a] Soriano, A.
Procedimientos de decisión para la equivalencia y la congruencia observacional de términos CCS
Anales de la XII Conferencia Latinoamericana de Informática. 1987. pp. 118-137.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Una representación no tradicional de producto cartesiano

Laura Almada - Gustavo Betarte - Paula Severi

Instituto de Computación
Universidad de la República
Montevideo - URUGUAY

Resumen

Se describe una representación general de producto cartesiano de componentes elementales. Se utiliza como herramienta de especificación a la especificación algebraica de tipos abstractos de datos (E.A.T.A.D.) [Gutagg 78].

1. Introducción

Este trabajo presenta parte de un proyecto desarrollado por el Area de Métodos y Lenguajes, Departamento de Programación, Instituto de Computación, Universidad de la República.

El proyecto surge como una iniciativa de un grupo de docentes de dicha Area, con el objetivo de experimentar nuevas técnicas de diseño, no aplicadas hasta el momento en el mercado uruguayo. Se trata del diseño de un sistema administrativo, a partir de una noción intuitiva de lo que cubre un sistema con estas características, prescindiendo de los detalles particulares de cada empresa. La elección de este tema se debe a que la mayor parte de los sistemas desarrollados en el medio nacional son sistemas administrativos.

Este trabajo se limita a presentar un modelo para los objetos de estructura finita del sistema administrativo.

En la segunda sección, se hace una primera aproximación a la representación general de los objetos del sistema. En la tercera sección se muestra el proceso a través del cual se llega a una representación de los mismos. En la cuarta sección se presentan conclusiones sobre el trabajo. En el apéndice A se presenta la representación usando especificación algebraica. En el apéndice B se muestra como se resuelven posibles requerimientos del usuario. En el apéndice C se ilustran diferencias entre la representación propuesta y la clásica a través de ejemplos.

2. Características generales de los objetos del sistema

Un sistema admistrativo trabaja con un conjunto muy amplio de objetos (cuentas corrientes, empleados, etc.). Sin embargo, si se los analiza con un nivel de abstracción suficientemente elevado, se puede llegar a la conclusión de que todos ellos pueden ser modelados mediante productos cartesianos de componentes elementales. En este trabajo se presenta una representación general del producto cartesiano, de forma tal de uniformizar la estructura de los objetos de un sistema administrativo, a los que se denomina de aquí en más S_objetos.

Como la estructura de todos los S_objetos es la misma (producto cartesiano de componentes elementales), pueden ser manipulados con un conjunto de operaciones comunes a todos ellos. Sin embargo, sería útil que los S_objetos fueran divididos en clases, de acuerdo a las propiedades que cumplen. A un S_objeto perteneciente a una clase se le pueden aplicar además de las operaciones generales, comunes a todos los S_objetos, operaciones particulares de su clase.

Las operaciones que manipulan a los S_objetos deben ser:

- generales - su semántica es común a todos los S_objetos.
- elementales - lo suficientemente sencillas de forma de que no haya superposición de funciones. O sea, deben ser disjuntas.
- fácilmente componibles - las operaciones sobre una clase pueden, en general, ser expresadas como composición de estas operaciones.

3. Fundamentos de la representación de los S objetos

El objetivo de esta sección es definir los S_objetos y como se dijo en la sección anterior éstos no son más que productos cartesianos de componentes elementales, es decir conjuntos de "campos" o atributos.

Estructura e instanciación de atributos

Un atributo puede ser descripto como producto cartesiano de los siguientes conjuntos:

.nombre
.tipo
.largo
.valor

Puede por tanto ser representado por el siguiente Tipo Abstracto de Datos (TAD):

TAD ELEM

creo-elem: ELEM-NOM X ELEM-TIPO X LARGO X VALOR ----> ELEM

Por ejemplo , el atributo Precio de un S_objeto cualquiera podría ser descripto con una cuadrupla de este estilo :

(Precio , Real , 5 , 124.7)

Se considera la componente largo de estas cuadruplas con el objetivo de conocer la cantidad máxima de caracteres que puede tomar el valor de ese atributo. El resto de las componentes tiene un significado bastante claro.

El TAD anterior representa una instancia de un atributo ya que tiene valor asociado. Para definir la estructura de un atributo (sin instanciarlo) alcanzan los tres primeros conjuntos. Se puede formular la definición de la misma con el siguiente TAD:

TAD INFO

creo-info: INFO-NOM X INFO-TIPO X LARGO ----> INFO

En el ejemplo anterior la estructura sería:

(Precio , Real , 5)

Estructura e instanciación de S_objetos

Los tads anteriores definen la estructura e instanciaciones de cualquier atributo de un S_objeto. La estructura de un S_objeto (como conjunto cuyos elementos son estructuras de atributos) puede ser descripto mediante el siguiente TAD:

TAD OBJ-ESTRUC

creo-obj-estruc: LISTA(INFO) X CANT ----> OBJ-ESTRUC

Pero este tipo abstracto presenta una carencia: no es posible distinguir estructuras de clases de S_objetos, es decir esta estructura no tiene nombre. Por tanto, se reformula el tipo anterior como sigue:

TAD OBJ-ESTRUC

creo-obj-estruc: NOM-TIPO X LISTA(INFO) X CANT ---> OBJ-ESTRUC

Este nuevo TAD define la estructura común por medio de la cual se representa cualquier S_objeto, con la siguiente información:

- Un nombre para cada estructura de S_objeto (NOM-TIPO).
- Una definición de la estructura del S_objeto como conjunto de campos (LISTA(INFO)).
- Una componente que representa la cantidad de campos que contiene el S_objeto (CANT).

Los ejemplos siguientes, aunque simplificados muestran la estructura de dos posibles S_objetos.

(a) (CUENTA CORRIENTE, (Numero, entero, 6) (Nombre, lista(char), 30) (Descrip, lista(char), 50) (Saldo, real, 20) , 5)

(b) (EMPLEADO, (Nombre, lista(char), 30) (Seccion, lista(char), 10) (Direccion, lista(char), 40) (Sueldo, real, 20) , 4)

Analizando el tipo abstracto OBJ-ESTRUC y los ejemplos vistos, es posible extraer las siguientes conclusiones:

- La definición de OBJ-ESTRUC es general, en el sentido de que con un único tipo abstracto es posible definir la estructura de cualquier S_objeto. Se puede notar en toda su amplitud esta generalidad si se la compara con la representación tradicional de producto cartesiano. En esta última, para definir una cuenta corriente y un empleado sería necesario definir dos tipos distintos:

TAD CUENTA-CORRIENTE

creo-cta-corr: ENTERO X STRING X STRING X REAL ----> CUENTA-CORRIENTE

TAD EMPLEADO

creo-empleado: STRING X STRING X STRING X REAL ---> EMPLEADO

Con lo cual se necesitaría operaciones distintas para manipular cada uno de los tipos definidos, perdiendo la generalidad deseada.

- Dos elementos distintos del tipo OBJ-ESTRUC definen dos subclases distintas. Entonces, dos S_objetos pertenecen a la misma subclase si su estructura se corresponde con el mismo elemento de OBJ-ESTRUC. Por tanto se ha definido la relación de equivalencia que divide a los S_objetos en subclases. Representar S_objetos distintos de una misma subclase es equivalente a definir instancias diferentes de la subclase, esto es, darle valor a los atributos de la estructura.

Un S_objeto es una instanciación de un elemento de OBJ-ESTRUC. Por tanto, se define el TAD objeto universal como sigue:

TAD OBJ-UNI

creo-obj-uni : NOM-TIPO X LISTA (ELEM) X CANT ----> OBJ-UNI.

En el ejemplo:

```
                    (Numero, entero, 6, 102430)
        (I)         (Nombre, lista(char), 30, "INCO")
(CUENTA CORRIENTE,  (Descrip, lista(char), 50, "Fac.Ing.") , 5 )
                    (Saldo, real, 20, 2340000.00)

                    (Numero, entero, 6, 102431)
        (II)        (Nombre, lista(char), 30, "IME")
(CUENTA CORRIENTE,  (Descrip, lista(char), 50, "Fac.Ing.") , 5 )
                    (Saldo, real, 20, 140000.00)
```

Operaciones sobre los S_objetos

Hasta ahora se ha centrado la discusión en la definición de los S_objetos. A continuación, se hace un análisis de las ventajas que se obtienen en la manipulacion de distintos S_objetos con un mismo conjunto de operaciones.

En primer lugar, como todos los S_objetos que se manipulan son instancias diferentes del mismo tipo, se necesita una única operación constructora.

El conjunto de operaciones selectoras (una para cada campo), típicas del producto cartesiano tradicional, se transforman en una única operación de selección, común a todos los S_objetos, independientemente de la subclase a la que pertenecen.

Con la representación propuesta es posible definir operaciones de selección más sofisticadas que las clásicas, como por ejemplo la selección de los tipos de los S_objetos y de los atributos correspondientes.

En función de estas operaciones (constructoras y selectoras de componentes) es posible construir nuevas operaciones generales sobre los S_objetos (por ejemplo actualizar valores de componentes), que le agregan poder de manipulación al tipo abstracto.

Subclases, estructuras e instanciaciones

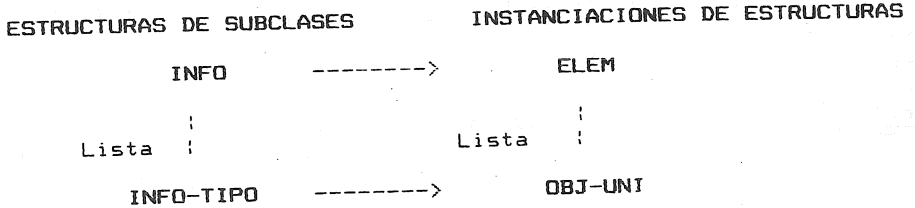
El problema original era representar todos los S_objetos con una estructura común, o sea definir una sola clase que los contenga. Esta clase tiene asociada operaciones propias que pueden ser aplicadas a todos los S_objetos. Pero además de éstas, existen operaciones que van a ser particulares de algún subconjunto de los objetos, por ejemplo habrán operaciones que interesará aplicar sobre objetos de tipo CUENTA CORRIENTE que no serán las mismas que se aplicarán sobre los objetos de tipo EMPLEADO. Cada subconjunto con operaciones propias asociadas es una subclase, y su estructura podrá ser definida como una instancia del TAD OBJ-ESTRUC.

Entonces, un S_objeto cualquiera puede verse como una instancia del objeto universal (TAD OBJ-UNI) y se construye asociándole valores al conjunto de atributos correspondientes a la subclase a la cual pertenece. Por lo tanto, existe una correspondencia entre los elementos de los tipos abstractos que definen las estructuras de los S_objetos y los elementos de los tipos que definen instancias de los mismos.

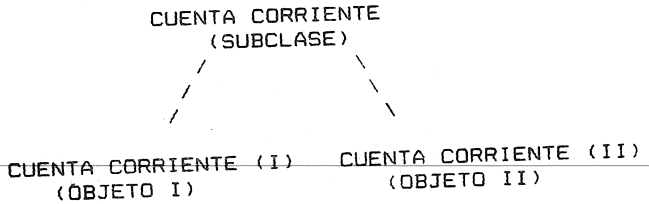
Una primera correspondencia se establece entre los elementos del TAD INFO (estructura de atributos) y los del TAD ELEM (instancias de los mismos).

Una segunda correspondencia se establece entre los elementos que define el TAD OBJ-ESTRUC (estructuras de subclases) y los que define el TAD OBJ-UNI (instancias de los anteriores).

Dichas correspondencias pueden ser esquematizadas como sigue:



Por ejemplo, la estructura de la subclase CUENTA CORRIENTE definida mediante OBJ-ESTRUC en el ejemplo (a) se corresponde con varias instancias de ésta como se vio en los ejemplos (I) y (II). Esto se puede ver con el siguiente esquema:



Una estructura para las estructuras

Las definiciones dadas no son suficientes para construir un S_objeto perteneciente a una subclase dado el nombre de la misma. Es necesario disponer de un conjunto de elementos de tipo OBJ-ESTRUC. La representación de este conjunto es una lista.

Con esto aún no es suficiente. Es necesario disponer de alguna operación que permita efectivamente crear un S_objeto. Esta operación, recibe el nombre de una subclase, selecciona de la lista de elementos de tipo OBJ-ESTRUC aquel cuyo nombre coincida con el dado, y aplica la operación cr-obj-uni a la componente del elemento de OBJ-ESTRUC hallado.

Evidentemente, esto no podría haber sido resuelto si no se tuviera una lista de todas las subclases. Esta última operación no es igual al resto ya que puede ser aplicada sin conocer la estructura del objeto universal.

Se debe hacer notar que se ha logrado la independencia deseada relativa a los tipos que se definen ya que no es necesario definir tipos distintos para cada caso particular como en el producto cartesiano clásico. Para ello, se pasan los tipos al dominio de los valores y se le asocian operaciones a éstos.

Se podría generalizar aún más y decir que es posible definir un conjunto de operaciones sobre el objeto universal, escondiendo en su aplicación no sólo la estructura del objeto universal sino también la existencia del resto de las estructuras como OBJ-ESTRUC. A este conjunto de operaciones que definen un lenguaje de distinto nivel de abstracción que el del objeto universal, lo llamamos lenguaje del usuario.

4. Conclusiones

Se ha presentado una representación general de producto cartesiano de componentes elementales. Esta representación, única para todos aquellos objetos que puedan ser modelados por medio de producto cartesiano, permite que se definan operaciones comunes a todos ellos.

Los dominios de las componentes del producto cartesiano pasan a ser variables, y por tanto también se hace variable el nombre que identifica al conjunto de estos dominios (su tipo). Esto hace que el nombre del tipo sea tratado como una componente más del producto cartesiano.

La definición propuesta permite la creación de nuevos tipos mediante operaciones. Para ello es necesario poder manipular el conjunto de definiciones de las subclases. Este conjunto puede ser visto como generador de los S_objetos (elementos del espacio del sistema).

Una instancia posterior a este trabajo sería definir un conjunto de S_objetos, y operaciones que lo manipulen.

Bibliografía

1. Meyer, Bertrand, "Reusability: The case for Object-Oriented Design", IEEE. 1987.

2. Stroustrup, Bjarne, "What is "Object-Oriented Programming"?, BIGRE (INRIA). 1987.

3. Liskov, Barbara y Guttag, John, "Abstraction and specification in program development", MIT PRESS. 1986

4. Cardelli, Luca y Wegner, Peter, "On understanding types data abstraction and polimorphism", ACM COMPUTING SURVEYS. 1985

Agradecimientos

Agradecemos especialmente la colaboración de las siguientes personas cuyos aportes contribuyeron a la elaboración de este trabajo:

- Msc. Juan José Cabezas
- Ing. Alfredo Viola
- Prof. Alvaro Tasistro
- Ing. Hermann Stephen
- Ing. Patricia Peratto

Y a todos los compañeros del Departamento de Programación, por el continuo apoyo recibido.

APENDICE A

Este apéndice presenta la especificación algebraica de los tipos abstractos utilizados para la definición de los S_objetos.

Representación de la estructura de un atributo

TAD INFO.

Cr-info: INFO-TIPO X INFO-NOM X LARGO ----> INFO.

Info-tipo: INFO ----> INFO-TIPO.
Info-nom: INFO ----> INFO-NOM.
Info-largo: INFO ----> LARGO.

Representación de la instanciación de un atributo

TAD ELEM.

Cr-elem: ELEM-TIPO X ELEM-NOM X LARGO X VALOR ----> ELEM.

Elem-tipo: ELEM -----> ELEM-TIPO.
Elem-nom : ELEM -----> ELEM-NOM.
Elem-largo: ELEM -----> LARGO.
Elem-val : ELEM -----> VALOR.

Representación de la estructura de un S_objeto

TAD OBJ-ESTRUC.

Cr-obj-estruc: NOM-TIPO X LISTA(INFO) X CANT ----> OBJ-ESTRUC.

Obj-estruc-nom: OBJ-ESTRUC ----> NOM-TIPO.
Obj-estruc-list: OBJ-ESTRUC ----> LISTA(INFO).
Obj-estruc-cant: OBJ-ESTRUC ----> CANT.

Representación de la instanciación de un S_objeto

TAD OBJ-UNI.

Cr-obj-uni: NOM-TIPO X LISTA(ELEM) X CANT ----> OBJ-UNI.

Obj-uni-nom: OBJ-UNI ----> NOM-TIPO.
Obj-uni-list: OBJ-UNI ----> LISTA(ELEM).
Obj-uni-cant: OBJ-UNI ----> CANT.

Las operaciones de todos los tads definidos son selectoras, por tanto su especificación se omite.

Tad's primitivos o predefinidos

- LISTA(T).
- INTEGER.
- CHAR.
- BOOLEAN.
- REAL.

Otros Tad's

Como se planteaba en la tercera sección, para poder representar y manipular el conjunto de subclases que definen los S_objetos, se define un TAD LISTA(OBJ-ESTRUC), para lo que se utiliza el TAD primitivo LISTA(T).

APENDICE B

Especificación de los requerimientos del usuario

En este apéndice se resuelven algunos de los posibles requerimientos del usuario sobre los S_objetos, trabajando con los TADs previamente definidos.

Es importante señalar, que la comunicación del usuario con el sistema, es resuelta por un módulo de interfase, pero dicho tratamiento escapa a los objetivos de este trabajo.

Primero se describen los requerimientos en lenguaje natural, y luego se especifican algebraicamente.

Descripción en lenguaje natural de los requerimientos

1- Crear un objeto de una subclase.
Dado un nombre de subclase, se obtiene un S_objeto perteneciente a la misma. El valor de la cuarta componente es nulo.

2- Asignar valor a un atributo de un S objeto.
Dado un S_objeto, el nombre de un atributo y un valor, se obtiene el S_objeto con el valor de ese atributo modificado.

3- Seleccionar un atributo de un S objeto.
Dado un S_objeto y un nombre de atributo, se obtiene toda la información asociada al mismo.

4- Seleccionar información de todos los atributos de un S objeto.
Dado un S_objeto se obtiene la lista de atributos del mismo.

5- Seleccionar el tipo de un atributo de un S objeto.
Dado un S_objeto y el nombre de un atributo, se obtiene el tipo asociado al atributo.

Especificación algebraica de los requerimientos

1 - Crear un objeto de una subclase.

Sintaxis

```
Dev-obj : NOM-TIPO ------> OBJ-UNI

Operaciones auxiliares

Crea-obj : OBJ-ESTRUC  ----> OBJ-UNI
Asig-val : LISTA(INFO) ----> LISTA(ELEM)
Subclase : LISTA(OBJ-ESTRUC) X NOM-TIPO ----> OBJ-ESTRUC
```

Semántica

```
Dev-obj(n)=Crea-obj(Subclase(l,n))

Crea-obj(l) =
  Cr-obj-uni(Obj-estruc-nom(l),Asig-val(Obj-estruc-list(l)),
                                        Obj-estruc-cant(l))

Asig-val(Cr-lista()) = Cr-lista()
Asig-val(Add(l,i)) =
     Add(Asig-val(l),Cr-elem(Info-tipo(i),Info-nom(i),
                                        Info-largo(i),"vn")

Subclase(Cr-lista(),n) = indef
Subclase(Add(l,i),n)   = if Obj-estruc-nom(i) = n then i
                         else  Subclase(l,n)
```

2 - Asignar valor a un atributo de un S objeto.

Sintaxis

```
Ins-val: OBJ-UNI X ELEM-NOM X VALOR ----> OBJ-UNI

Operacion auxiliar

Cambiar-Valor: LISTA(ELEM) X ELEM-NOM X VALOR ---->
                                              LISTA(ELEM)
```

Semántica

```
Ins-val(o,n,v) =
    Cr-obj-uni(Obj-uni-nom(o),
        Cambiar-valor(Obj-uni-list(o),n,v), Obj-uni-cant(o))
```

```
Cambiar-valor(Cr-lista(),n,v) = indef
Cambiar-valor(Add(l,e),n,v) =
            if Elem-nom(e) = n then
                Add(l,Cr-elem(Elem-tipo(e),n,Elem-largo(e),v)
            else
                Add(Cambiar-valor(l,n,v),e)
```

3 - Seleccionar un atributo de un S objeto.

Sintaxis

Obt-elem: OBJ-UNI X ELEM-NOM ----> ELEM

Operación auxiliar.

Get-elem: LISTA(ELEM) X ELEM-NOM ---->ELEM.

Semántica

```
Obt-elem(o,ne) = Get-elem(Obj-uni-list(o),ne)

Get-elem(Cr-lista(),n) = indef
Get-elem(Add(l,e),n)   = if Elem-nom(e) = n then e
                         else  Get-elem(l,n)
```

4 - Seleccionar información de todos los atributos de un S objeto.

Sintaxis

Obj-uni-list: OBJ-UNI ----> LISTA(ELEM)

Semántica

```
Obj-uni-list(Cr-obj-uni(n,l(e),c) = l(e)
```

5 - Seleccionar el tipo de un atributo de un S objeto.

Sintaxis

Tip-atr: OBJ-UNI X ELEM-NOM ----> ELEM-TIPO

Semántica

```
Tip-atr(o,ne)  =  Elem-tipo(Obt-elem(o,ne))
```

APENDICE C

Este apéndice ilustra a través de ejemplos el alcance que tienen las operaciones definidas sobre las estructuras propuestas.

Ejemplo1.

Dados dos S_objetos, en caso de que sean del mismo tipo devolver una lista de reales donde cada real sea la suma de las componentes reales respectivas de cada S_objeto.

En este ejemplo, se muestra la posibilidad de manejar los tipos de los objetos y sus componentes como variables.

Sintaxis

```
Suma-dos-objetos: OBJ-UNI X OBJ-UNI ----> LISTA(REAL)

Operaciones auxiliares

Selec-valor: LISTA(ELEM) X ELEM-TIPO ----> LISTA(VALOR)
Suma-lista: LISTA(VALOR) X LISTA(VALOR) ----> LISTA(VALOR)
```

Semántica

```
Suma-dos-objetos(Cr-obj-uni(n,l,c), Cr-obj-uni(n',l',c')) =
    if  n = n' then
       Suma-lista(Selec-valor(l,REAL), Selec-valor(l',REAL))
    else
       indefinido
    endif

Selec-valor(Cr-lista(), n) = Cr-lista()
Selec-valor(Add(l,e), n) =
                  if Tipo-elem(e) = n then
                        Add(Selec-valor(l,n), Elem-val(e))
                  else
                        Selec-valor(l,n)
                  endif

Suma-lista(Cr-lista(), Cr-lista()) = Cr-lista()
Suma-lista(Cr-lista(), Add(l,e)) = Add(l,e)
Suma-lista(Add(l,e), Cr-lista()) = Add(l,e)
Suma-lista(Add(l,e), Add(l',e')) =
                     Add(Suma-lista(l,l'),Suma-nros(e,e'))
```

Donde Suma-nros es una operación que suma 2 números (reales o enteros).

Notar que esta operación se ha definido sobre cualquier par de S_objeto independientemente de la clase a la que pertenezcan. Esto es posible ya que existe una estructura común para todos los S_objetos.

Si se definieran los S_objetos como en el producto cartesiano clásico, no existiría una representación única para todos los S_objetos, y sería necesario definir operaciones distintas para cada clase, aunque la semántica de ellas sea idéntica excepto por los tipos involucrados.

La distinción de tipos es necesaria en ambos casos. Cuando se define una estructura común alcanza con comparar los tipos ya que es una componente más. Cuando se usa el producto cartesiano clásico la distinción de tipos está implícita en su estructura por lo que es necesario definir una operación distinta para cada tipo.

Por otra parte, en el primer caso el tipo de los atributos de un S_objeto es variable, por lo que se puede realizar una operación que dado un S_objeto y un tipo de atributo seleccione todos los atributos que sean de ese tipo. La estructura definida para los S_objetos permite definir operaciones generales para las clases de los S_objetos y sus atributos. Esto no es posible en el producto cartesiano clásico.

Ejemplo 2.

Dado un elemento de OBJ-ESTRUC y una lista de INFO-NOM obtener un nuevo elemento de OBJ-ESTRUC que esté compuesto únicamente por aquellos atributos cuyos nombres se corresponden con los de la lista.

Sintaxis

Proyectar: OBJ-ESTRUC X LISTA[INFO-NOM] X NOM-TIPO ---> OBJ-ESTRUC

Operaciones auxiliares

Cr-lista-proy: LISTA[INFO] X LISTA[INFO-NOM] ----> LISTA[INFO]

Semántica

```
Proyectar(Cr-obj-estruc(n,l,c), l', n') =
    Cr-obj-estruc(n',Cr-lista-proy(l,l'),
                                Cantidad(Cr-lista-proy(l,l')))

Cr-lista-proy(Cr-lista(), Cr-lista()) = Cr-lista()
Cr-lista-proy(Cr-lista(), Add(l,e)) = Cr-lista()
Cr-lista-proy(Add(l,e), Cr-lista()) = Cr-lista()
Cr-lista-proy(Add(l,e), Add(l',e')) =
    if Pert-nom(Add(l,e), e') then
       Add(Cr-lista-proy(Add(l,e),l'), Obt-nom(Add(l,e),e')
    else
       Cr-lista-proy(Add(l,e), l')
    endif
```

Pert-nom es una función que dados una lista[info] y un nombre de atributo verifica si éste pertenece a la misma. Obt-nom es una función que dados un lista[info] y un nombre de atributo, selecciona el atributo de la lista correspondiente al mismo.

En este caso, se construye una nueva subclase por medio de una operación. Notar que se esta definiendo una forma de construir nuevos tipos a partir de otros dados.

En el producto cartesiano clásico no solo es necesario conocer las componentes que definen al tipo que se quiere proyectar sino que además debe realizarse una nueva especificación en la que se define la estructura del tipo obtenido de la proyección.

Ejemplo:
Si a partir del tipo CUENTA CORRIENTE (ver ejemplo de parte 3) se quisiera obtener un nuevo tipo que solo tuviera el número y el nombre de esta, entonces lo que se debería hacer en cada caso es:

i) Aplicar la siguiente operación:

```
proyectar( (CUENTACORRIENTE, l, c), {Numero, nombre}, DESCUE )
```

ii) Definir un nuevo tipo:

```
TAD DESCUE

cr_descue:  REAL X STRING ----> DESCUE.
```

Capítulo 4

Base de Datos

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Un Modelo de Administración de Buffers para la Predicción del Desempeño de Sistemas de Bases de Datos *

Ignacio R. Casas
Departamento de Ciencia de la Computación
Escuela de Ingeniería
Universidad Católica de Chile
Casilla 6177, Santiago, Chile
Fax: 56-2-5510046 Telex: 240395 PUCVA CL
e-mail: icasas@puc.cl o ...uunet!uchdcc!juncal!icasas

Kenneth C. Sevcik
Computer Systems Research Institute
University of Toronto
Toronto, Ontario M5S 1A4
Canada

Resumen

El diseño de sistemas de bases de datos es muy complejo debido al gran número de soluciones de compromiso que afectan la futura eficiencia y desempeño del sistema. Es una necesidad, entonces, el poder contar con herramientas de modelación de desempeño que ayuden en las distintas fases del desarrollo de un sistema de bases de datos.

El desempeño de un sistema de bases de datos se mide, principalmente, en base a tiempos de respuesta de transacciones y cantidad de accesos a memoria secundaria. En este contexto, uno de los aspectos importantes en la eficiencia de un sistema de bases de datos es el comportamiento del administrador de buffers el cual debe reducir los accesos a memoria secundaria que generan las distintas transacciones. En este artículo se presenta un modelo de desempeño de la administración de buffers, como parte de un model generalizado para la predicción del desempeño de sistemas de bases de datos. El modelo generalizado permite comparar decisiones de diseño, y sus correspondientes medidas de desempeño, en distintos niveles de abstracción del sistema computacional.

Se presenta la hipótesis de que las referencias a una base de datos se comportan de acuerdo a la distribución empírica Bradford-Zipf. Para fundamentar esta hipótesis, se presenta evidencia empírica obtenida de la observación y medición, durante cinco días, de un sistema de bases de datos en un ambiente de producción. Se presentan también los resultados de validación del modelo, con los datos obtenidos del monitoreo del sistema de bases de datos de producción.

* Esta investigación ha sido financiada por el Consejo Candiense de Investigación en Ciencias Naturales e Ingeniería (NSERC-Canada) y por la Dirección de Investigación de la Pontificia Universidad Católica de Chile (DIUC-Chile).

1. Introduction

Database (DB) systems are difficult to design and evaluate. The efficiency and performance of these systems is substantially affected by many decisions required throughout the development stages. Unfortunately, the impact on performance of many design alternatives is not yet well understood.

A number of comprehensive database performance models and methodologies have been proposed to help in the selection among design alternatives ([Miya75], [Naka75], [Sevc81], [Teor76], [Yeh77]). A weak point of these models has been the evaluation of the effect of buffer management on the performance of the database systems. Usually, it is assumed that block (page) references are uniformly distributed over all the database blocks. This assumption causes the use of buffering to appear relatively ineffective. Consequently, the performance predictions produced by models that use this assumption are unrealistically insensitive to the selection of buffer sizes and allocation strategies.

It is generally accepted that the buffer manager of a DB system can significantly affect the overall performance of the database system and computer system on which it runs. For this reason, in this paper we propose an improved submodel of buffer management for use within a comprehensive database performance prediction methodology. Our results will show that it is necessary to make a more realistic asssumption about the distribution of block accesses, but that the specific replacement policy used for buffer management need not be modelled in detail. We replace the assumption of uniform distribution of block accesses with the assumption that block references are distributed across database blocks according to a Bradford-Zipf (B-Z) distribution ([Broo69]).

Our analytic model will be based on the assumption of a random replacement policy for buffer management. Measurements of an operational database management system will be presented to validate our model, showing both that empirical block access distributions have a B-Z form and that conclusions based on a random replacement policy are indicative of those based on a variety of replacement policies used in real systems. We conclude that the resulting simple analytic model represents a suitable tradeoff between simplicity and accuracy in choosing a buffer management submodel for use in the context of a comprehensive database system performance prediction package.

In the next section we briefly describe a multi-level database performance model into which the buffer management model proposed in this paper can be embedded. While the terminology and notation used are consistent with that of the database literature, analogies with the virtual memory problem are indicated when appropriate. In section 3 we discuss the main differences between DB buffer management and OS virtual memory management. In section 4 we provide some empirical evidence for the operational hypothesis that database block references are Bradford-Zipf (B-Z) distributed. In section 5 we describe the database buffer management submodel and report the results of validating it against data from a real-life environment.

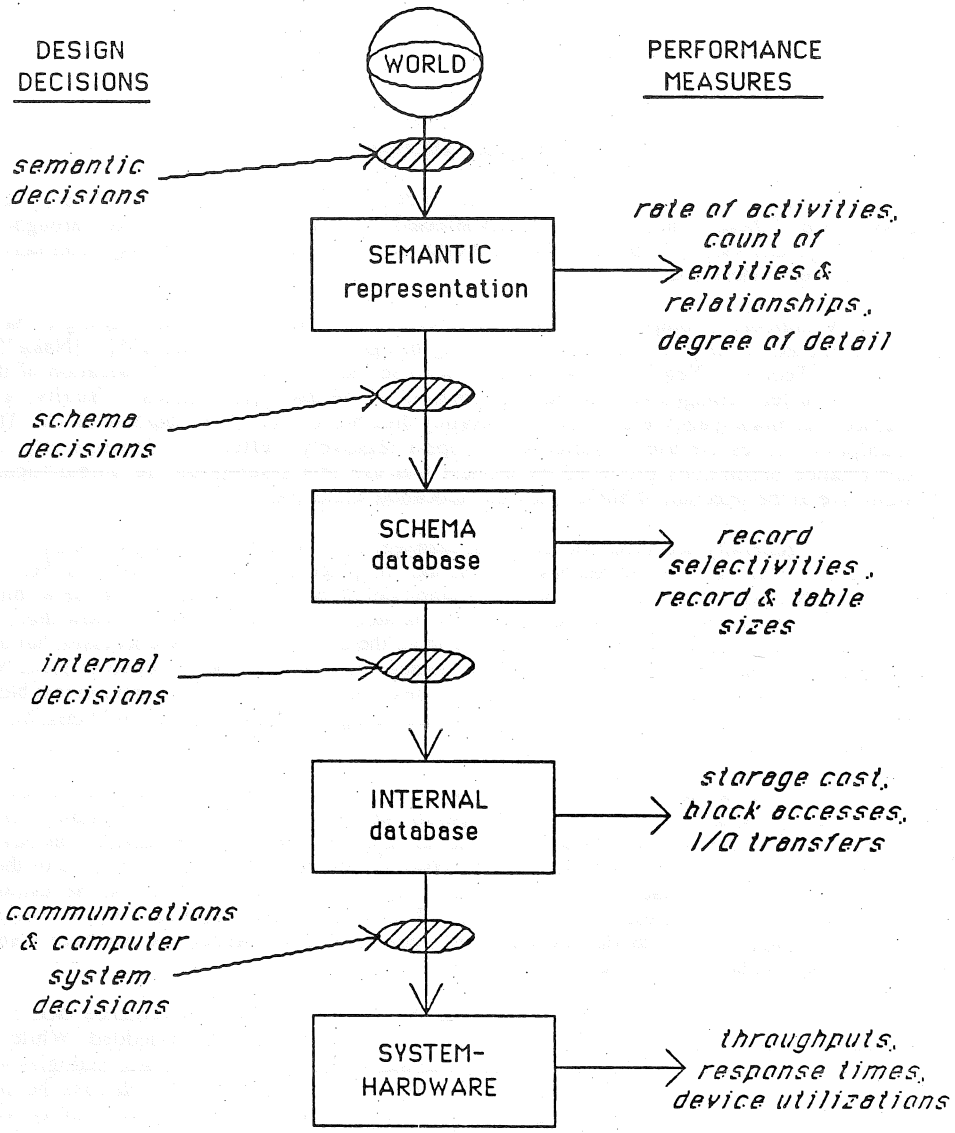


Figure 1: Layered Database Performance Model

2. A Layered DB Performance Model

In this section we briefly describe a layered performance model that provides a means of formal comparison of database design alternatives. A detailed description of the model and its application to actual systems is given elsewhere ([Casa86]).

The database design process can be viewed as the gradual transformation of an abstract data model into successively more concrete representations. Several levels of abstraction can be distinguished, where each successive level uses the decisions demarcated at this and higher layers, to represent the environment in more detail. As shown in Figure 1, we identify four levels of database description: semantic, DB schema, DB internal, and system-hardware. The bottom level, the most concrete one, includes the hardware and system software that runs the database application.

A set of design decisions is identified at each layer of the model, such that the representation at the previous level can be transformed into a more concrete representation at this level. Central to this hierarchical representation are models of computation (defined at each level) that provide performance measures to evaluate design choices.

A four-layer structure was chosen for several reasons. Firstly, levels are distinguished by the nature and longevity of the design decisions. For instance, decisions at the system-hardware level refer to the computer system where the database would eventually reside (e.g., file placement on devices, and processor scheduling strategies). The DB internal level refers to decisions relevant to the choice or design of a DBMS (e.g., file structures, access paths, and buffer management), while decisions at the DB schema level refer to the way the abstract data from the semantic level (e.g., entities and relationships) is represented in a formal logical data model (e.g, a relational schema). Secondly, this four-level structure can be naturally transformed into a software model for DBMS-specific performance predictors.

The first transformation layer of the system (see figure 2) involves decisions that lead to a formal specification of the entities to be represented in the database, the attributes of each of these entities to be recorded, and the types of operations (queries and updates) to be carried out. Criteria that can be used to evaluate the choices made at this level, with respect to their ultimate impact on system performance, include counts of the entities to be represented and rates of enquiry and update activities.

The second transformation layer (see figure 3) involves decisions that lead to a logical database design in the context of a chosen logical data model. Also, classes of transactions that perform database updates or pose queries are characterized as parameterized templates expressed in the data manipulation language associated with the logical data model. At this level, the number of records of each type and their sizes are determined. Also, for each transaction class, the rate of transactions is known, and the average number of records accessed can be estimated .

The third transformation layer (see figure 4) includes decisions that lead to a physical database design. It is at this level that the buffering strategy is represented. After determining the file structures and supporting access structures for the database, along with the encoding of database attributes, blocking factors for various portions of the database can be chosen. This provides enough information to estimate the average number of block accesses to each file structure component for each class of transactions. If buffers are used to hold some blocks in main memory for one use to the next, then the average number of physical block transfers (I/O operations as opposed to block accesses or references) per transaction can be reduced. The

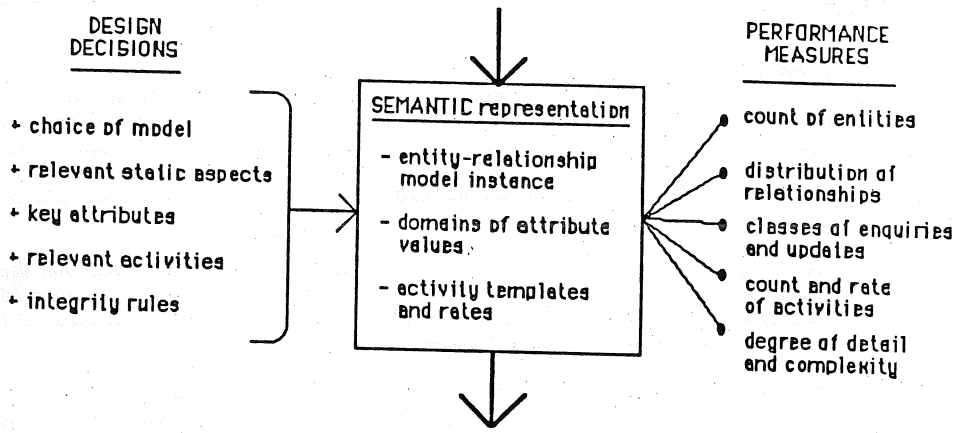


Figure 2 : Design Decisions, Representation, and Performance Measures at the Semantic Layer

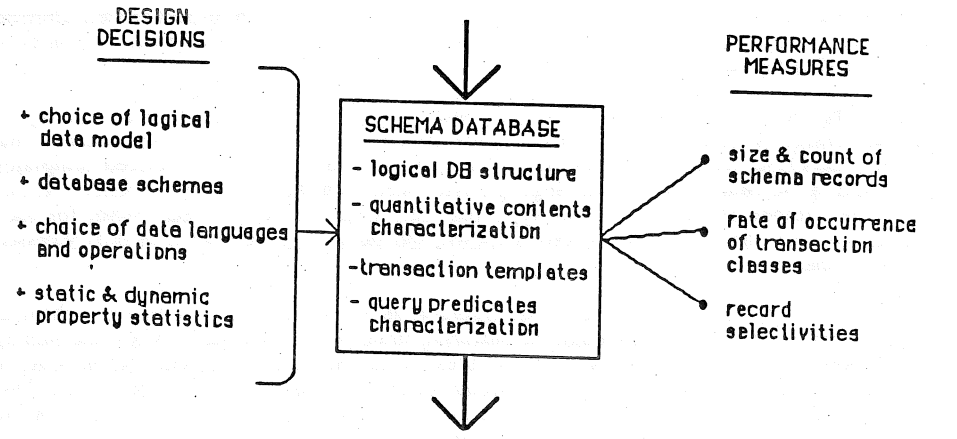


Figure 3 : Design Decisions, Representation, and Performance Measures at the Scheme Layer

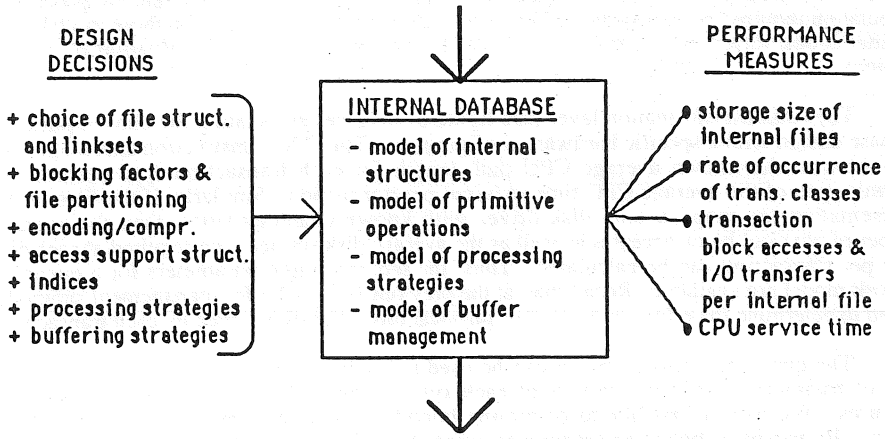


Figure 4 : Design Decisions, Representation, and Performance Measures at the Internal Layer

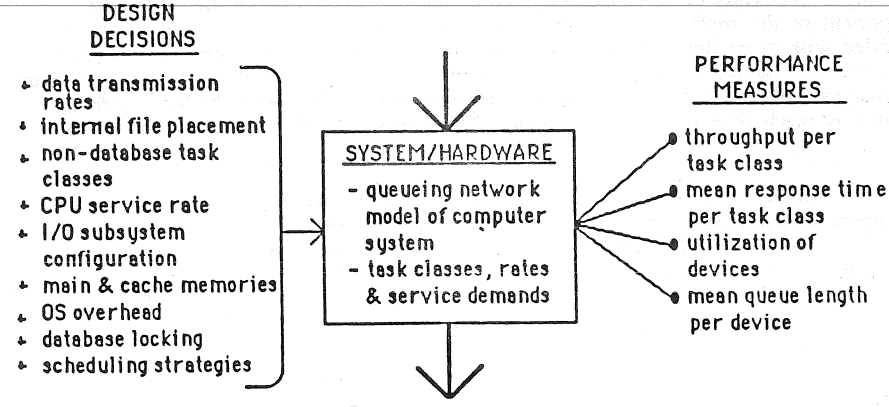


Figure 5 : Design Decisions, Representation, and Performance Measures at the System-Hardware Layer

larger the buffer size chosen to hold blocks of one part of the file structure, the greater the reduction in number of physical block accesses per transaction. At this layer, the evaluation of all design decisions made to this point is based on the expected number of physical block transfers required per transaction (and, to a lesser extent, on the amount of processor computation required per transaction). This is an improvement with respect to those models that consider block references instead of physical block transfers for the evaluation of design decisions.

The final transformation layer (see figure 5) involves associating the chosen physical database design with a specific hardware configuration. The CPU's instruction execution rate, together with the known average CPU path length for each transaction type permits the determination of the average CPU time required per transaction. Similarly, if the files of the representation are mapped onto disk drives with known characterisitics, then the average number of physical block accesses as well as the average disk service time required at each disk drive per transaction can be calculated. Thus, the required input parameters for a queueing network model are available. Recall that, at the previous layer, a buffer management submodel is used to determine the effective number of physical block transfers issued by each transaction.

The queueing network model can be used to predict average response times for each class of transactions, and utilizations of each system device, as well as other performance measures. We would first like to determine bounds on the performance measures of the system. By means of bounding techniques, upper bounds on throughput and lower bounds on mean response times can be obtained. These bounds are easy to calculate, even by hand, and they give a rough idea of the behaviour of the system. Thereafter, if these bounds show that performance would satisfy requirements and expectations, the model can be solved to estimate throughputs and mean response times for each task class, and the utilization and mean queue length by task class for each device.

At each level of the multi-layered model, simplifying assumptions are required to make the transformation tractable. Because of this, the design of various components of the multi-layered model must be balanced. Each component must achieve sufficient accuracy to keep it from limiting the overall accuracy desired from the model, and it should do so without requiring unnecessary detail or excessive computation. In the case of the buffer management component of the model, this balance requires that the distribution of block accesses be modelled with more detail than simply assuming it to be uniform. On the other hand, the assumption of "random" as the buffer replacement model simplifies both parameter specification and analysis while not reducing accuracy to the point where it becomes the limiting factor in the accuracy of predictions of the overall model. The facts will be demonstrated in the sections that follow.

In the next sections we describe the buffer management submodel and present measurements that indicate that actual DB block access distributions have a Bradford-Zipf form. We first discuss the main differences between DB buffer management and OS virtual memory management. The application of the multi-layered model to several commercial DBMSs, including the design, implementation and validation of a performance predictor for System 2000 databases is reported elsewhere ([Casa86], [Casa87]).

3. DB Buffering and OS Paging

Even though there are many similarities in the activity of DB transactions using a buffer pool and the activity of concurrent programs using virtual memory, important differences have motivated the particular study of buffer allocation and replacement algorithms within the context of data base management systems. In this section we discuss some of these differences.

As a first issue we note the importance of active and passive elements in both the virtual memory and DB context. That is, while a virtual memory operating system (OS) assigns physical memory partitions to software programs (active elements), some traditional DB applications allocate buffer partitions to internal files. In the latter case, buffers are assigned to the passive elements (internal files) rather than to the active elements (transactions) of the DB system. The System 2000 DBMS, for instance, establishes a maximum number of buffer frames for each internal file. If the buffer limit for a given file is reached, when performing some I/O transfers, then the least recently accessed buffer from the set of buffers currently associated with that file is chosen for replacement.

A combination of buffer partitions to passive and active DB elements can be used as well. For instance, the buffer management strategy for relational DB systems proposed by Chou and DeWitt ([Chou85]) subdivides the buffer pool and allocates it on a per file (relation) basis. Each query is associated with a distinct active instance of a file. Each active file instance is then given a distinct buffer pool partition.

A second issue refers to the fact that concurrent DB transactions frequently share a set of pages in the buffer pool. Indeed, the concurrent execution of DB transactions may increase the probability of pages being re-referenced in the buffer pool ([Effe84]). This effect is referred to as *inter-transaction locality* ([Maga81]). Consequently, the locality of database references depends not only on individual transactions but it is also dependent on the degree of data sharing of the database workload. (*Intra-transaction* locality is the locality exhibited in the page references issued by a single database transaction.) Although a similar situation may occur in a virtual memory context where programs can share code pages, the potential for data sharing is greater in the database context.

A third issue is the existence of well-defined internal access paths in a DBMS which permits the prediction of some aspects of the reference behaviour of transactions ([Chou85], [Sacc82]). It is often the case that DB pages containing specific indices, special directories, etc., are more often referenced than other data pages. In relational systems, for instance, a query optimizer uses pre-defined access plans such that the patterns of generated page references are regular and can be predicted.

Finally, a fourth issue deals with the fact that all references to records within a database page are treated as a single page (block) reference, while each machine instruction that specifies a program page accounts for a page reference ([Effe84]). Indeed, immediate re-references to a database page within the execution of a transaction are usually not considered as part of the database locality behaviour. This is so because records within that page can be addressed without a new block access or I/O transfer.

In retrospect, all the described differences between reference bahaviour of DB systems and programs under virtual memory have motivated the study of buffering strategies for database systems in particular. In spite of these differences, many of the replacement algorithms developed for virtual memory paging may be adapted for DB buffer management, due to the locality observed in database reference strings ([Effe84], [Sacc82]).

4. Locality of Reference

As in program reference behaviour, locality of database (DB) references represents the tendency of DB workloads to concentrate their block (page) references heavily into subsets of the DB stored blocks. Several empirical studies of operational DB systems indicate the presence of locality patterns in workload block references ([Effe84], [Hawt79], [Kear83], [Maga81]). The buffer manager of a DBMS, which attempts to exploit this locality behaviour, can significantly affect the overall performance of the computer system.

Within the context of the multi-layered DB performance model of the previous section, a model of DB buffer management is necessary for predicting the miss ratio at the buffer pool of a DB system. Once this ratio is known, the I/O transfer count (number of physical block accesses) for distinct transactions can be estimated. Prior to discussing a buffer management performance model in more detail, we discuss in this section the type of locality that empirical DB block references present.

Bunt and other researchers ([Bunt84]) have proposed a measure of program reference locality based on the Bradford-Zipf (B-Z) distribution ([Broo69]). This measure is based on the following expression:

$$L = B \times \bar{n}$$

B is intended as a measure of the *concentration* or *clustering tendency* of locality, while $\bar{n}$ reflects the *persistence tendency* of locality. B is a function of the proportion of highest productivity items that contribute to half the total productivity. $\bar{n}$ is defined as the mean number of consecutive references to a set of pages.

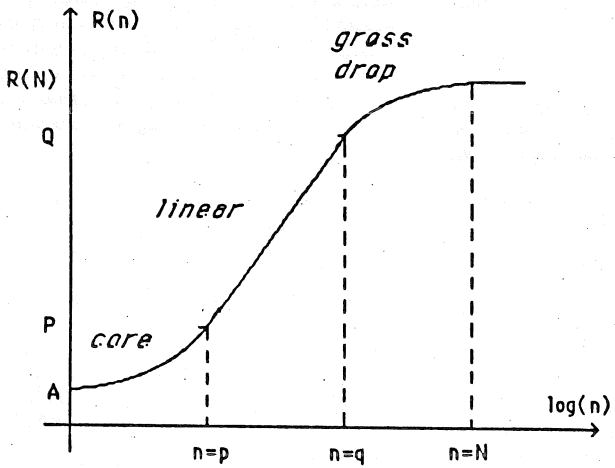


Figure 6: A Typical Bradford-Zipf Distribution ([Bunt84])

A typical B-Z distribution is depicted in figure 6 ([Bunt84]) where the natural logarithm of the rank of the items *(log(n))* has been plotted against the cumulative

productivity (relevance) of the items *(R(n))*. Items in the sample are ranked in order of decreasing productivity. Phenomena exhibiting behavioral patterns that fit the B-Z distribution include the yearly circulation of books or journals in a library, the number of articles on a given field in a given journal, the number of references to the article in subsequent journals, and the distribution of wealth, among others.

In the realm of computer systems, the Bradford-Zipf distribution has been used, for instance, to model information retrieval ([Benn75]), and to model the distribution of search keys in hashing tables ([Sams78]). More recently, Peachey ([Peac82]) has applied the B-Z distribution to examine the page reference patterns of executing programs. By plotting the natural logarithm of the rank of the pages against the cumulative productivity (frequency of references) for a series of program traces, Peachey observed a B-Z type of distribution. The assumption that program references are Bradford-Zipf distributed has been used as an operational hypothesis to analyze program restructuring methods, for the generation of synthetic program reference strings, and for the identification and analysis of program phase behaviour ([Peac82], [Bunt84]).

Motivated by Peachey's findings and the similarities between a database buffering setting and a virtual memory paging setting ([Effe84]), we have investigated the reference pattern in an operational database environment. We used the block reference strings generated during a 5-day-long observation period of five System 2000 databases (with a combined size of about 200 Megabytes) in a production setting. Daily average block reference frequencies were obtained for accesses to each of the DB internal files and for the databases as a whole ([Maga81]).

In this paper we describe only the results obtained with one of the databases. Similar results were obtained for the other four databases. The reported database is 1 Megabyte in size and activities of a single day include, on average, 361 commands executed, 41 seconds of CPU, and 27,556 block (page) references (block size is 2492 bytes).

A System 2000 database is composed of six internal files ([Casa86]). Given the measured block reference strings and frequencies, a graph of cumulative block reference productivity was obtaind for each file. Thereafter, for each of the measured cumulative productivities we have produced a corresponding hypothetical (fitted) B-Z distribution. To do this, we have used the B-Z formulation proposed by Brookes ([Broo69]):

$$R(n) = A \times n^{b} \qquad (A \leq R(n) \leq P)$$

$$R(n) = k \times \log(n / s) \qquad (P \leq R(n) \leq Q)$$

Parameter b determines the curvature of the "core" portion of the graph (Figure 6). P is the cumulative productivity at which the graph becomes linear. For the "gross drop" portion of the graph (Figure 6) we use:

$$R(n) = Q + (1 - e^{(q - n)}) \times m \qquad (Q \leq R(n) \leq R(N))$$

Q is the point at which the drop of the graph begins, and N represents the number of productive blocks.

Given these formulae, four points are needed to produce a fitted B-Z distribution (see figure 6): A, P, Q and N. These four points where determined for each of the measured block reference distributions of the System 2000 experiments, and corresponding fitted distributions

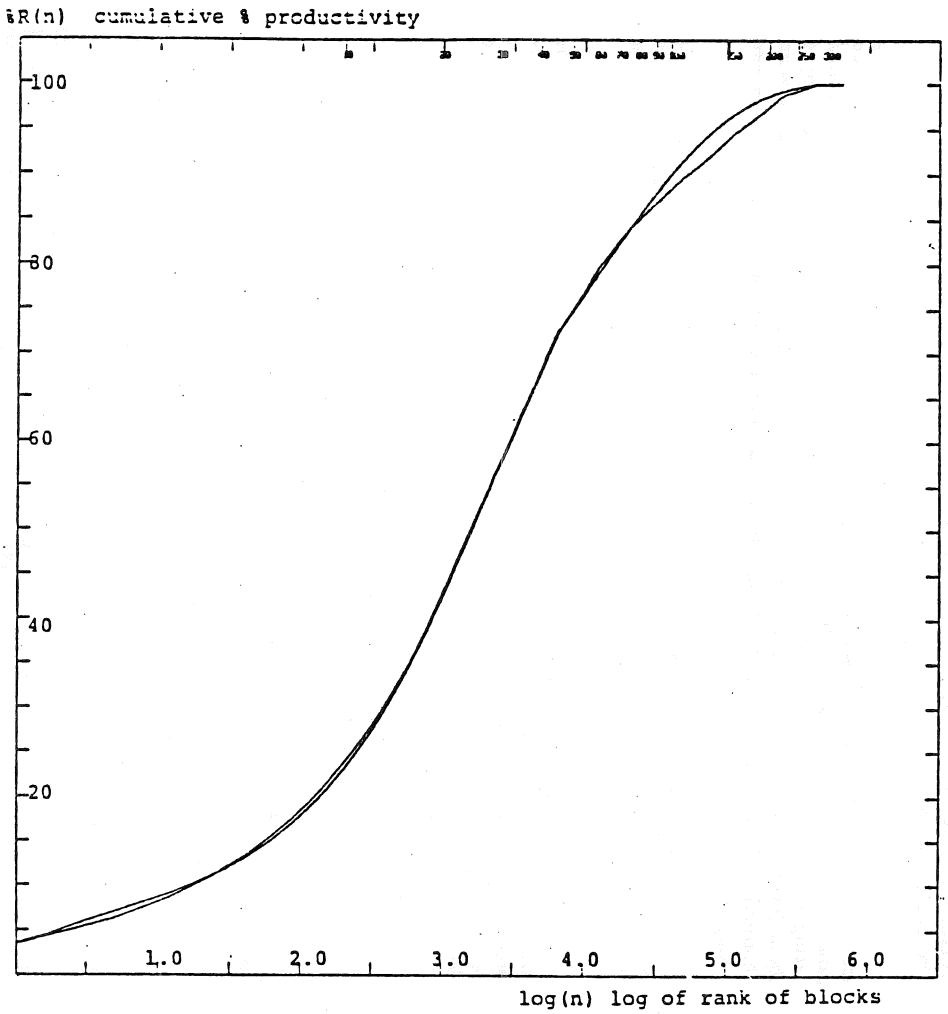


Figure 7(a)

Database B, All Files
Measured and
Hypothetical Distrib.

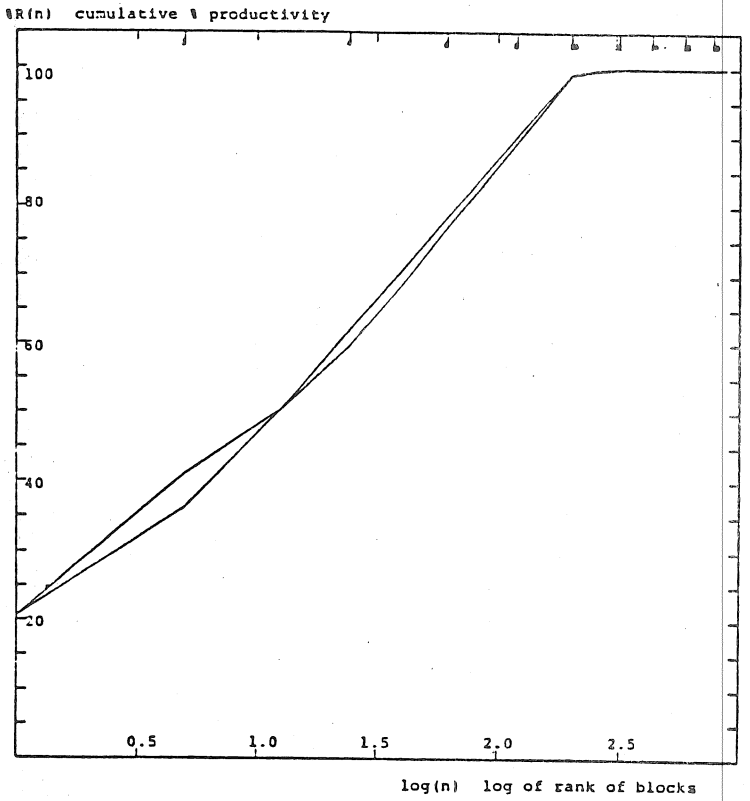


Figure 7(b)

File 2
Measured and
Hypothetical Distr.

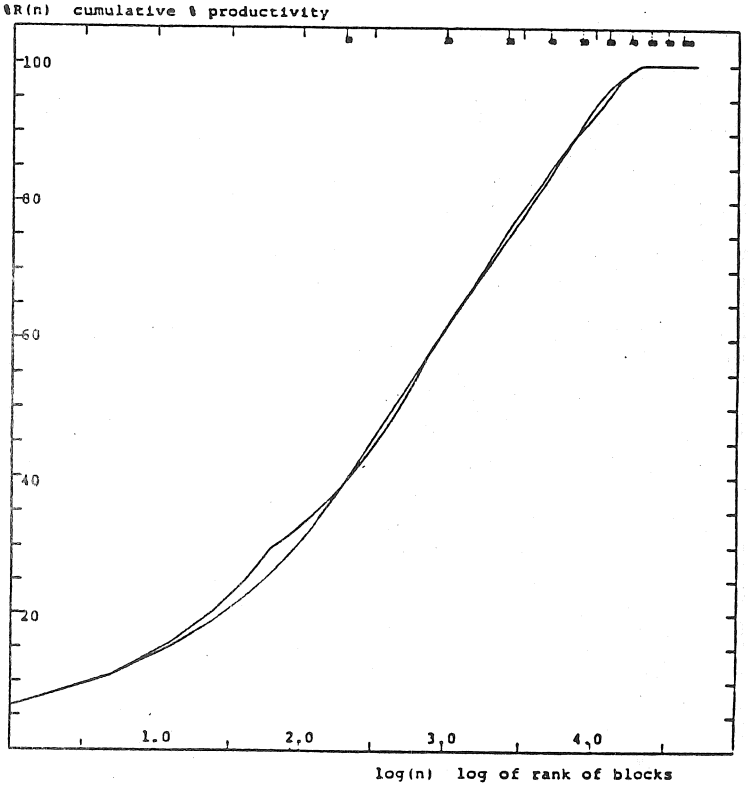


Figure 7(c)

File 4
Measured and
Hypothetical Distr.

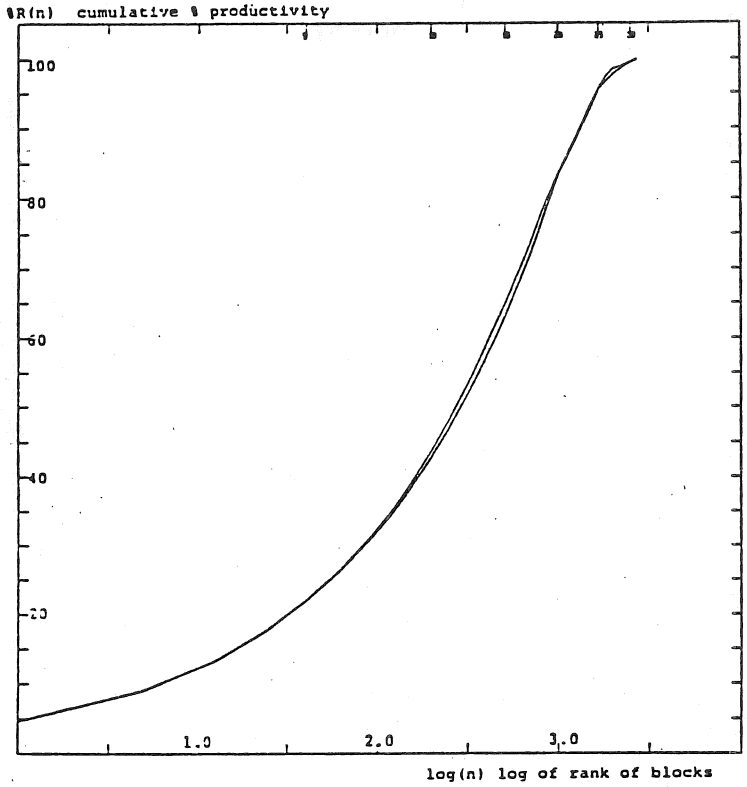


Figure 7(d)

File 5
Measured and
Hypothetical Distrib.

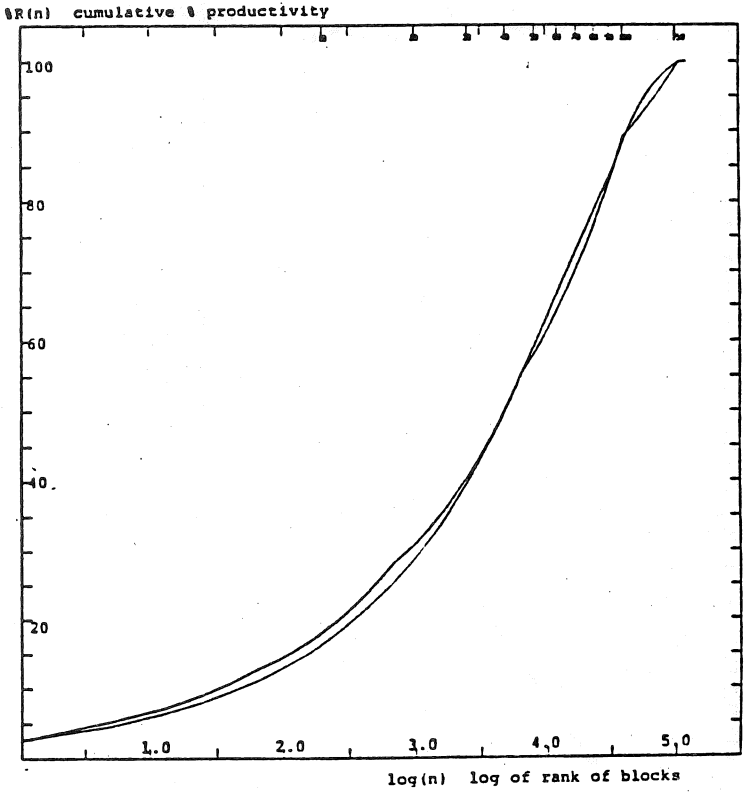


Figure 7(e)

File 6
Measured and
Hypothetical Distrib.

were generated. Figures 7(a) through 7(e) show the measured distribution (for the internal files of one of the System 2000 databases) compared to the corresponding fitted distribution. All five graphs indicate that the block reference behaviour of the database application closely approximate a B-Z type of distribution. A Kolmogorov-Smirnov test of goodness of fit was conducted for the observed distributions. The hypothesis of Bradford-Zipf distribution was accepted for all cases of the test with a level of significance of 95%. Similar results were obtained for the block reference distributions of the other four System 2000 databases.

As we mentioned earlier, most previous comprehensive performance models for database systems have used the simple (possibly simplistic) assumption of uniformly distributed accesses over database blocks. This assumption limited the ability of those models to reflect the performance improvement achievable through buffering. The B-Z distribution proposed here requires only a few parameters, but allows block access distributions to be represented with much greater accuracy.

5. A Performance Prediction Submodel of Buffer Management

Assuming then a Bradford-Zipf-like distribution of database block references, we derive now expressions that estimate the miss ratio achieved by a buffer manager. The buffer model is then validated against data from a real-life, operational database system. Model results are compared with results obtained using the traditional assumption of uniform distribution of references over the blocks of the database.

We assume that block references are independent. Even though not realistic, the independent-reference assumption simplifies analysis and, as we shall see, seems to allow fairly good performance estimates. Let us define the following terms:

TOTN	:	size (total number of blocks) of the database or internal file under observation
N	:	number of productive blocks; i.e., the number of distinct blocks that are referenced during a conveniently large observation period *(N ≤ TOTN)*
b	:	(variable) number of buffers (frames) in the buffer pool ($b \geq 1$)
s	:	allocation/replacement strategy used by the buffer manager
P	:	an observed block reference string

The miss ratio $M(s,b,P)$, a function of the buffer pool size (b), the allocation/replacement strategy (s), and the block reference string (P), is defined as

$$M(s,b,P) = 1 - H(s,b,P)$$

where $H(s,b,P)$ represents the hit ratio; that is, the proportion of times that a requested block is found in the buffer pool.

Given the experimental evidence from program behaviour and the locality behaviour displayed by database references, we expect that the miss ratio for an hypothetical "Uniform" reference string, that references each database block with probability $1/TOTN$ at each reference, provides a pessimistic approximation to the miss ratio of a real-life reference string under some practical buffering strategy. For this "Uniform" reference string, the hit ratio for any replacement strategy s is given by

$$H(s,b,T_Unif) = \frac{b}{TOTN}$$

T_Unif denotes a block reference string with uniform probability of reference over all *(TOTN)* blocks of the database.

Given that N represents the number of productive blocks, let *N_Unif* denote a block reference string with uniform probability of reference over the productive N blocks of the database. It is assumed that the remaining $TOTN - N$ blocks are not touched at all by the reference string. We would expect the miss ratio for *N_Unif* to be better (lower) than for *T_Unif*, but still, it would be a pessimistic approximation to the miss ratio of a real-life reference string. For all replacement strategies s, the hit ratio achieved by *N_Unif* is given by

$$H(s,b,N_Unif) = \frac{b}{N}$$

If, given a reference string P, p_i (where $i=1, \ldots, N$) represents the relative frequency of accesses to block i (e.g., as given by a Bradford-Zipf distribution), and assuming a replacement strategy (with global allocation) *FIX* that fixes in the first *(b-1)* buffers the most frequently used blocks and uses the remaining buffer to read in other blocks, the hit ratio is given by

$$H(FIX,b,P) = \sum_{i=1}^{b-1} p_i + \frac{p_b^2 + \cdots + p_N^2}{\sum_{k=b}^{N} p_k} \qquad ;b \leq N$$

For large databases (with a large number of productive blocks) we can approximate the hit ratio achieved by P under *FIX* by

$$H(FIX,b,P) \approx \sum_{i=1}^{b-1} p_i + \frac{p_b^2}{\sum_{k=b}^{N} p_k} + \frac{\left(\sum_{k=b+1}^{N} p_k\right)^2}{(N-b) \times \sum_{k=b}^{N} p_k}$$

H(FIX,b,P) may give an optimistic approximation to the real hit ratio rendered by actual replacement algorithms, including RANDOM and LRU, for real-life database reference strings. A compromise between the hit ratio for the *N_Unif* string *(H(s,b,N_Unif))* and the hit ratio achieved by *FIX* could represent a better estimate of the real hit ratio. For this reason, we assume an hypothetical replacement strategy, *BZ*, whose hit ratio is estimated as a weighted average of *N_Unif* and *FIX*, as follows:

$$H(BZ,b,P) = \frac{1}{2} \times \left(H(s,b,N_Unif) \times (1-Q) + H(FIX,b,P) \times Q \right)$$

$$+ \frac{1}{2} \times H(FIX,b,P)$$

That is,

$$H(BZ,b,P) = \frac{b}{N} \times \frac{(1-Q)}{2} + H(FIX,b,P) \times \frac{(Q+1)}{2}$$

where Q is given by

$$Q = \left[H(FIX,b,P) \right]^{10 \times (1 - H(FIX,b,P))}$$

Since Q is in the range (0,1), $H(BZ,b,P)$ is between $H(s,b,N_Uniform)$ and $H(FIX,b,P)$ for all practical values of b ($b \geq 1$). For small values of b, Q provides more weight to $H(s,b,N_Unif)$, while for bigger values of b it provides more weight to $H(FIX,b,P)$.

We compare now the estimated miss ratio curves (using the formulae for T_Unif, N_Unif, FIX, and BZ) for the System 2000 application, against the real miss ratio curves measured by Magalhaes ([Maga81]). To trace block reference strings, Magalhaes logged all the transactions issued to a System 2000 database during an observation period, and then re-issued them against the original database within an experimental environment. For the measurements, Magalhaes set the buffer pool size to one (only one frame), so that block references could be collected at read/write time ([Maga81]). (System 2000 does not provide a monitoring tool for recording all block references.)

For this reason, the block reference strings in Magalhaes' data contain all references to blocks except immediate re-references (i.e., references to the block just previously referenced). Some immediate re-references are included in Magalhaes data, however, because System 2000 allows the enforcement, on user request, of write operations whenever a record is updated, as opposed to writing the updated block at the next buffer fault (the normal case).

To take into account in our model the effect of a reduced block reference string, the hit ratio formulae were adjusted as follows:

$$H_{adj}(s,b,P) = \frac{H(s,b,P) - H(s,1,P)}{1 - H(s,1,P)}$$

Notice that with $b=1$, the hit ratio $H(s,1,P)$ represents the proportion of immediate re-references (over all references) in the buffer pool.

Figures 8(a) through 8(e) show the predicted miss ratios (based on the distribution of block reference frequencies measured by Magalhaes for one of the databases) and the

measured miss ratios (for LRU and RANDOM replacement strategies). Figure 8(a) represents the average over all the files of the database, while figures 8(b) to 8(e) each cover individual internal files.

Curves marked *N_Unif* are obtained for the "Uniform" reference string (considering only productive blocks), while curves *BZ* represent the miss ratio estimated with the formula for *H(BZ,b,P)*. Curves marked *L* and *R* represent the real miss ratios measured by Magalhaes for (global) LRU and RANDOM replacement strategies respectively ([Maga81]). Curves marked *T_Unif* represent the miss ratio for the Uniform reference string considering all blocks of the database (or internal file), as opposed to productive blocks only. Curves *FIX* indicate the estimated miss ratio for the FIX strategy.

It is interesting to notice from the measured miss ratio curves, how badly LRU performs in those cases when the buffer pool is not of an appropriate size to contain a whole locality set.

Table 1 presents an error analysis for the estimation formulae (*T_Unif, N_Unif, FIX,* and *BZ)* considering: (1) the average over all files of the database ("Whole DB"), (2) each file alone ("File x"), and (3) the average obtained for all observed cases ("Total Cases").

Table 1
Miss Ratio Curves Error Analysis
(with measured block reference distributions)

	Model	Whole DB	File 2	File 4	File 5	File 6	Total Cases
Avg.	T_Unif	32.2	26.8	33.9	6.6	31.0	27.3
Error	N_Unif	30.3	7.4	27.2	6.6	30.5	23.9
%	BZ	3.7	5.2	10.9	2.8	23.4	8.7
	FIX	10.8	9.5	1.0	5.3	16.3	8.9
Max.	T_Unif	57.4	48.5	50.0	18.5	37.9	57.4
Error	N_Unif	53.6	16.7	36.5	18.5	36.9	53.6
%	BZ	7.5	11.0	17.0	7.0	26.1	26.1
	FIX	14.7	19.0	3.3	11.1	18.4	19.0

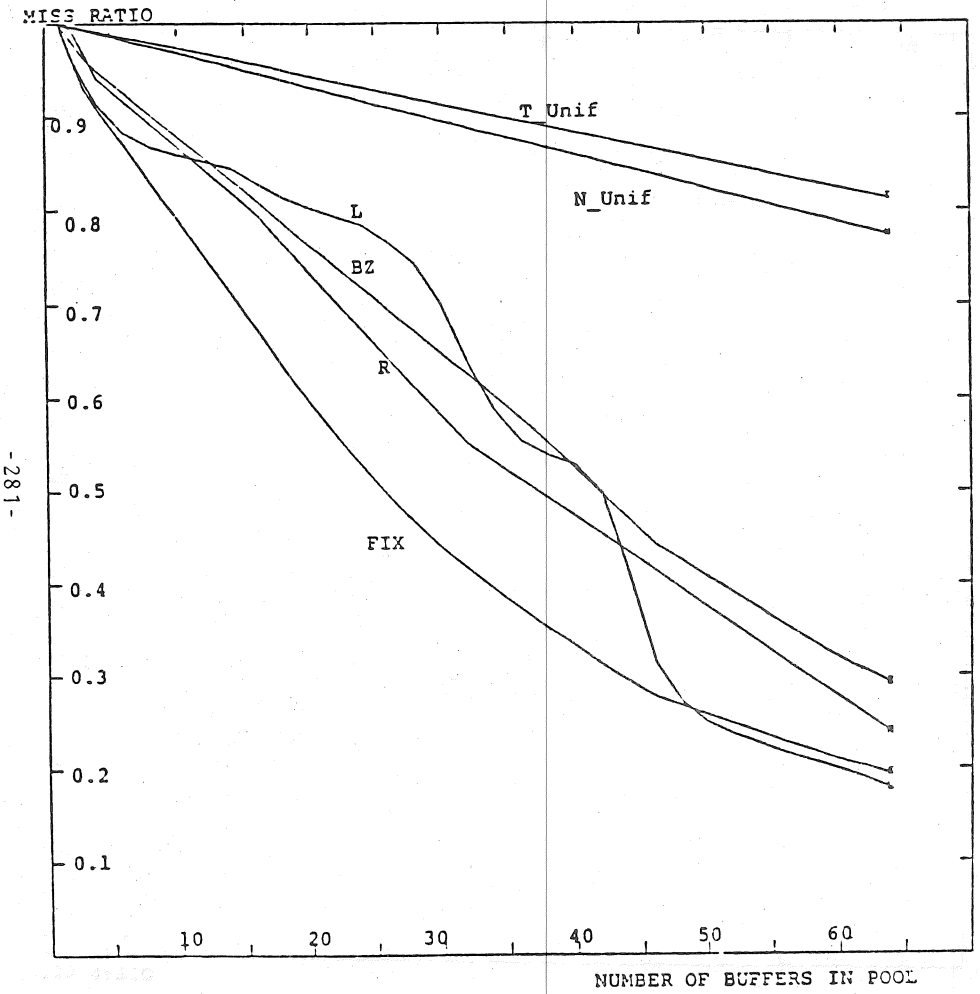


Figure 8(a)

Database B, All Files
Total # Blocks = 335
Prod. Blocks = 278

R =measured RANDOM
L =measured LRU
BZ =model estimated
FIX =model estimated
N_Unif =uniform over prod. blocks
T_Unif =uniform over all blocks

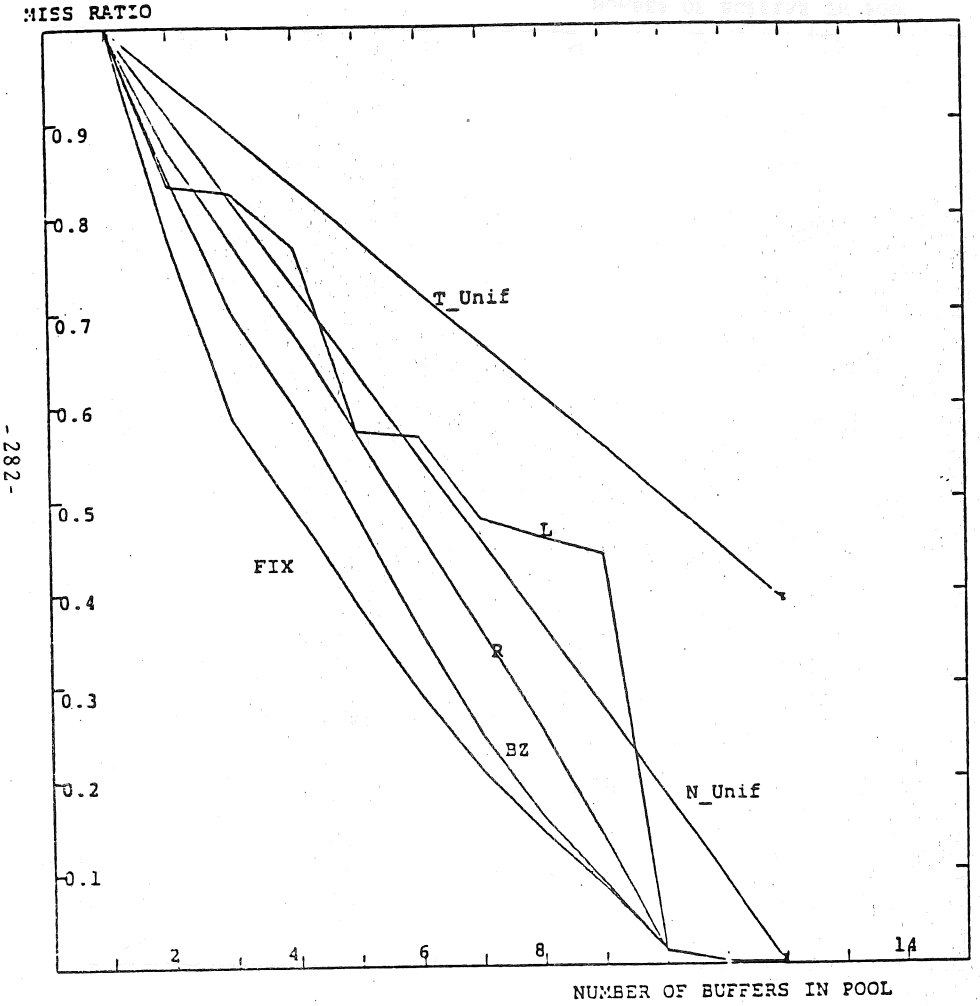


Figure 8(b)

File 2
Total # Block = 19
Prod. Blocks = 12

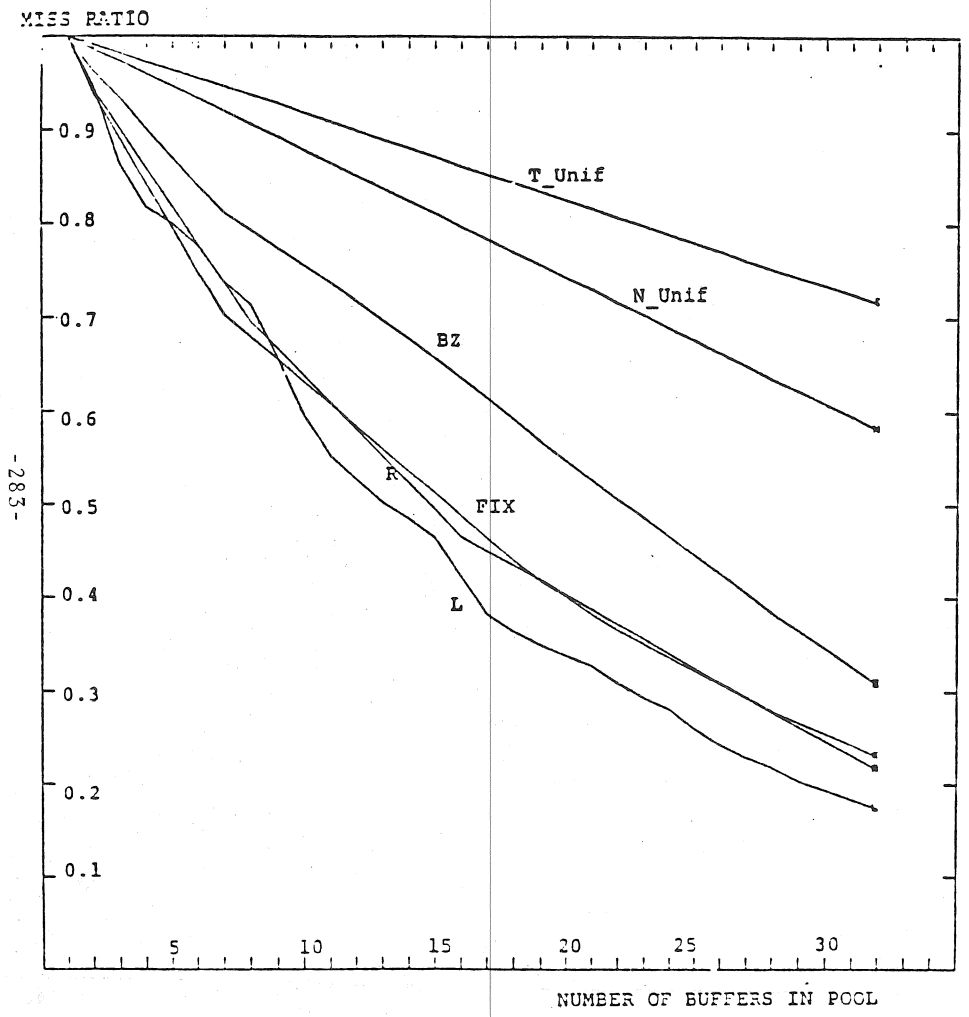


Figure 8(c)

File 4
Total # Blocks = 110
Prod. Blocks = 75

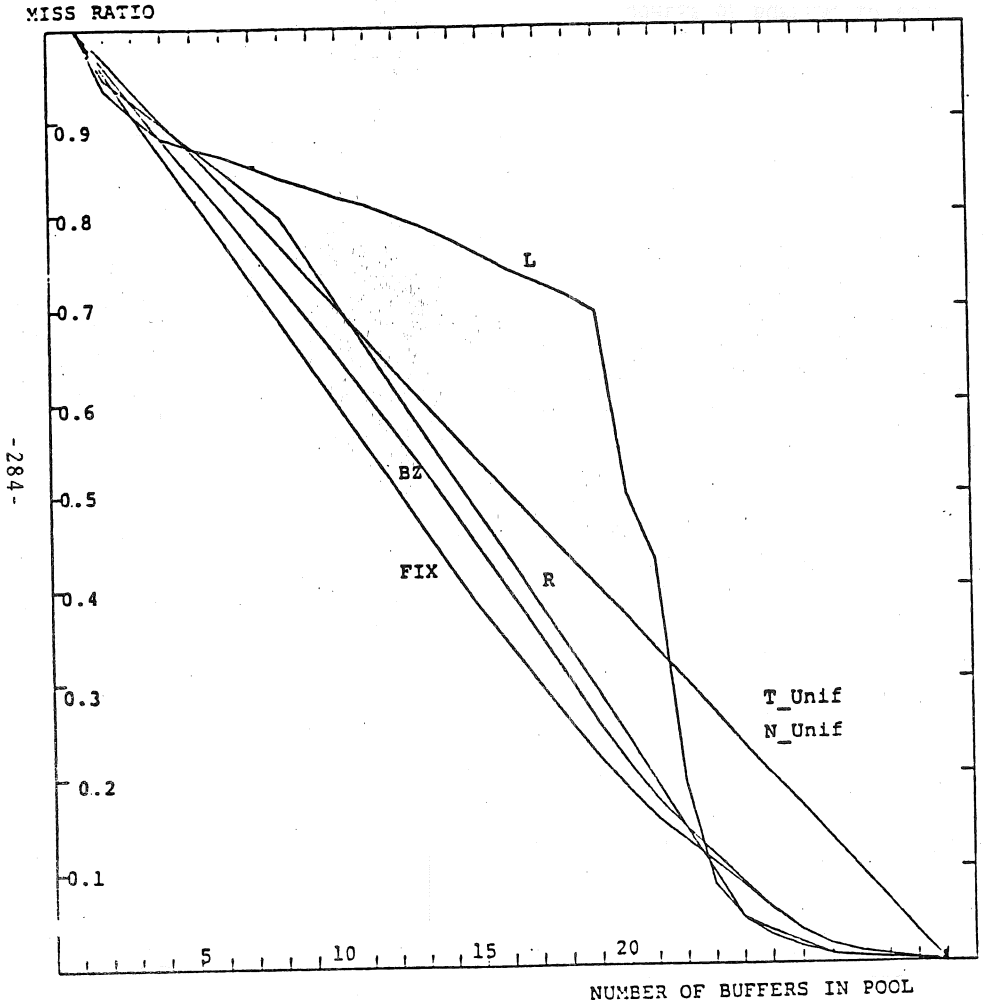


Figure 8(d)

File 5
Total # Blocks = 31
Prod. Blocks = 31

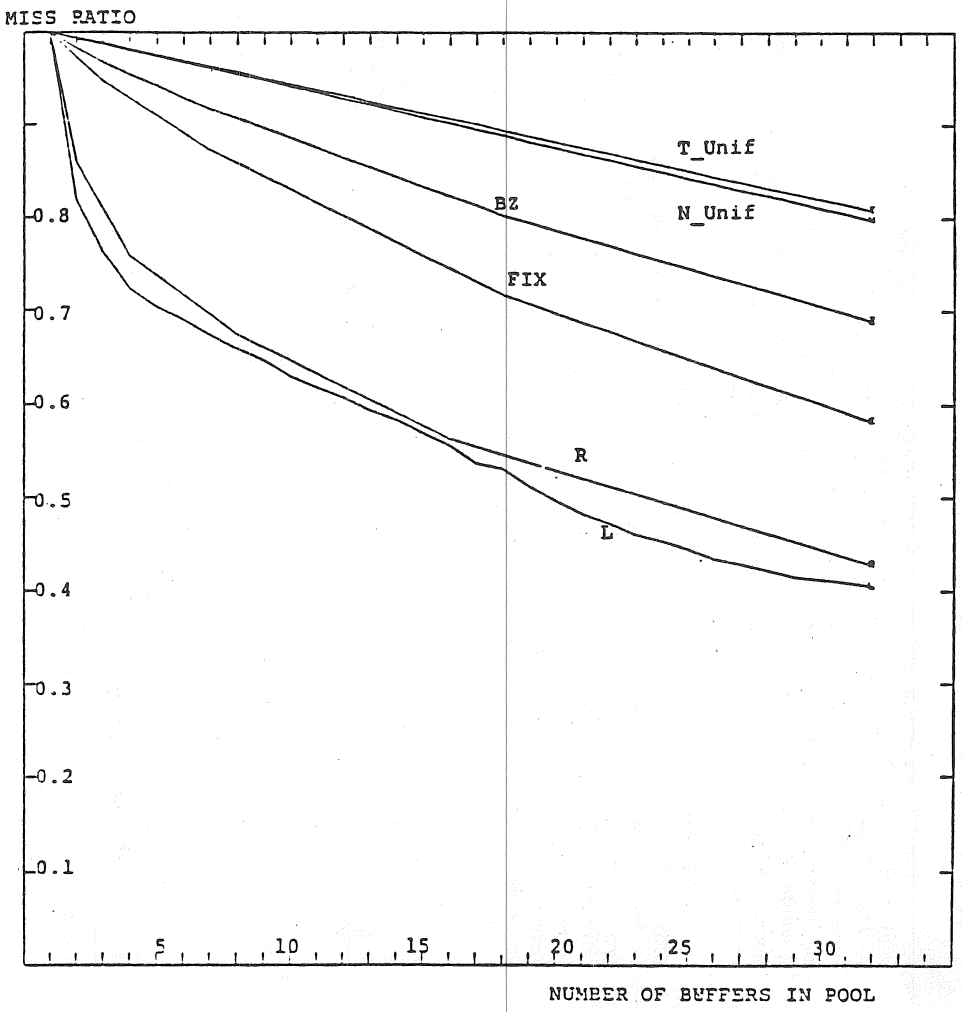


Figure 8(e)

File 6
Total # Blocks= 162
Prod. Blocks = 154

Using as a reference the miss ratio measured for the RANDOM replacement strategy, error estimates are computed as:

$$ERROR\ \% = \left| predicted\ M(s,b,P) - measured\ M(RANDOM,b,P) \right| \times 100$$

Maximum error percentages ("Max. Error %" in Table 1) represent the maximum errors rendered by the cost formulae in the observed cases. As expected, curve *N_Unif* (uniform references over productive blocks only) yields a pessimistic approximation to the real miss ratio (miss ratio *N_Unif* is higher than measured RANDOM miss ratio in 92.4% of all the observed cases). Still, *N_Unif* with a 23.9% average error and a 53.6% maximum error for all observed cases achieved a better approximation to the real miss ratio than *T_Unif* (uniform distribution of references over all blocks of the file or database) which rendered a 27.3% average error and 57.4% maximum error.

The *BZ* estimation delivered a 8.7% average error over all measured cases, with a maximum error of 26.1%, against a 23.9% average error and a 53.6% maximum errors of *N_Unif*. With respect to *N_Unif*, *BZ* represents a 63.9% improvement in terms of average error and a 51.3% improvement in terms of maximum error. Except for Files 4 and 6, curve *BZ* renders a better approximation to the real miss ratio than *FIX* does. For the database as a whole, *BZ* renders a 3.7% average error and a 7.5% maximum error against a 10.8% average error and 14.7% maximum error of *FIX*. A similar range of average and maximum errors was obtained for the other four databases in the study.

The least accurate estimates achieved by *BZ* are those for file 6 (23.4% average error, and 26.1% maximum error). This is probably due to the fact that the reference strings for file 6 contain a large number of insertions. New data is inserted at the bottom (or at empty spaces left by deletes) of file 6. Since there are many more insertions than deletions in the reference strings, new blocks (at the bottom of file 6) are referenced from day to day, and insertions continue to be done on the same block until the block is full. As a consequence, a strong correlation of references (persistence aspect of locality) is obtained which is not accounted for by the BZ formula. (This particular condition can be incorporated into our model by an appropriate separation of read and write block reference counts.)

The same argument applies to file 4. File 4 is only referenced once the required data records (and blocks) have been previously selected by means of content addressibility and data relatability operations performed on the other files of the database. The relative proportion of write operations is therefore greater on file 4 than on the other files of the database.

Also, we have conducted another set of experiments using this time the fitted B-Z block access distributions (as opposed to the measured distributions used in the previous experiments). In these experiments, we assume that we do not have complete information about block reference distributions, so we use hypothetical B-Z distributions that model the behaviour of the real database. Figure 9 shows the predicted miss ratios for *BZ* and *FIX* with the hypothetical distributions, and the measured LRU and RANDOM miss ratios. Notice that figures 8(a) and 9 are almost identical even though, in figure 9, the predicted miss ratio curves *(BZ* and *FIX)* were obtained using a fitted B-Z distribution, while in figure 8(a) the same curves were obtained using the measured block reference distribution. In both figures, curves *L* (measured LRU miss ratio) and *R* (measured RANDOM miss ratio) are exactly the same.

Table 2 presents an error analysis for the model predictions using the fitted B-Z

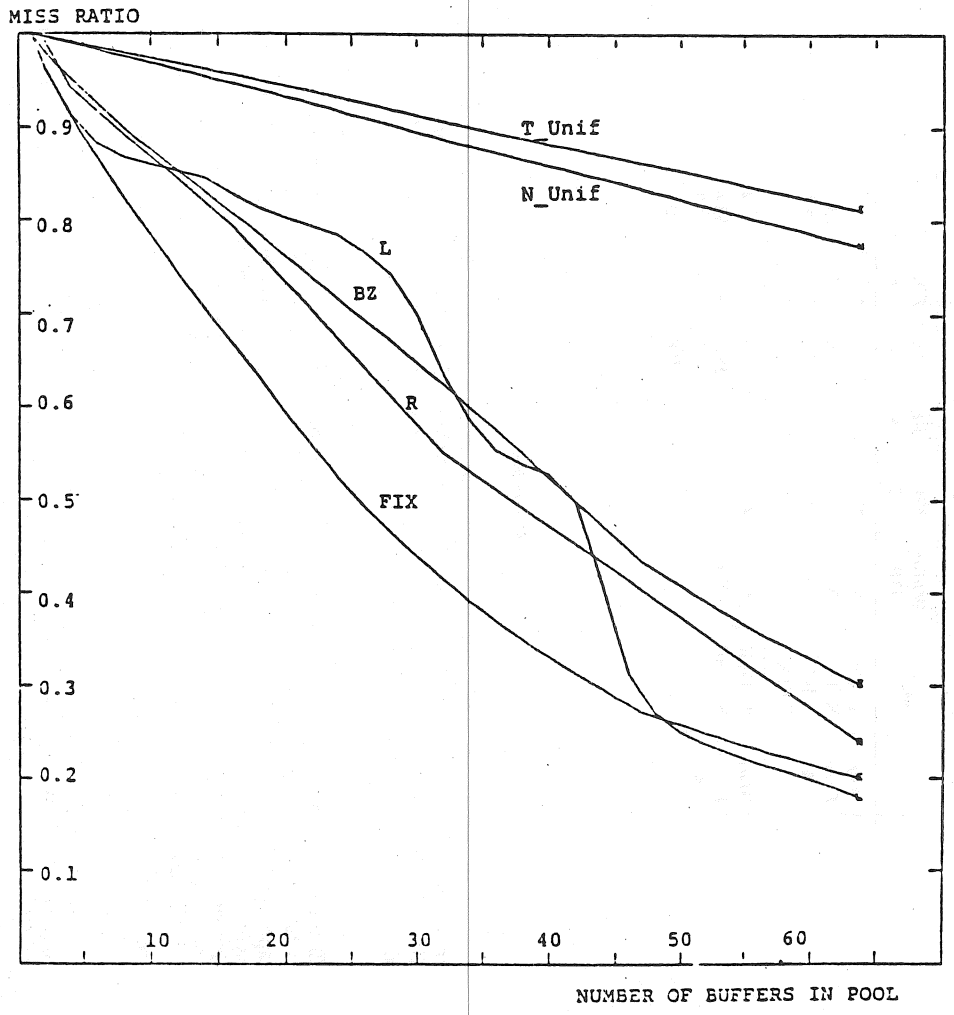


Figure 9

Database B, All Files
Miss Ratio using
Hypothetical Bradford-
Zipf Distribution

distributions against the measured RANDOM miss ratio. It can be noticed from Table 2 that average and maximum error percentages for *BZ* and *FIX* have increased slightly (with respect to Table 1) in some cases and decreased slightly in others (files 4 and 5). The average error is approximately 9% for both estimates, and the maximum error is 27.3% for *BZ* (an increment of 1.2) and 20.7% for *FIX* (an increment of 1.7). Consequently, the Bradford-Zipf hypothesis provides satisfactory predictions for all the observed cases.

Table 2
Miss Ratio Curves Error Analysis
(with hypothetical Bradford-Zipf block reference distributions)

	Model	Whole DB	File 2	File 4	File 5	File 6	Total Cases
Avg. Error %	T_Unif	32.2	26.8	33.9	6.6	31.0	27.3
	N_Unif	30.3	7.4	27.2	6.6	30.5	23.9
	BZ	3.8	5.7	11.2	2.4	24.4	8.9
	FIX	10.8	9.7	0.6	4.9	17.8	9.1
Max. Error %	T_Unif	57.4	48.5	50.0	18.5	37.9	57.4
	N_Unif	53.6	16.7	36.5	18.5	36.9	53.6
	BZ	7.5	12.7	16.3	6.9	27.3	27.3
	FIX	15.0	20.7	2.7	10.8	20.6	20.7

6. Summary

We have studied the reference behaviour of an actual production database workload and provided some evidence that block references follow a Bradford-Zipf-like distribution. We have indicated that this observation can be a useful operational hypothesis for the analysis of database systems, particularly for database performance predictions.

We have proposed a model of buffer management (the *BZ* and *FIX* formulae with B-Z block reference distributions) that estimates the buffer miss ratio of a database workload as a function of the relative frequency of block references, and the buffer pool size. These approximations can be used to determine the I/O activity generated by a particular workload, within the context of a layered database performance prediction model. This represents an intermediate step in the process of predicting database performance in terms of response times and throughputs. The model can be used to assess the effect of buffer sizing, buffer allocation (to internal files), and buffer locking strategies. In the context of selecting an appropriate buffer replacement strategy, our model predictions can be used as an indication of achievable buffering performance.

The buffer miss ratio model was validated against data from a production system, yielding an 8.7% average error and a 26.1% maximum error with the *BZ* estimate, and an 8.9% average error and a 19.0% maximum error with the *FIX* estimate. For all the observed cases, the use of an hypothetical Bradford-Zipf distribution (as opposed to the exact block reference distribution) does not increase significantly the error in the performance estimates.

It is clear that a B-Z distribution of block references conveys only the clustering aspects of locality but not its persistency tendency ([Bunt84]). For models of the type described in section 2, it is not realistic to attempt to capture the persistency aspect of DB references in a production environment, due to the high cost involved in the process. In contrast, the block access frequency distribution generated by an operational system can be easily obtained by direct measurement. For a system in the design stages, relative frequencies of block accesses can be estimated based on the B-Z distribution and the observation of similar operational systems.

The performance model of database buffer management proposed here represents a balance between simplicity and accuracy. It is more accurate than the simpler models based on the assumption that block references are uniformly distributed. To achieve this increased accuracy, the several parameters required to specify a B-Z distribution must be determined for each database file. Still our model requires for less detailed specification than do many models that concentrate on the buffer management problem in isolation (e.g., [Fern78]).

By assuming that the buffer replacement strategy in our proposed model is random, we lose the capability of comparing one buffer replacement strategy against another. This can be viewed as more of a benefit than a fault, however, in that the behaviour of the random replacement algorithm is much less sensitive to the details of specific reference strings than is LRU, for example. (Note how much more erratic the behaviour of LRU is than that of RANDOM in figures 8 and 9.) Thus, for the purpose of predicting the effects of changes in database block sizes and buffer sizes, a model based on random replacement may well yield better estimates of relative performance changes than one based on LRU replacement.

Overall, the database buffer management model proposed here has accuracy commensurate with that of the other components of multi-layer database performance models, yet does not require extensive parameterization or excessive computation.

References

[Benn75] Bennet J.M., Gelenbr E., Potier D.
"Storage Design for Information Retrieval: Scarrott's Conjecture and Zipf's Law", Proc. Int. Computer Symp. 1975, North-Holland Publ. Co., 1975, 233-237.

[Broo69] Brookes B.C.
"The Complete Bradford-Zipf 'Bibliography'", Journal of Documentation #25.1, March 1969, 58-60.

[Bunt84] Bunt R.B., Murphy J.M., Majumdar S.
"A Measure of Program Locality and Its Application", Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, ACM Performance Evaluation Review Special Issue #12.3, August 1984.

[Casa86] Casas I.R.
"Prophet: A Layered Analytical Model for Performance Prediction of Database Systems", Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto, Tech. Report CSRI-180, Toronto, Canada, May 1986.

[Casa87] Casas I.R., Sevcik K.C.
"Structure and Validation of an Analytic Performance Predictor for System 2000 Databases", Proc. 5th Canadian Information Processing Society Conf., Edmonton, Canada, Nov. 1987.

[Chou85] Chou H.T., DeWitt D.J.
"An Evaluation of Buffer Management Strategies for Relational Database Systems", Proc. 11th VLDB Int. Conf., Stokholm, Sweden, August 1985, 127-141.

[Effe84] Effelsberg W., Haerder T.
"Principles of Database Buffer Management", ACM Trans. Database Syst. #9.4, Dec. 1984, 560-595.

[Fern78] Fernandez E., Lang T., Wood C.
"Effect of Replacement Algorithms on a Paged Buffer Database System", IBM J. Research & Development #22.2, March 1978.

[Hawt79] Hawthorn P., Stonebraker M.
"Performance Analysis of a Relational Data Base Management System", Proc. ACM SIGMOD Int. Conf., Boston, 1979.

[Kear83] Kearns J.P., DeFazio S.
"Locality of Reference in Hierarchical Database Systems", IEEE Trans. Software Eng. #SE-9.2, March 1983, 128-134.

[Maga81] Magalhaes G.C.
"Improving the Performance of Data Base Systems", Ph.D. Thesis, Dept. Computer Science, Univ. of Toronto, Tech. Report CSRG-138, Toronto, Canada, Dec. 1981.

[Miya75] Miyamoto I.
"Hierarchical Performance Analysis Models for Data Base Systems", Proc. First VLDB Int. Conf., Boston, 1975, 322-352.

[Naka75] Nakamura F., Yoshida I., Kondo H.
"A Simulation Model for Data Base System Performance Evaluation", Proc. AFIPS Nat. Comp. Conf., 1975, 459-465.

[Peac82] Peachey J.B.
"The Bradford-Zipf Distribution and Program Behaviour", M.Sc. Thesis, Dept. of Computational Science, University of Saskatchewan, Saskatchewan, Canada, Tech. Report 82-1, 1982.

[Sacc82] Sacco G.M., Schkolnick M.
"A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model", Proc. 8th VLDB Int. Conf., Mexico, 1982, 257-262.

[Sams78] Samson W.B., Davis R.H.
"Search Times Using Hash Tables for Records with Non Unique Keys", The Computer Journal #21.3, Aug. 1978, 210-214.

[Sevc81] Sevcik K.C.
"Data Base System Performance Prediction Using an Analytic Model", Proc. 7th VLDB Int. Conf., Cannes, France, Sept. 1981.

[Teor76] Teorey T.J., Das K.
"Application of an Analytical Model to Evaluate Storage Structures", Proc. ACM SIGMOD Int. Conf., 1976, 9-19.

[Yeh77] Yeh R.T., Baker J.W.
"Toward a Design Methodology for DBMS: A Software Engineering Approach", Proc. 3rd. VLDB Int. Conf. , Tokyo, Japan, 1977, 16-27.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

MECANISMOS DE CONTROLE PARA UM SGBDR ATIVO

Décio Fonseca (*)

SUMÁRIO

Este artigo apresenta um Sistema de Gerenciamento de Banco de Dados Relacional e Ativo, através da inclusão no SGBDR de mecanismos de controle. Estes mecanismos fazem parte de um sistema global de gatilhos. Os controles serão realizados em diferentes níveis, tais como: Interface, Autorização e Execução.

ABSTRACT

This work presents a Data Base Management System Actif, with the contr mechanisms. This mechanisms are included in the triggers global system. The controls in the different levels are: Interface, Authorisation and Execution.

(*) Doutor em Informática Universidade Paris VI-INRIA, Professor Visitante do Departamento de Informática da UFPE: áreas de interesse Bancos de Dados, Sistemas de Informação Sistemas Especialistas. AV. Agamenon Magalhães, 1500 apt 201 Torreão - 52030 - Recife-PE

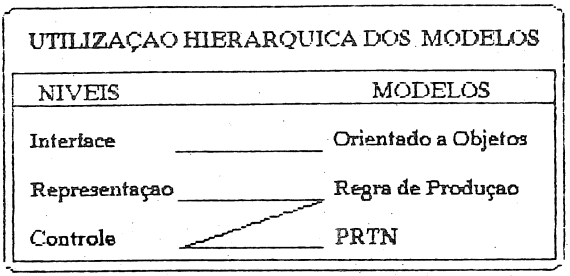
1) INTRODUCÃO

O conceito de gatilhos é muito utilizado na modelagem dos aspectos dinâmicos dos sistemas de informação, ou seja, do seu funcionamento e de sua ativação automática. A dinâmica de um sistema de banco de dados é caracterizada por seus estados e pelas modificações validas. É preciso que as transições dos estados do banco de dados, a partir de um conjunto de regras aplicadas nos objetos , atinjam sempre um estado coerente para exprimir corretamente o comportamento do sistema de informação . Várias foram as publicações que destacaram a importância dos gatilhos nos sistemas de banco de dados (BUNE79, ESWA76,FONS87,STON85....).

A evolução convergente das áreas de bancos de dados e de inteligência artificial autoriza o desenvolvimento de ferramentas tais como os gatilhos, permitindo exprimir e resolver certos problemas clássicos de bancos de dados. A motivação para retomar o estudo sobre os gatilhos tem sua origem na possibilidade de abordar este assunto sob um novo ângulo, na fronteira entre as áreas de banco de dados e de inteligência artificial.

Duas razões principais são favoráveis a um enfoque através de gatilhos. Eles simplificam consideravelmente a descrição das operações de atualização deixando ao SGBD o controle automático dos problemas de integridade, e simplificam também a utilização do SGBD, onde o usuário não precisa conhecer as regras de comportamento do sistema e suas dependencias.

Neste artigo trataremos particularmente dos aspectos relativos aos controles para a modelagem global de um sistema de gatilhos. A proposição de um mecanismo para gatilhos se inspira em três modelos: orientado a objetos, os sistemas de produção e as Redes de Petri a predicado(PRTN) Estes três modelos são utilizados em três níveis .



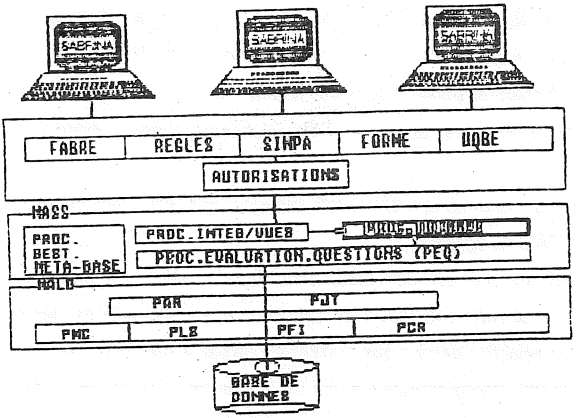
A utilização dos modelos em níveis permite uma melhor adequação dos modelos. A utilização combinada dos objetos com regras de produção oferece a possibilidade de estruturar bem os objetos do sistema, o que permite um controle mais estrito sobre as seqüências das operações. Três razões motivaram a escolha do modelo de PRTN para o controle da execução dos gatilhos: o modelo é bem adaptado à modelagem das regras de produção; êle modela bem as repercussões das atualizações e constitui uma ferramenta formal e gráfica potente, capaz de validar e de verificar a consistência das especificações.

Nosso enfoque visa alcançar os seguintes objetivos: - Oferecer ferramentas que permitam controlar de maneira eficaz a especificação e a execução de um sistema de gatilhos ; Controlar a definição precisa dos objetos e a seqüência da seleções das funções de cada objeto; - Controlar as repercussões das atualizações graças a uma modelagem orientada a objetos , a utilização das meta-regras , a interação amigável com os usuários e ao gráfico das transições(PRTN) ; - Controlar as mensagens enviadas pelos usuários ao sistema e vice-versa;

Na seção 2 mostramos a arquitetura geral do sistema . Na seção 3 apresentamos o sistema de controles , e na seção 4 concluimos com algumas observações sobre o trabalho e propomos novas orientações para futuros trabalhos.

2. ARQUITETURA GERAL

Esta seção define a arquitetura geral onde se insere o monitor de gatilhos. A arquitetura do SGBD SABRINA [GARD85] e composta de três máquinas virtuais: interface,assercional e algébrica lógica. Este trabalho engloba as máquinas de interface (interface orientada a objetos) e assercional (monitor de gatilhos).



O monitor de gatilhos é definido na camada de controle da máquina assercional, com uma alta interação com o monitor de restrições de integridade e com o monitor de autorizações. A complexidade do gerenciamento das restrições de integridade justifica a definição de um monitor independente de gatilhos. Esta solução apresenta como vantagens uma melhor integração dos elementos do sistema e o aumento do desempenho na execução dos gatilhos. O monitor vai se comportar como um motor de inferência de um sistema de regras de produção integrado ao SGBD, selecionando os gatilhos e ativando automáticamente suas ações. Estas ações serão controladas pelo sistema de integridade para garantir a coerência do banco. O funcionamento do monitor é baseado fundamentalmente nos encadeamentos para frente. A independência entre o monitor de gatilhos, o banco de dados e o conjunto de gatilhos torna o sistema evolutivo.

3. SISTEMA DE CONTROLES

O sistema de controle se encarrega do seqüênciamento das operações, e mais especificamente gerencia as informações de controle, testa as pre-condições de ativação e estabelece as post-condições para assegurar a coerência do sistema. Ele também envia mensagens ao mecanismo de ativação para provocar a execução das ações e controlar todas a interações com os usuários inclusive as autorizações de acesso ao sistema.

3.1 MODELO DE CONTROLE

O modelo de representação e controle (Rede de Petri A Predicado) é constituido de:

(1) Um grafo de disparo(TAD,T,AC) onde TAD , T e AC são chamados de tipos abstratos de dados, transição e ação . Os vértices correspondem aos TAD . As transições definem as condições de disparo do gatilho e os arcos podem representar ou uma conexão entre as relações fonte e as transições, ou as ações do gatilho com uma conexão entre as transições e a relação destino.

(2) Relações fonte onde as operações que são nelas aplicadas como também os efeitos colaterais constituem o TAD do sistema.

(3) Vários TAD que contêm um conjunto de regras, onde cada uma destas regras(gatilhos) é constituida de uma parte condição e de uma parte ação. Na parte das ações podem existir várias ações sobre as relações destino que pertencem ou não a um TAD. Se as ações não agem sobre as relações destino que pertencem a um TAD, não teremos repercussões de atualização. Esta situação é um caso favorável de fim de ciclo. Cada TAD é representado uma só vez no grafo de transição.

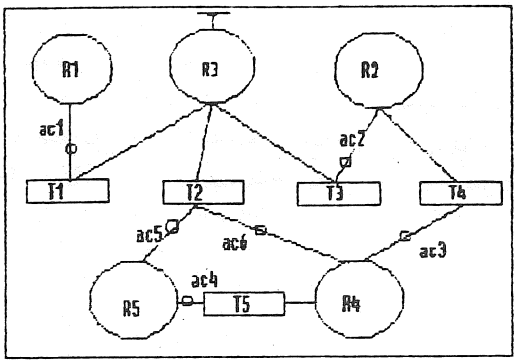
(4) Gatilhos disparáveis. Um gatilho é dito disparável se as seguintes condições são satisfeitas:- as ocorrências das nuplets das relações satisfazem as condições do gatilho; - as condições temporais são verificadas; - as condições agindo nas operações de atualização são satisfeitas.

(5) Gatilhos dos TAD e das transações atômicas. No modelo todas as ações são definidas como unidades atômicas de tratamento. Se um gatilho tem uma ou mais ações a disparar elas devem ser executadas até o fim de sua execução normal. Uma nova execução do gatilho não se realizará antes do final das execuções das ações do gatilho precedente. O efeito de cada atualizaçao sobre o TAD provocará uma execução atômica dos gatilhos selecionados no TAD e em seguida é feita a análise dos efeitos colaterais dos gatilhos.

(6)Verificações e de tratamentos sobre relações temporária, ou seja, o sistema não trabalha diretamente sobre as relações permanentes. Entre as ações, as operações de atualização atuam sobre as relações temporária apenas dentro do espaço de trabalho alocado, enquanto outras como as mensagens ou a ação Solicitar são enviadas diretamente ao usuário. Em relação às condições, os critérios também são aplicados no espaço de trabalho e correspondem ao caso mais geral de nosso enfoque.

(7) Uma seqüência de disparos, respeitando o caráter atômico de cada gatilho e de cada TAD. O encadeamento dos gatilhos se faz a partir das relações destino.

Exemplo geral do modelo: Seja o grafo de transições com as relações , as transições e as ações:



Suponhamos que uma atualização chega ao sistema e que modifica uma tupla da relação R3 que pertence a um TAD. A seqüência do ciclo durante a execução é a seguinte: 1)T1((R3,ac1,R1), (2) T2(R3,ac5,R5), 3)T2(R3,ac6,R4), (4)T2(R3,ac2,R2), (5)T4(R2,ac3,R4), (6)T5(R4,ac4,R5).

Este exemplo permite de destacar os principais aspectos do ciclo de execução por etapas:

1) Na chegada de uma query ativa sobre uma relação pertencente a um TAD inicia o ciclo de execução.

2) Para prevenir os ciclos, a relação fonte do inicio de ciclo e o tipo de atualização são memorizados. Além disso, para cada gatilho que segue as repercussões das atualizações são guardados o tipo de atualização e a relação destino.

3) As transições são construidas por prioridade da esquerda para a direita.

4) A pesquisa das transições se faz da esquerda para a direita.

5) Para cada transição todas as ações são executadas, igualmente na horizontal da esquerda para a direita até a execução da última ação.

6) As relações destino são memorizadas na fila de espera durante a execução das ações, eliminando-se as eventuais relações duplicadas. Assim, após a execução de todos os gatilhos de cada tipo, o ciclo de execução continua tomando como TAD corrente o primeiro TAD seguinte da fila de espera. Este processo se desenvolve iterativamente até que não se tenha mais gatilhos disparáveis.

Este exemplo mostra também que uma relação destino (R2) pode tornar-se uma relação fonte e vice-versa. Ainda neste exemplo todas as relações pertencem a um TAD, exceto R5.

3.2 CONTROLES GERAIS E META-CONTROLES

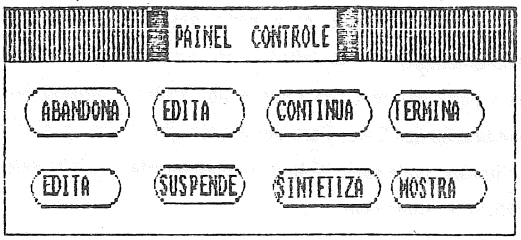
O ambiente de trabalho para propor o sistema global integrado é essencialmente constituido pelos seguintes elementos: os gatilhos, as restrições de integridade ativas e as meta-regras. A manutenção da coerência do banco é feita na inserção de cada um deste elementos. A seleção de um gatilho se faz por um procedimento de unificação baseado na comparação das condições. O controle preventivo de ciclos eventuais provocados pelos gatilhos é também controlado pelo sistema. Todas as vezes que esta situação se verifica o usuário é advertido. Se o usuário deseja a existência eventual de ciclos ele indicará através da interface de definição de gatilhos.

É fundamental num sistema de ativação automática garantir a correção das execuções , uma evolução coerente do banco de dados e a limitação dos efeitos das repercussões das atualizações sob pena de se colocar em perigo o sistema de informação. Para isto definimos vrias estratégias de verificação: 1- uma interface orientada a objetos amigável , capaz de fazer um certo número de controles quando da definição e execução dos gatilhos; 2- definição de ferramentas de controle algoritimica ; 3- utilização de modelos de representação e controle bem adaptado ao problema; 4- definição de um sistema para controlar as autorizações de acesso ao sistema de gatilhos. No entanto estas proposições estão ainda longe de serem suficientes; por isto propomos duas estratégias de simulação para melhor controlar o problema. A primeira corresponde ao controle algoritimico sintático e semântico das repercussões que podem provocar a inclusão no sistema de cada novo gatilho . A segunda é uma execução utilizando-se uma cópia do banco real e observando-se o comportamento do sistema, através de ferramentas gráficas de controle em cima do grafo de transição.

Para se dispor de uma estatística das execuções e das avaliações dos gatilhos, e contribuir para melhorar a administração da evolução do sistema de gatilhos definimos o objeto META_CONTROLE. Esta relação é atualizada em inserção quando o usuário define o gatilho em modificação após avaliação ou execução de cada gatilho, e em supressão após a supressão do gatilho correspondente.

Para se utilizar esta relação e para permitir um controle mais eficaz da evolução dos estados do banco de dados e do sistema de gatilhos foi definido um painel de controle . Este

painel usa interface orientada a objeto para simplificar as interações com o usuário e o administrador do banco de dados.



ABANDONA - abandona a transação ; SUSPENDE - suspende a execução do gatilho para analisar suas eventuais conseqüências ; EDITA - edita o grafo de transição para visualizar o comportamentos dos gatilhos e as propagações das atualizações provocadas por estes; CONTINUA - continua a execução interrompida pela função SUSPENDE; VISUALIZA- edita os gatilhos e para cada gatilho são associadas duas colunas. A primeira indica o estado do gatilho no ciclo da transação correspondente (ativo, ...) e a segunda indica o número de execuções do gatilho no ciclo ; SINTETIZA - permite consultar um conjunto de elementos do sistema antes de tomar certas decisões. A esta função são associadas outras funções de ajuda à investigação; MOSTRA - edita um certo número de informações sobre os gatilhos e sobre o estado da base; TERMINA - deixa o painel de controle.

Existem vários métodos para selecionar os gatilhos a executar: - Indeterminista - A seqüência das transições candidatas ao disparo é escolhida arbitrariamente. A execução termina quando o conjunto de transições candidatas ao disparo engloba a totalidade das transições. - Interativa - a transição candidata ao disparo é função do diálogo interativo aleatoriamente escolhido ou explicitamente designado. - Prioridade - as transições candidatas são escolhidas seguindo sua prioridade. - Estatística - o tipo da regra(gatilho) determina as transições candidatas.- Temporal - pela ordem de satisfação das condições. O mecanismo combina estes métodos, o que garante uma grande eficacia no gerenciamento do processo de controle e de execução, e utiliza as meta-regras para controlar de maneira uniforme o sistema de gatilhos.

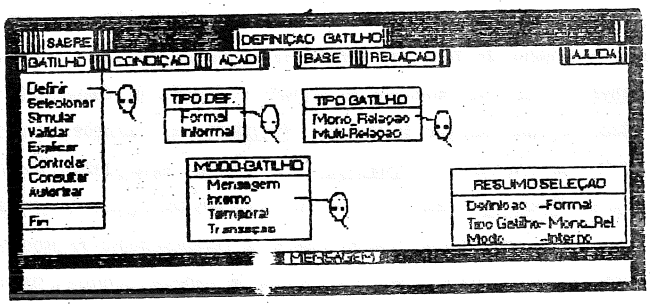
3.3 CONTROLES PELA INTERFACE

Na interface orientada a objetos a seleção dos objetos é efetuada em função da chegada de mensagens do usuário, portando sobre as classes dos objetos. Em cada objeto existe um

conjunto de ações permitindo responder as mensagens que são enviadas. A definição de um objeto gatilho pode ser feita de duas maneiras: a partir de um gatilho já existente incluindo outras propriedades como uma condição, ou a partir dos objetos CONDIÇÃO e AÇÃO e outras informações suplementares. A sua definição provoca a transformação das relações de base em relações ativas a partir das relações fonte e dos atributos referenciados na parte condição do gatilho, e a modificação dos TAD. Para que a execução de um gatilho seja protelada para o final da transação, o usuário pode explicitá-lo através da interface ou não selecionar o tipo de operação (inserção,modificação, supressão).

Todos os objetos à exceção das Relações, dos Atributos e dos Conhecimentos podem ser definidos e manipulados de maneira independente. Para evitar que o usuário defina CONDIÇÕES ou AÇÕES que eventualmente não serão jamais usadas, incluimos um controle a cada avaliação. Se após um certo tempo as condições e as ações não são jamais avaliadas, se estabelece um diálogo com o usuário para se decidir sobre a manutenção ou não destas ações.

Cada seleção de um objeto com o "Mouse" corresponde a uma mensagem para o sistema de interface. Contudo há certas mensagens que exigem um controle mais importante em função das repercussões que elas provocam, como por exemplo a escolha da função "Definir Gatilho". Para controlar os encadeamentos das seleções usamos as ferramentas disponíveis no sistema XWINDOWS e um mecanismo do tipo mapa de bits que controla as modificações de estado das funções do sistema.



Um objeto importante é chamado "CONHECIMENTO". Ele descreve o nível meta do sistema. Ele permite a definição das meta-regras e a descrição de todo tipo de conhecimento sobre a aplicação, sobre os gatilhos e sobre o sistema em geral. Ele permite além disso controlar a execução dos gatilhos através de um quadro de controle que é ativado a partir da função "meta-

controle". Este objeto corresponde a um menu fantasma e pode ser solicitado a todo momento a partir do "mouse" apoiado na janela principal

3.4 CONTROLE DAS AUTORIZAÇÕES

O controle das autorizações para os gatilhos é muito importante. A especificação de um sistema confiavel e ao mesmo tempo flexivel é apresentado nas seguintes etapas:

ETAPA 1 : ATRIBUIÇÃO DE AUTORIZAÇÃO

Na realidade a unica autorização para se manipular as relações e os atributos deveria ser dada a aquele que as definisse . A autorização de se definir gatilhos poderá ser dada aos usuários que tenham pelo menos um bom conhecimento semântico da aplicação .

É preciso também definir o tipo de autorização de manipulação do usuário. O usuário pode ter uma autorização de modificar um atributo, mais eventualmente ele não poderá realizar operações do tipo agregado sobre este atributo. Neste caso incluimos uma autorização associada ao tipo de manipulação . Ainda nesta etapa é previsto um controle suplementar: quando a um usuário é atribuido uma autorização de modificação, o sistema verifica se todos os gatilhos podendo ser disparados pelas atualizações resultantes são possiveis de ser ativados pelo usuário. Se o usuário tem a autorização de modificar ou de suprimir uma relação do tipo principal, o sistema cria automáticamente a autorização para manipular as relações dependentes (dependência referencial).

Ainda na etapa 1 o administrador pode atribuir uma autorização de ativação, se o usuário tiver a autorização de modificação da relação fonte. Dispondo das autorizações necessárias o usuário pode passar a etapa 2. A autorização de se definir um gatilho evita que certos usuários que têm a autorização de manipular as relações fonte ou destino definam um gatilho sem terem o conhecimento necessário para esta definição e aumenta também a proteção de bases de gatilhos mal definidas.

ETAPA 2 : DEFINIÇÃO DE GATILHO

Após o controle sobre as relações e os atributos fontes, existe o controle sobre os atributos e as relações destino. Resumindo, para se definir um gatilho é necessário se ter as

autorizações para modificar as relações fonte e destino, além da autorização para se definir o gatilho. Um vez definidos, os gatilhos são objetos do sistema onde são definidas as autorizações (atualização, consulta, ativação)

ETAPA 3 : AUTORIZAÇÃO DE ATIVAÇÃO E ATUALIZAÇÃO DOS GATILHOS

Nesta etapa é feita uma simulação do gatilho definido para garantir que no momento de sua execução o ciclo da transação não sera abandonado. É tambem atribuida a autorização de ativação para os gatilhos dependentes. Esta autorização é dada quando o usuário não tem a autorização de modificar a relação destino, mas tem a autorização de modificar a relação fonte. Esta autorização dá apenas o direito de executar o gatilho. No entanto o usuário não pode modificar diretamente o atributo da relação fonte correspondente. A execução dos gatilhos lhe é transparente.

ETAPA 4 : EXECUÇAO DOS GATILHOS

Quando o gatilho vai ser executado ele pode provocar o abandono da transação . Se o usuário não tem as autorizações necessárias este abandono é natural. No entanto se ele tem uma autorização de execução concedida na etapa 2(dois), antes da execução do gatilho uma atualização temporaria das autorizações é efetuada para possibilitar as modificações da relações por outros gatilhos, evitando assim o abandono indevido de transações. Para se ter a autorização de execução é preciso se ter pelo menos a autorização de modificação sobre a relação fonte.

A autorização de ativação tem como vantagens:

- Uma transparencia de execução na medida em que certas ações podem ser executadas sem o conhecimento do usuário.
- Uma proteção de atributos sem rigidez no controle: a manipulação destes atributos sendo feita por gatilhos interpostos de maneira indireta.
- Uma redução nos controles das restrições , pois certos gatilhos podem compensar as incoerências.
- Uma maior garantia da coerência como conseqüência das duas vantagens precedentes.

5) CONCLUSÃO

Este trabalho apresenta uma solução eficaz para controlar a especificação e a execução de gatilhos, através de ferramentas historicamente ligadas a inteligencia artificial como as regras de produção e de outras ferramentas (algoritimicas, interface orientada a objetos) . O sistema apresentado é ativo, possuindo capacidades reativas a partir dos gatilhos e da restrições de integridade ativas.

As orientações futuras deste trabalho são:

- explicitar as razões das incoerências;
- oferecer maiores possibilidades de dedução de informações e de detectar eventuais incoerências;
- definir ferramentas capazes de evitar incompatibilidades ou redundancias entre gatilhos;
- oferecer mais possibilidades gráficas que permitirão um controle mais simples e interativo com os usuários.

A modelagem de um sistema de gatilhos tende a evoluir em direção a soluções paralelas, beneficiando-se das máquinas com arquiteturas paralelas. Este novo ambiente paralelo modificará o conceito de gatilho temporal. Neste caso, teremos uma modelagem e uma avaliação de vários gatilhos ao mesmo tempo, sendo também incorporada a noção de estado .

BIBLIOGRAFIA

[BUNE79] BUNEMAN O.P, CLEMONS E.K.:"Efficiently Monitoring Relational Databases". ACM TODS Vol4, N°3, Sept. 1979.

[ESWA76] ESWARAN K. P.:Specifications,Implementations and Interactions of a Trigger Subsystem in an integrated Database System".IBM Research RJ1820,Aug1976.

[FONS87] FONSECA D. "Un Mecanisme d'Activation et de Controle de Declencheurs Orienté Objets". Thèse de l'Université Paris VI,dec 1987.

[GARD85] GARDARIN G. et al:SABRINA:Une Evolution du SGBD SABRE vers un Systeme Intelligent et Actif. Actes PRC BD3, St Pierre de Chartreuse, Mars 1985.

[MELK83] MELKANOFF M., CHEN Q. : "Integrating Action Capabilities into Information Databases". 2nd Int. Conf. on Databases, Cambridge, Sept 1983.

[STON85] STONEBRAKER M.:" Triggers and Inferences in Database Systems", College of Engineering University of California, Berkeley, 1985.

[TUCH85] TUCHERMAN L., FURTADO A., CASANOVA M. : "A Tool for Modular Database Design". VLDB Conf. Stockholm 1985.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Artesano : Una arquitectura de administración de bases de datos basada en vistas con enfoque funcional

c.c. Paulina Frenkel,
Lic. Nicolás Casariego,
Lic. Jorge Zarauza

Servicio de Procesamiento de Datos SECYT - CONICET
Rivadavia 1906 2do piso (1033) Buenos Aires
TE: 48-7017 , 48-2173 TX: 18052 CICYT AR

RESUMEN

Se presenta un desarrollo de una arquitectura de administración de bases de datos basada en vistas con un enfoque funcional a nivel definición, manipulación y recuperación.

Por un lado este abordaje, que distingue explícitamente el nivel externo (visión de los usuarios) del nivel conceptual (modelo lógico), hace posible su implementación interactuando a nivel físico tanto con un DBMS relacional standard o directamente sobre un file manager más un desarrollo ad-hoc.

Por otro, la modalidad elegida apunta a :

- rescatar del modelo de relación universal RU la estructura de nombres de atributos como único conocimiento del usuario respecto del modelo conceptual.

- utilizar el grafo de dependencias funcionales respecto de las claves y las vinculaciones explícitas del modelo Entidad-Relación en los algoritmos de navegación de la base de datos.

- aprovechar la semántica de objetos tratada para lograr la actualización de vistas libre de efectos colaterales.

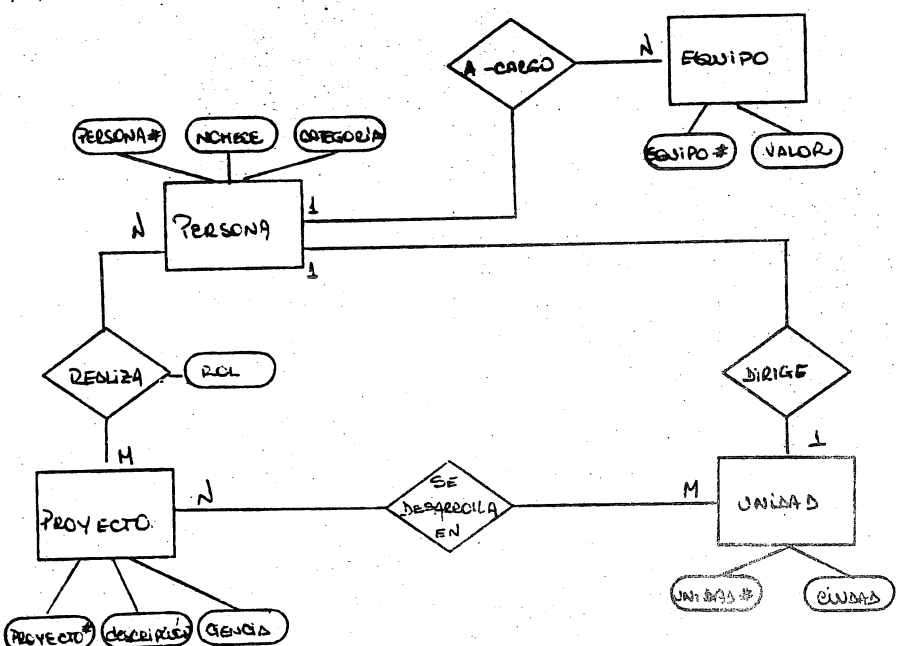
- proveer una interfase más natural de consulta.

EQUIVALENCIA DE MODELOS

Un modelo Entidad-Relación (ER) [CHE76] puede representarse mediante un conjunto de tablas en forma equivalente a un modelo relacional.
En una simplificación, para cada conjunto de entidades (CE) se puede definir una tabla (relación entidad [DAT87]), con sus atributos descriptivos, y con los atributos de identificación (identificador) como clave primaria.
Los CE débiles se representan por una tabla cuya clave primaria esta formada por la clave del CE que la sustenta más un componente discriminatorio (relación característica [DAT87]).
Las vinculaciones de 1:1 y de 1:N entre dos CE se representan incluyendo el identificador de uno, como atributo descriptivo del otro (clave foranea) y las vinculaciones de N:M como relaciones asociativas [DAT87] con la unión de los identificadores de los CE participantes, como clave primaria.
En el modelo resultante, denominado base de datos ER [BRA85], se verifican dos condiciones:

- Integridad de Entidad : ningún componente de una clave puede tener un valor nulo [DAT83].
- Integridad Referencial : una clave foranea o bien es nula o bien se corresponde con la clave primaria de una tupla de la relación foranea [DAT83].

Asumimos que los aspectos del mundo real que se modelan en la base de datos, han sido representados en un diagrama ER bien formado, previo a su traducción en un conjunto de tablas.
Tomemos como ejemplo el siguiente diagrama ER con vinculaciones binarias que representa la interacción entre personas que desarrollan proyectos en unidades de investigación. Estas personas son responsables de determinado equipamiento científico y pueden dirigir estas unidades y/o participar en los proyectos.



La base de datos ER queda compuesta por las tablas:

```
tablas
PERSONA          (persona#, nombre, categoria)
EQUIPO           (equipo#, valor, responsable#)
PROYECTO         (proyecto#, descripción, ciencia)
UNIDAD           (unidad#, ciudad, director#)
REALIZA          (realizador#, actividad#, rol)
SE-DESARROLLA-EN (desarrollo#, lugar#)
```

donde cada atributo representa el uso de un dominio dentro de cada tabla.
En nuestro caso, veamos qué atributos están asociados a un mismo dominio:

```
dominios
PERSONA-ID  (persona#, director#, realizador#, responsable#)
PROYECTO-ID (proyecto#, actividad#, desarrollo#)
UNIDAD-ID   (unidad#, lugar#)
EQUIPO-ID   (equipo#)
IMPORTE     (valor)
CODIGO      (rol, ciencia, grado, categoria)
TEXTO       (nombre, descripción, ciudad)
```

A partir del diagrama ER pueden explicitarse las dependencias funcionales entre atributos. La integridad de entidad garantiza la existencia de una dependencia funcional (DF) de los atributos respecto del identificador de un CE o lo que es equivalente de la clave de la tabla a la que pertenecen.

```
persona# -> nombre                equipo# -> valor
         -> categoria                     -> responsable#

proyecto# -> descripción          unidad#  -> ciudad
          -> ciencia                       -> director#

realizador#,actividad# -> rol
```

Tanto persona#, como responsable#, director# y realizador# son nombres distintos de atributos con valores pertenecientes a un mismo dominio: persona-id; al igual que lugar# y unidad# con respecto a unidad-id. Estos sinónimos los usamos cuando incluimos los atributos de una tabla en otra (claves foraneas restringidas por la integridad referencial), en un modelo de datos con exigencia de nombres únicos.
El uso de estos sinónimos, más las propiedades de la DF (reflexividad, aumentación y transitividad) nos definen otras dependencias funcionales tales como:

```
persona# -> persona#              director# -> persona#
         -> realizador#                     -> realizador#
         -> director#                       -> responsable#
         -> responsable#                    -> director#
                                            -> nombre
```

-> categoría

desarrollo#,lugar# -> desarrollo#
-> lugar#

En nuestro enfoque la DF se representa como una función denominada primitiva, en la que el atributo queda determinado genéricamente por el dominio al que pertenece la clave primaria de la cual depende [KEN81]; por ej.: Si el nombre depende funcionalmente de persona#, decimos que:

nombre(persona-id)

es la representación funcional de dicha dependencia, donde el cuerpo de la función es el atributo en cuestión y el parámetro de la función es el dominio al cual pertenece la clave primaria. Entonces a partir de un valor del dominio persona-id queda determinada una instancia del atributo nombre.
Esta función primitiva es la misma para:

persona# -> nombre
director# -> nombre
realizador# -> nombre
responsable# -> nombre

Las siguientes son funciones primitivas del ejemplo dado:

persona#(persona-id)
nombre(persona-id)
categoría(persona-id)

proyecto#(proyecto-id)
descripción(proyecto-id)
ciencia(proyecto-id)

realizador#(persona-id,proyecto-id)
actividad#(persona-id,proyecto-id)
rol(persona-id,proyecto-id)

desarrollo#(proyecto-id,unidad-id)
lugar#(proyecto-id,unidad-id)

equipo#(equipo-id)
valor(equipo-id)
responsable#(equipo-id)

unidad#(unidad-id)
ciudad(unidad-id)
director#(unidad-id)

En un esquema con nombres de atributos únicos, el conjunto de DF implementado como funciones primitivas, contiene la información suficiente para que baste la mención de esos nombres para identificarlos.

Esto esta fundamentado por la teoría de la Relación Universal RU [ULL83] [KEN81] [M&U82] [M&S88] donde un esquema queda definido por un conjunto finito de objetos, más un conjunto de dependencias funcionales respecto de la clave.
Un objeto es un conjunto de atributos, y se asume que todos los objetos son un subconjunto de un conjunto universal de atributos. Podemos asimilar el concepto de objeto del esquema RU al de una relación que representa un CE o vinculaciones entre éstos, donde cada atributo tiene asociado un dominio de igual forma que en el modelo relacional.

Artesano define estos objetos a través de un mecanismo de vistas

(tablas "virtuales" derivadas de proyecciones, juntas y/o uniones de las tablas básicas [DAT83]) con el conocimiento de las dependencias entre atributos expresadas en las funciones primitivas.
En el modelo relacional la existencia de vistas otorga mayor independencia de datos [DAT83] [COD79] al pertenecer a un nivel por encima del conceptual, logrando inmunidad ante los cambios tales como crecimiento y reestructuración del esquema.
Por otro lado, las vistas permiten al usuario tener una percepción más simplificada del universo modelado, al tiempo que preparan una vía para implementar la seguridad en cuanto a qué datos pueden ser accedidos por qué usuarios.
Artesano está concebido para interactuar con un esquema lógico preexistente haciendo posible su aplicabilidad tanto sobre un administrador de bases de datos standard, como sobre un "file manager" más un desarrollo ad-hoc.
En esta arquitectura basada en vistas, la correspondencia entre las actualizaciones sobre estas con actualizaciones sobre las tablas resultan complejas debido a que las vistas que tienen atributos asociados a un mismo dominio provenientes de tablas distintas, ocasionan actualizaciones interdependientes.
Según la complejidad de la vista puede ocurrir que una modificación de un atributo desencadene una sucesión de inserciones, modificaciones y/o eliminaciones en las diferentes tablas involucradas.
En efecto, se espera que la actualización de vistas actúe como si fuera sobre los objetos básicos de la base de datos, esto es, libre de efectos colaterales.
Para que esto ocurra es necesario restringir el tipo de vistas que se permiten definir. Hay varios enfoques estudiados por distintos autores [M&SBB], que van desde el más estricto, que consiste en definir vistas coincidentes con tablas, hasta el más flexible, por medio del cual se pueden definir vistas que garanticen el "mapping" de actualizaciones (inserción y eliminación), pasando por aquellas vistas que sólo garantizan el "mapping" de eliminaciones.
Artesano, en su primer versión, permite definir y manipular vistas que involucran atributos de varias tablas, con la restricción que cada atributo sea accesible a través de un camino de longitud uno dentro del grafo que representa la clausura [SAL86] respecto a la clave de la vista.
De esta manera es posible actualizar las vistas y respetar la integridad de la base de datos.

ENFOQUE FUNCIONAL

Una vista se crea, definiéndola mediante una expresión funcional que contiene un conjunto de atributos (como funciones primitivas), literales y/o vistas preexistentes. Los parámetros de la función determinan la clave de la vista, resultando de la unión de todos los parámetros de las funciones primitivas y/o vistas incluidas.

```
⟨definición⟩   ::= ⟨declaración⟩ = ⟨término⟩ [+⟨término⟩ ...]
⟨declaración⟩  ::= nombre-vista( dominio [,dominio ...] )
⟨término⟩      ::= literal /
                   atributo( ⟨argumento⟩ [,⟨argumento⟩ ...] ) /
                   vista( ⟨argumento⟩ [,⟨argumento⟩ ...] )
```

```
⟨argumento⟩    ::= dominio /
                   ⟨término⟩
```

Una vista se recupera mediante la invocación de dicha función usando como argumento efectivo un valor del dominio que tiene asociado la clave de la vista.

```
⟨invocación⟩   ::= nombre-vista( ⟨argumento⟩ [,⟨argumento⟩ ...] )
⟨argumento⟩    ::= valor  /
                   ↕
```

Siguiendo con nuestro ejemplo, explicitaremos la semántica del enfoque funcional mediante sentencias en SQL standard [ORA86] que producen un efecto equivalente, tanto en la definición como en la recuperación de las vistas.

Dadas las siguientes tablas:

PERSONA

persona#	nombre	categoria
100	paulina	1
200	nicolás	3
300	jorge	3

EQUIPO

equipo#	valor	responsable#
55	1000	300
56	500	200

UNIDAD

unidad#	ciudad	director
2111	buenos aires	300

PROYECTO

proyecto#	descripción	ciencia
AA	base de datos	MC
BB	lenguajes	MC

REALIZA

realizador#	actividad#	rol
100	AA	02
100	BB	01
200	BB	02

SE-DESARROLLA-EN

desarrollo#	lugar#
BB	CNEA
AA	CAICYT

En la definici^["^\n funcional vemos que no es necesario incluir dentro de la vista, los atributos que forman parte de la clave; ni mencionar la tabla de donde provienen todos los atributos.
En esta version de Artesano la recuperaci^["^\n o bien es de todo el conjunto o bien es de una instancia.

Definición:

vista1(persona-id) = nombre(persona-id) + categoria(persona-id)

CREATE VIEW vista1 AS

```
SELECT nombre, categoría, persona#
FROM   persona ;

Recuperación:

vista1(200) = nicolás , 3
SELECT nombre, categoría FROM vista1 WHERE persona# = 200 ;

vista1(100) = paulina , 1
SELECT nombre, categoría FROM vista1 WHERE persona# = 100 ;

vista1(*)   = paulina , 1
              nicolás , 3
              jorge   , 3
SELECT nombre, categoría FROM vista1 ;

-----------------------------------------------------------------

vista2(equipo-id)   = nombre(responsable#(equipo-id)) +
                      categoría(responsable#(equipo-id))

CREATE VIEW vista2 AS
SELECT nombre, categoría, equipo#
FROM   persona, equipo
WHERE  persona# = responsable# ;

vista2(55)  = jorge , 3
SELECT nombre, categoría FROM vista2 WHERE equipo# = 55 ;

-----------------------------------------------------------------
```

Dentro de la definici^["^\n de una vista se puede incluir una vista ya existente.

```
-----------------------------------------------------------------

vista2'(equipo-id)  = vista1(responsable#(equipo-id))

CREATE VIEW vista2' AS
SELECT nombre, categoría, equipo#
FROM   vista1, equipo
WHERE  persona# = responsable# ;

vista2'(55) = jorge , 3
SELECT nombre, categoría FROM vista2  WHERE equipo# = 55 ;

-----------------------------------------------------------------
```

Los atributos que intervienen en vistas con clave multi-dominio pueden depender de cualquiera de los dominios componentes o cualquier combinaci^["^\n de ellos.

```
-----------------------------------------------------------------

vista3(persona-id,proyecto-id) = rol(persona-id,proyecto-id) +
                                 nombre(persona-id) +
                                 descripción(proyecto-id)
```

```
CREATE VIEW vista3 AS
SELECT rol, nombre, descripción, persona#, proyecto#
FROM   realiza, persona, proyecto
WHERE  persona#  = realizador#
AND    proyecto# = actividad# ;


vista3(100,AA) = 02 , paulina , base de datos
SELECT rol, nombre, descripción FROM vista3
WHERE persona# = 100 AND proyecto# = 'AA' ;

vista3(100,*)  = 02 , paulina , base de datos
                 01 , paulina , lenguajes
SELECT rol, nombre, descripción FROM vista3 WHERE persona# = 100 ;

vista3(*,BB)   = 01 , paulina , lenguajes
                 02 , nicolás , lenguajes
SELECT rol, nombre, descripción FROM vista3 WHERE proyecto# = 'BB';
```

Los literales se utilizan en forma equivalente a una constante en el cuerpo de una funci^["^\n genérica, pudiendo ser usados como componentes de una vista o para instanciar una funci^["^\n primitiva.

```
vista4(unidad-id)   = nombre(director#(unidad-id)) +
                      'es el director de la unidad' +
                      unidad#(unidad-id)

CREATE VIEW vista4 (nombre,texto,unidad#) AS
SELECT nombre, 'es el director de la unidad', unidad#
FROM   persona, unidad
WHERE  persona# = director# ;

vista4(2111) = jorge es el director de la unidad 2111
SELECT nombre, texto, unidad# FROM vista4 WHERE unidad# = 2111 ;
```

CONCLUSIONES

Hacia un Lenguaje de Consulta más Natural

Una de las formas más simples de implementar una interfase de consulta y recuperación basada en un lenguaje "cuasi-natural" con un vocabulario controlado consiste en el reconocimiento de palabras clave en el contexto adecuado, descartando la "cáscara" sintáctica.
En nuestro enfoque, la disponibilidad de una estructura de nombres de atributos con un alto contenido semántico, preparan el camino para la construcción de interfases más amigables y simples

con el usuario.
La interpretación de un requerimiento expresado en un subconjunto del castellano, es uno de los próximos objetivos a encarar en la implementación de Artesano, luego, la definición de la vista surgiría de la interpretación del requerimiento.

Ejemplo:

- Requerimiento:

" quiero la descripción y la ciencia del proyecto 'AA', más el nombre del director de las unidades donde se desarrolla ".

- Interpretación como vista:

```
vista(proyecto-id,unidad-id) =
    descripción(proyecto-id) +
    ciencia(proyecto-id) +
    nombre(director#(lugar#(proyecto-id,unidad-id)))
```

- Recuperación:

```
vista(AA,?)
```

IMPLEMENTACION

Este trabajo está parcialmente implementado con la Base de Datos de Recursos Científico-Tecnológicos del Servicio de Procesamiento de Datos SECYT -CONICET.
Paralelamente, alumnos de la ESLAI en una pasantía dirigida por los autores " Diseño de un módulo de manejo de visiones y acceso a datos ", están construyendo, con un enfoque equivalente, un prototipo para interactuar con un file-manager.

ANEXO

Propiedades de la Dependencia Funcional [SAL86]

```
NOMENCLATURA : atributos                (minúsculas)
               conjunto de atributos    (mayúsculas)
               ->                       (determina funcionalmente)
               U                        (unión de conjuntos)

REFLEXIVIDAD o "Dependencia Trivial" :
     (x,y) -> x
     (x,y) -> y

AUMENTACION :
     X -> Y  implica  X U Z -> Y U Z

TRANSITIVIDAD :
     si  X -> Y  y  Y -> Z  entonces  X -> Z

"Atributos Extra en Parte Izquierda" :
     X -> Y  implica  X U Z -> Y

"Misma Parte Izquierda" :
     X -> Y  y  X -> Z  implica  X -> Y U Z

"Unico Atributo en Parte Derecha" :
     Z = { z1, z2, ... zn }
     X -> Z  si y solo si     X -> z1
                              X -> z2
                              ...
                              X -> zn

CLAUSURA de X :
     X+ = { y / X -> y }
```

REFERENCIAS BIBLIOGRAFIAS

[CHE76] P.P. CHEN
" The entity-relationship model - Towards a unified view of data "
ACM, Transactions on Databases System (March 1976).

[COD79] E.F. CODD
" Extending the database relational model to capture more meaning "
ACM Transactions on Database Systems Vol.4 N.4 (1979)

[KEN81] W. KENT
" Consequences of assuming a universal relation "
ACM Transactions on Database Systems Vol.6 N.4 (1981)

[M&U82] R. FAGIN & A.O. MENDELZON & J.D. ULLMAN
" A Simplified universal relation and its properties "
ACM Transactions on Database Systems Vol.7 N.3 (1982)

[ULL83] J.D. ULLMAN
" Universal relation interfases for databases systems "
Information Processing, IFIP (1983) North Holland.

[DAT83] C.J. DATE
" Introduction to Database Systems "
Addison-Wesley (1983)

[BRA85] L.I. BRADY
" A Universal relation assumption based on entities and relationships "
IEEE (1985).

[SAL86] B.J. SALZBERG
" An introduction to data base design "
Academic Press (1986).

[ORA86] J. SACHS
" Oracle: SQL*Plus user's guide " version 2.0
Oracle Corporation, California USA (1986)

[DAT87] C.J. DATE
" Relational Database "
Addison-Wesley Cap.19 (1987)

[M&S88] V.M. MARKOWITZ & A. SHOSHANI
" Abbreviated query interpretation in entity relationship oriented databases "
Computed Science Research Dept. Berkeley (1988).

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

LOS SGBD EXTENSIBLES

H.STEFFEN*, A. GUTIERREZ, R. RUGGIA

* Laboratoire M.A.S.I. - Univ. Paris VI -
4 place Jussieu, 75252 Paris Cedex 05.
&
PE.DE.CI.BA. Casilla de Correos 1207
Montevideo - Uruguay

IN.CO. Facultad de Ingeniería
J. Herrera y Reissig 565
Montevideo - Uruguay

Resumen : *La multiplicación de nuevas necesidades y aplicaciones, sobre todo en las áreas de CAD, geografía, asistencia a la ingeniería, etc. ha puesto en evidencia las limitaciones de los SGBD basados en el modelo relacional para implantar ese tipo de aplicaciones en forma adecuada. Como respuesta a esta inadecuación una nueva clase de SGBD está apareciendo, proponiendo extensiones al modelo relacional o permitiendo la definición de nuevos modelos de datos. Esos manejadores, denominados SGBD Extensibles, permiten una representación más rica y adecuada de los datos, así como mecanismos de implantación adaptados a cada especificidad. Varios centros de investigación trabajan sobre proyectos de SGBD Extensibles y algunos prototipos están disponibles. Esta nueva área de trabajo es un punto de confluencia entre disciplinas tradicionales de las Bases de Datos (Ingeniería de sistemas) y otras como Lenguajes (Orientación Objeto y Funcionales), Inteligencia Artificial (Base de Datos Deductivas) y Concepción de Sistemas de Información (propuestas de nuevos modelos de datos).*

I - INTRODUCCION

El área Base de Datos participa actualmente a un conjunto de trabajos sobre una nueva generación de SGBD. Esta actividad está en gran parte orientada a dar respuesta a los numerosos problemas planteados por nuevas aplicaciones de tipo no tradicional, como la CAD, la gestión de documentos textuales, la concepción de VLSI, etc.

Este tipo de aplicaciones no tradicionales poseen características suficientemente diferentes de las aplicaciones clásicas de gestión como para motivar el estudio ya sea de extensiones al modelo relacional, ya sea de SGBDs de nuevo tipo.

Estas extensiones están motivadas por el hecho de que los SGBD relacionales no son capaces de resolver en forma eficiente y adecuada a la aplicación los nuevos problemas que se presentan.

El modelo relacional en sí tiene como gran ventaja su simplicidad, lo cual facilita el diseño y la manipulación. Pero en esta simplicidad tiene como desventaja el hecho de que, para la resolución de problemas cuyos elementos tienen una estructura que se aparta de las del modelo se muestra pobre en herramientas de modelización y manipulación.

Por ejemplo, supongamos un caso bastante simple y clásico para resolver en modelo relacional: Se tiene una empresa que desea llevar el registro de los empleados con datos como nombre, dirección y los nombres de sus hijos.
En modelo relacional, esto se resolvería por medio de dos esquemas de relación:

EMPLEADOS(Nombre_empleado,Dirección)
HIJOS(Nombre_empleado,Nombre_hijo)

cuando una representación más natural sería:

EMPLEADOS(Nombre_empleado,Dirección, Conjunto_hijos)

donde Conjunto_hijos es un atributo de tipo conjunto; y sus elementos son accedidos por medio de operadores sobre conjuntos.

Este problema, si bien es resuelto en forma más natural utilizando "extensiones" al modelo relacional, puede ser resuelto utilizando las herramientas "puras" del mismo. Sin embargo existen casos en que los objetos a modelizar poseen una estructura tal que se hace poco clara su representacion con modelo relacional, y difícil su manipulación por medio de los operadores relacionales.

Por ejemplo consideremos un caso en el cual deben manipularse objetos geométricos (rectas, círculos, polígonos) en el plano.
Trabajando en modelo relacional se tendría:

RECTAS(id_r,info_r)
CIRCULO(id_c,info_c)
POLIGONO(id_p,info_p)

donde cada id sería un conjunto de atributos atómicos que caracterizan el objeto.

Y se desean realizar consultas como:

- Conjunto de rectas de coeficiente angular determinado.
- Conjunto de polígonos con igual área.
- Conjunto de polígonos que se intersectan.
- etc.

Para realizar este tipo de consultas, trabajando en un modelo relacional puro, se deberán realizar programas en algún lenguaje anfitrión, ya que las condiciones que se piden no son expresables en lenguajes de consulta relacionales. En cambio si existieran definidos los tipos Recta, Círculo, Polígono, con un conjunto de operaciones asociadas (área, coeficiente angular, pertenencia de un punto a una recta o a un círculo o polígono, etc) las consultas anteriores podrían ser resueltas directamente por el lenguaje de consulta.

Por ejemplo, podría definirse:

```
CREATE TABLE RECTAS
    id_r   : Recta;
    info_r : String;
END.
```

(los esquemas CIRCULOS Y POLIGONOS se definen de forma análoga).

y expresar las consultas de la forma:

```
SELECT info_r
FROM RECTAS
WHERE coef_ang(id_r) = 30.
```

```
SELECT info_r
 FROM RECTAS 1, RECTAS 2
 WHERE coef_ang(1.id_r) =
          coef_ang(2.id_r).
```

I. 1 Limitaciones del modelo relacional.

En general, pueden considerarse un conjunto de aspectos que no son resueltos por los SGBD relacionales actuales. Dentro de este conjunto se encuentran: manipulación de grandes objetos, de versiones, definición de nuevos tipos de datos y operadores por parte del usuario, operadores más poderosos sobre conjuntos, transacciones largas y anidadas.

¿ Qué problemas se plantean con estos puntos ?

Si consideramos el aspecto de *manipulacion de grandes objetos*, vemos que existen ciertas aplicaciones que necesitan manipular objetos de gran tamaño, como documentos, imagenes, voz, programas [BERN87]; pero los SGBD relacionales permiten el almacenamiento (y manipulación) de objetos de tamaño chico (artículos, tuplas).

Actualmente los SGBD relacionales resuelven este problema descomponiendo los grandes objetos en partes más chicas. Como consecuencia de esto, su manipulación exige un trabajo de reconstrucción que significa una pérdida importante de performance, y dificulta la expresión de las consultas.

Por otro lado los SGBD relacionales ofrecen una cantidad muy limitada de alternativas de almacenamiento y métodos de acceso, lo cual impide la optimización del espacio de almacenamiento y del tiempo utilizado en manipulación mediante el uso de estructuras y métodos de acceso más adaptados a las características de los objetos.

Considerando el aspecto de *mantenimiento de versiones* de un mismo objeto, se puede observar que esta facilidad es fundamental para ciertas aplicaciones como CAD, gestión de documentos, de programas,etc. [ADIB86] [DAYA85] [KATZ86]. Sin embargo las características del manejo de versiones cambia según la aplicación y parece interesante el ofrecer mecanismos parametrables.

Otro tema clave es el del *manejo de tipos*. En general, casi todas las aplicaciones - tradicionales o no - necesitan manejar objetos de tipos no atómicos (conjuntos, tipos estructurados, tipos recursivos, grafos), o incluso necesitan que sea el propio usuario quien se defina sus propios tipos. En ciertos casos puede ser interesante contar con la posibilidad de representar datos por medio de las consultas (SQL por ejemplo) que permiten calcularlos [STON84a] [ZANI83].
Por el contrario, los tipos de datos en los SGBD relacionales se limitan en general a artículos (no recursivos), formados por campos de tipos atómicos.

Si consideramos el tema de las *formas de acceso*, observamos que una posibilidad que tienen los SGBD actuales es la de acceder a los datos basandose en su contenido (acceso asociativo). Esta facilidad es muy útil en gran cantidad de aplicaciones; y en los últimos años se ha trabajado mucho en el desarrollo de nuevas estructuras y nuevos métodos más eficientes para el acceso asociativo. [FAG79, LARS80, LITW80, LOME83, LITW87, LOME87, BENT75, BENT79, NIEV84, ROBI81, SCHE82]. Sin embargo, pocos de estos mecanismos se encuentran incorporados en SGBD comerciales.[BERN87]

Otra posibilidad es la de poder acceder a los datos, no según su valor sino por su identidad. Por ejemplo, sean dos empleados que tienen un hijo con igual nombre. ¿ Se trata del mismo individuo o de individuos distintos pero iguales en su representación en la base de datos ? Manejar el concepto de identidad entre objetos permite distinguir estos casos.

Una caracteristica de los operadores relacionales es su calidad de operadores sobre conjuntos de tuplas. Podría ser interesante también tener operadores que actúen sobre conjuntos de otros tipos de objetos (por ejemplo que actuen sobre grafos). Es más, ser ía deseable que fuera el usuario quien agregara sus propios operadores a su medida. Por ejemplo, por medio de la definición de nuevas funciones agregadas del tipo PROMEDIO, SUMA, MEDIA, SEGUNDO_MAYOR, etc. [STON88]

Estas limitaciones de los SGBD Relacionales frente a loas actuales y sobre todo a las nuevas aplicaciones a impulsado el desarrollo de numerosas propuestas de mejora y de extensión de las funcionalidades de los SGBD, dando nacimiento así a los SGBD Extensibles.

II - CARACTERISTICAS DE LOS SGBD EXTENSIBLES

La denominación SGBD Extensible es utilizada para designar sistemas con objetivos y características diferentes, abarcando un espectro muy amplio.
En la primer parte de este capítulo estudiaremos los principales aspectos sobre los que puede aplicarse la extensibilidad de los sistemas, y en la segunda parte veremos los distintos enfoques arquitectónicos que pueden ser propuestos.

II.1 - EXTENSIBILIDAD

La extensibilidad de los sistemas puede ser vista como dando respuesta en los siguientes aspectos :

1) Enriquecer la representación externa y la manipulación de los datos de los usuarios.
2) Permitir la definición de métodos específicos de almacenamiento y de acceso a los datos.
3) Ofrecer nuevas funcionalidades para facilitar la implantación de nuevos operadores .

II.1.1 - Enriquecer la representación

Actualmente el diseño de una aplicación se realiza en (por lo menos) dos niveles diferentes : el nivel abstracto (utilizando la especificación y modelos semánticos) y un nivel conceptual del esquema de la Base de Datos (utilizando el modelo relacional enriquecido con restricciones de integridad y dependencias funcionales).

Tal como hemos visto, una primer dificultad de este enfoque es que el modelo relacional es pobre para representar directamente objetos complejos. Una alternativa es dar más poder de expresión a los modelos de datos a través de tipos primitivos más potentes.

Otro inconveniente es que la representación estructural de los datos está separada de su manipulación. Mientras que la estructura de los datos es conocida por el SGBD a partir del esquema de la BD, los programas de aplicación son externos al mismo.

Una solución interesante es de autorizar la definición de Tipos Abstractos de Datos (ADT) a nivel mismo del SGBD. De esta forma la representación de los datos se aproxima o se confunde incluso con la modelización semántica.

La definición de ADT permite la descripción de la estructura de los datos a un nivel arbitrario de complejidad, así como del conjunto de rutinas de manipulación de esos datos (rutinas que en el enfoque orientado objeto llamamos métodos).
Los ADT soportan la noción de encapsulación, de forma que el único acceso autorizado a los datos se realiza a partir de las rutinas definidas en el ADT. En este caso la descripción estructural y comportamental de los datos es hecha de forma integral y al mismo tiempo.

Relacionada con los tipos de datos, se encuentra la idea de **integridad**. Referente a la misma el SGBD debe chequear que los objetos se ajusten a sus definiciones de tipo, o cumplan ciertas propiedades.

Por otro lado, los objetos de la base de datos pueden tener restricciones de integridad que son más complejas que las que pueden ser expresadas en el lenguaje de definición de tipos. Por ejemplo, chequear que un documento sea consistente con el estándar de una organización. [BERN87].

Otra noción importante para enriquecer la representación y la manipulación de los datos es la de **identidad de los objetos**.
Los sistemas relacionales están basados sobre el valor de los objetos. Los tratamientos son de tipo de conjunto sobre el contenido de los objetos (tuplas). Este tipo de orientación no es suficiente para modelizar situaciones más complejas y poder realizar tratamientos adecuados.
Así es introducida la noción de identidad de los objetos, lo que nos permite :

- crear la noción de **referencia** sobre un objeto, o identificador de objeto (OID).
  Un OID es un valor externo al objeto que lo desgna en forma única e invariable.
- poder comparar dos referencias a objetos y saber si se trata del mismo o de diferentes objetos.
- poder construir objetos estructurados a partir de otros objetos, utilizando los identifiacdores (OID) de éstos en lugar de copiar su contenido. Esto nos evita las copias redundantes y las puestas al día múltiples.
- Realizar asociaciones explícitas entre objetos.

Con la noción de identidad aparece un nuevo operador: IDENTICO(x,y) que toma un valor verdadero si x e y referencian al mismo objeto.

Implementar los ADT en un SGBD

Ofrecer la definición de ADT presenta un conjunto de problemas a resolver por los SGBD extensibles :

- integrar un lenguaje de definición de ADT interactivo
- permitir la programación de los métodos asociados
- estudiar mecanismos de almacenaje adecuados
- implementar las nociones de identidad y de igualdad de objetos

Una primera solución de conjunto consiste en dividir la gestión de los ADT en dos partes : una a cargo del SGBD y otra a cargo del usuario. EL usuario define nuevos tipos, caracterizándolos únicamente por su nombre y su almacenaje físico (largo en octetos, fijo o variable). El SGBD memoriza esta definición y se encarga del almacenaje y del acceso a los objetos de dicho tipo.
En cuanto a la definición de métodos, el usuario indica al SGBD el nombre y los tipos de los parámetros formales, así como el archivo donde el programa (fuente o código) de dichos métodos han sido almacenados.
El código del SGBD no es modificado ya que los nuevos métodos son externos.

La invocación a un método identificable hace que el SGBD lo cargue a partir del archivo y los ejecute sobre los objetos designados.
En este enfoque el control de la estructura interna del nuevo tipo está enteramente a cargo del usuario.

Esta solución es la más simple y puede implementarse a partir de un SGBD clásico, al que se le agregan las funcionalidades descriptas.

Una segunda solución es de proponer un enfoque integrado, en el cual los nuevos tipos estén completamente bajo control del SGBD. En este caso los nuevos métodos pasan a formar parte del SGBD. Esta última solución permite una gestión uniforme de las funciones - sistema y usuario-, un mejor control de coherencia así como permite tener en cuenta el costo de ejecución de las funciones para poder realizar la optimización de operaciones y consultas.

La implementación de esta solución presenta dos grandes problemas :
1) estudiar métodos de almacenaje capaces de soportar objetos complejos
2) estudiar un mecanismo de integración del código de los métodos.

1) almacenar objetos complejos :
Si vemos el almacenaje en el primer enfoque (no integrado), la solución es simple, puesto que los objetos son vistos por el SGBD como una simple cadena de caracteres (cuyo largo es limitado por parámetros del sistema). La estructura interna de los objetos es conocida únicamente por el usuario.
En este segundo enfoque, el SGBD debe soportar el almacenaje de estructuras complejas de largo variable y que pueden alcanzar dimensiones muy importantes. Una simple gestión de los objetos por contigüidad no es siempre posible ni deseable (costo de almacenaje, de modificación).
El SGBD debe conocer la estructura interna de esos objetos y ser capaz de asegurar la correspondencia entre representación externa (contigua) --> implantación física (paginada).

2) integrar nuevos métodos
- fabricar un nuevo código del SGBD a partir del viejo y de los métodos agregados. Esto se hace por edición de lazos estática.
Este mecanismo permite una integración total de las rutinas pero es pesado y costoso, poco adaptado a un trabajo interactivo con el usuario.

- mantener separados el código del SGBD y el de los métodos, utilizando una librería de métodos y una edición de lazos dinámica. Este mecanismo es más costozao en tiempo de ejecución y exige un editor de lazos dinámico integrado al SGBD.

Las soluciones propuestas van a optar por el primer caso cuando la definición de ADTs sea realizada en función de adaptar un SGBD a una aplicación dada, y luego no se vuelve a modificar (más allá del mantenimiento del sofware). Por el contrario, si la definición de nuevos ADTs es frecuente, la segunda solución parece imprescindible.

II.1.2 - Métodos de acceso y de almacenaje

La posiblidad de definir nuevos mecanismos de acceso y de almacenaje de datos tiene como objetivo adaptar estos mecanismos a las características específicas de las aplicaciones. En efecto, los métodos de acceso (índices, hash, secuencial) y de almacenaje (relaciones en archivos Vsam, hash, secuenciales) propuestos en forma estándar por los SGBD pueden no ser una buena solución para ciertas aplicaciones. Por ejemplo el acceso a objetos espaciales - figuras geométricas etc.- es optimizado con una representación basada en R-Tree [Gutt 84] y KDB-Tree [Robi 81], o de tipo Grid-files [Niev 84].
De la misma forma, la definición de ADTs puede apoyarse en tipos de base existentes (el almacenaje es hecho por el SGBD) o puede ser completamente redefinida por el usuario. Un ejemplo de esto es el ADT Matriz. Una forma de representarla es a partir de una tabla de tablas - tabla a una dimensión sería un tipo existente en el SGBD -.
Esta definición es rápida y simple, pero ineficaz, puesto que un objeto ´matriz´ puede estar físicamente disperso entre las diferentes tablas que la componen.

Una solución a este problema es de definir un ADT a partir de macanismo de bajo nivel, sin utilizar lista o tabla, permitiéndonos reagrupar físicamente cada objeto ´matriz´ .

Implementar nuevos métodos de acceso

Para autorizar la implantación de nuevos métodos de acceso los SGBD Extensibles deben ofrecer funcionalidades de gestión de la memoria central y secundaria del tipo acceso-a-pagina. El usuario utiliza estas primitivas de bajo nivel y contruye su propia estructura de datos. Estos mecanismos de bajo nivel deben incluir funcionalidades de servicio como control de concurrencia, recuperación, buffer.

II.1.3 - Nuevas funcionalidades

La diversidad de nuevas aplicaciones exige de los SGBD Extensibles la posiblidad de implantar nuevos operadores y de emplear posibilidades del sistema .
Estas funcionalidades pueden ser estudiadas en las siguientes áreas :

1 - orientadas a aplicaciones de tipo SGBD Deductivo.
2 - gestión de versiones históricas.
3 - gestión de transacciones largas.
4 - varios tipos de gestión de la concurrencia.
5 - estrategias de buffer adaptables.

Orientadas a la deducción

Un SGBD Deductivo necesita mecanismos adaptados al almacenaje y acceso rápido a las reglas [Gard87], así como estructuras de datos que autoricen el empleo de algoritmos de cálculo deductivo.

El almacenaje de reglas necesita estructuras de datos adaptadas, del tipo listas, conjuntos, tablas dinámicas, tuplas. Sólo éstas últimas forman parte de los SGBD tradicionales.
De la misma forma es importante disponer de índices adaptados a la búsqueda de reglas cuando su cantidad comienza a ser importante .

La ejecución de una regla necesita determinar la definición de todos sus predicados, que pueden a su vez depender de otros predicados derivados. Encontrar recursivamente todos los predicados a partir de una regla se resuelve de forma fácil y eficaz por un cierre transitivo de un grafo, donde cada nodo representa un predicado - derivado o de base -.
De forma análoga, el cálculo de reglas lineales recursivas puede ser realizado eficazmente por un operador de punto fijo basado en un cierre transitivo de un grafo [Puch87][Gard 87b].
De esta forma un constructor ´grafo´se presenta como un útil de gran interés en un contexto deductivo.

Versiones

Como veíamos en el capítulo I, ciertas aplicaciones necesitan mantener versiones históricas de sus objetos. Esta gestión debe permitir una utilización simple y rápida de un objeto de una versión dada. Los dos principales parámetros a tener en cuenta son 1) copiar la cantidad mínima posible del objeto tal que obtengamos dos versiones completas del mismo, y 2) estudiar mecanismos de reconstrucción rápida de una versión a partir de todas sus partes.

Transacciones

El manejo de transacciones es un tema importante en el desarrollo de nuevas aplicaciones. En los SGBD relacionales de hoy, los mecanismos de manejo de transacciones están encarados hacia el manejo de transacciones de duración corta mientras que en nuevas aplicaciones (CAD/CAM, VLSI) es común la existencia de transacciones que duran varios días. En esta situación, no sería aceptable el hecho de que se perdiera todo el trabajo por una falla del sistema. Por esto surge la necesidad de contar con transacciones de larga duración que permitan evitar este tipo de problemas [Born 87].

Otra forma de transacciones que podría desearse manejar son transacciones anidadas. Es decir que dentro de una transacción se lanza otra, y la posible falla de la última no implica la falla de la primera.

Concurrencia y recuperación

Los SGBD clásicos proponen mecanismos de concurrencia basados en un bloqueo en dos faces - con gránulo fijo o variable -. Ciertas aplicaciones - ingeniería, texto - requieren un mecanismo de concurrencia que tenga en cuenta las versiones de los objetos [Katz 83]. Otras como las aplicaciones sobre bases de imágenes - acceso únicamente en lectura - pueden prescindir de un control de concurrencia y de la recuperación.
Otra característica de la concurrencia es su carácter inmediato o diferido. El mecanismo diferido consiste en bloquear únicamente los objetos derivados del resultado obtenido, y no todos los objetos de forma sistemática a medida que se accede a ellos - ésto sólo es posible si las consultas son ejecutadas en forma atómica -.
Si las aplicaciones clásicas utilizan un mecanismo inmediato, la implantación de un mecanismo diferido de control de la concurrencia permitiría reducir su costo de forma significativa .
Implementar este mecanismo en un SGBD implica que el manejador sea capaz de encontrar todas las tuplas que permitieron contruir el resultado, para así bloquearlas al final del tratamiento.

Estrategias de buffer adaptadas

Para obtener una buena utilización de la disponibilidad en memoria central y reducir el costo en E/S, los SGBD realizan una gestión de los datos en ante-memoria -buffer - [Chou 85].
Si tradicionalmente esta gestión es uniforme para todos los tipos de datos, parece interesante incorporar nuevas técnicas de manejo de buffer dedicadas a cada tipo de datos, debido a que las operaciones asociadas a los mismos pueden utilizar distintas formas de acceso.

II.2 - PRINCIPALES ENFOQUES DE ARQUITECTURA

Recientemente varios prototipos de sistemas extensibles vienen siendo presentados. Sus objetivos y características varían en un espectro muy amplio. En particular podemos distinguir tres grandes enfoques : los ´sistemas completos´, los ´sistemas para armar´, y los ´sistemas orientados objeto´.

Sistemas Completos

Los Sistemas Completos están dirigidos directamente a los usuarios finales. Son productos sofware completos y monolíticos, no modificables, que ofrecen numerosos mecanismos de extensibilidad.
Estos SGBDs están basados en un modelo de datos bien definido y fijo (relacional extendido, modelos funcionales , etc.) . El usuario debe adaptar su visión de los datos al modelo propuesto por el SGBD.

La extensibilidad de los sistemas completos está sobre todo orientada a autorizar la definición de nuevos ADTs por los usuarios. En algunos casos la extensibilidad está también dirigida a mejorar las performances del sistema, permitiendo la definición de nuevos métodos de almacenaje y/o de acceso a los datos.

Sistemas para armar

El enfoque de los ´sistemas para armar´ consiste en proponer un esqueleto de base de un SGBD, compuesto de módulos y de librerías de funciones de base. Estos sistemas permiten el prototipaje y la fabricación rápida de SGBD específicos, bien adaptados a las aplicaciones que deben utilizarlos.

Los sistemas para armar no están dirigidos a los usuarios finales sino a un especialista (DBI -Data Base Implementor-) que debe diseñar un SGBD a partir del conocimiento de la aplicación a la cual sera destinado.
A diferencia del caso anterior, estos sistemas no poseen un modelo de datos fijo al que los usuarios deban adaptarse.
Estos sistemas deben proponer un conjunto de módulos y herramientas para facilitar el trabajo de prototipage del nuevo SGBD. Dentro de esos módulos, un núcleo de base de gestión del almacenaje de objetos parece como indispensable. A éste se le pueden agregar otros como librerías (índices B-tree, operadores clásicos, etc). Otro elemento interesante son los generadores: programas que permiten obtener un módulo del SGBD a partir de ciertos parámetros. El caso más claro es el generador del módulo de optimización de consultas.
Las herramientas de ayuda al prototipage pueden ser lenguajes de alto nivel, útiles gráficos, etc.

Sistemas Orientados Objeto

Los SGBD orientados objeto presentan un intento reciente de asociar a un SGBD las características de un lenguaje orientado objeto (Smalltalk, C++, Objectif C)[Banc 88].
Estos sistemas son naturalmente extensibles a nivel de nuevos tipos usuarios, definidos en el lenguaje orientado objeto. Un SGBD OO puede comportarse como un sistema completo (basado en un modelo de datos orientado a objeto) o como un sistema para armar, permitiendo el prototipage rápido de nuevos SBBD específicos.

III - ALGUNOS EJEMPLOS DE SGBD EXTENSIBLES

Para completar el estudio sobre la extensibilidad presentaremos seis prototipos de SGBD extensibles. Estos sistemas son a nuestro entender los casos más importantes y originales. La diversidad en sus enfoques y en las soluciones propuestas es ilustrativa del estado actual del área.

III.1 POSTGRES.

POSTGRES [STON86] [STON88], es un SGBD sucesor de INGRES que esta siendo desarrollado en la Univesidad de California, Berkeley.
Es un sistema completo basado en el modelo relacional.
En este proyecto la extensibilidad esta dada en cinco áreas : tipos de datos, funciones, operadores, agregados y metodos de acceso.

Tipos de datos.

POSTGRES además de proveer tipos primitivos como enteros y cadena de caracteres, permite definir nuevos tipos de datos de base, tablas de tipos base, tipos procedimiento y tipos procedimiento parametrizado por parte del usuario. Dichas extensiones son detalladas en [ROWE87].Una caracteristica a destacar de esta facilidad es que la representación del ADT es dejada al usuario.
Para la definición de nuevos tipos de datos de base se especifica nombre del tipo,largo en octetos (puede ser variable), el procedimiento que hace la correspondencia entre la representación externa (ASCII) y la representacion interna, y el procedimiento que hace la correspondencia en forma inversa.Por ejemplo, si se desea definir el ADT BOX :

```
define type BOX is
    ( Internal lenght = 16,          /* Largo en octetos.
      Input proc = CHARTOBOX,        /* Procedimientos que hacen la correspondencia de
      Output proc = BOXTOCHAR,       /* la representación externa a la interna y viceversa.
      Filename = 'BOXCODE',          /* Nombre del archivo donde se encuentra el fuente
                                     /* de los procedimientos anteriores.
      Default = '')                  /* Valor por defecto.
```

Con el ADT BOX definido anteriormente el usuario puede entonces crear relaciones cuyos atributos sean de dicho ADT. Por ejemplo, si el usuario desea crear la tabla que relaciona los 'BOXs' con su identificación :

```
create IDENTBOX ( ID = integer, DESCRIPCION = box )
```

Funciones.

Permite extender el lenguaje de consulta por medio de nuevas funciones definidas por el usuario.

La forma de definir funciones por el usuario consiste en especificar el nombre de la misma y los parametros, el tipo del resultado, el lenguaje fuente en que está escrito el procedimiento que implementa la función, y el nombre del archivo en que se encuentra dicha función.Por ejemplo, si el usuario desea definir una función que calcule el área de una BOX :

```
define procedure BOXAREA ( BOX )
 return int4 is                          /* int4 es el tipo del resultado.
 ( Language = 'C',                       /* lenguaje en que esta escrito el procedimiento.
   Filename = 'BOXCODE')                 /* nombre del archivo donde se encuentra el
                                            procedimiento.
```

La idea en cuanto a la implementación de esta facilidad de extensión es que POSTGRES acepte el fuente , lo compile almacenandolo en una biblioteca del sistema y que a la hora de la ejecución la función sea cargada dinámicamente, de tal forma que luego quede en memoria. Además POSTGRES hace el chequeo automático de tipos de los parámetros de la función.

Operadores definidos por el usuario.

Los operadores se diferencian de las funciones en que tienen un número limitado de parametros y en que POSTGRES intenta optimizar la ejecución de las consultas que incluyen operadores, utilizando la información definida en los mismos.

Para definir un operador de debe dar el nombre, el procedimiento asociado, el nivel de precedencia (utilizable en caso de aparecer en expresiones sin paréntesis), propiedades del operador (asociatividad, conmutatividad, negación) e información para el optimizador de consultas.

En forma similar es posible definir nuevos operadores agregados.

Métodos de acceso.

No se trata de dar una forma sencilla de extensibilidad ya que no encaran la extensibilidad en este punto hacia usuarios finales, sino hacia programadores con experiencia.

Los métodos de acceso posibles a extender son solo indices secundarios, es decir conjunto de claves con puntero a artículos.
Cuando se extienden los métodos de acceso, se puede usar el manejador de buffer del POSTGRES. Si es asi, se debe llamar a las rutinas del POSTGRES para leer y escribir páginas en el pool de buffers y para agregar y retirar páginas del buffer.

El implementador debe manejar el bloqueo para lograr mecanismos de concurrencia; pudiendo usar un mecanismo de POSTGRES o uno propio. También puede usar funciones de recuperación del POSTGRES.

Estado actual del proyecto y perspectivas.

El prototipo cuenta actualmente con un conjunto de metodos de acceso y permite definir tipos de datos y operadores por parte del usuario. El parser y el optimizador de consultas se encuentran operativos. Los operadores deben ser precargados, por no encontrarse pronto el loader dinámico. Las funciones y los agredados definidos por el usuario todavia no se encuentran operativos.

III.2 STARBURST

El proyecto Starburst [McPh 87] es desarrollado en el IBM Almaden Research Center. Starburst es un sitema completo basado en el modelo relacional extendido.
Sus principales objetivos son :

- extensibilidad de tipos usuarios
- performance
- portabilidad
- datos y funciones distribuidas

La arquitectura del SGBD está organizada en dos sub-sistemas llamdos Core y Corona. Core permite el acceso tupla a tupla de las relaciones. Este subsitema se encarga de la gestión del almacenaje, de los índices, de los mecanismos de buffering, así como de la gestión de transacciones, de realizar controles de integridad y del mecanimos de triggers.

Corona asegura la interface con los usuarios. Este subsistema se ocupa del analizador de consultas, de descomponer y optimizar las consultas así como de generar un plan de acceso. También mantiene un catálogo y controla las autorizaciones.
Core y Corona contituyen una arquitectura similar a la del System R [Astr76], compuesto de los dos subsitemas RSS y RDS.

Extensibilidad

La extensibilidad en Starburst se sitúa a dos niveles : 1) definición de nuevos tipos por los usuarios bajo forma de ADTs. y 2) definir nuevos mecanimos de almacenaje y de acceso a las relaciones.

Definición de nuevos tipos

Starburst se propone autorizar la definición de nuevos tipos de datos para representar objetos complejos. Su enfoque es similar a XSQL [Hask 83] [Lori 83], donde un objeto complejo es una colección de componentes heterogéneos, represntados por tuplas. El almacenaje de esas tuplas es independiente (clustering posible) y su estructuración como objeto complejo se realiza por composición utilizando predicados y claves externas.
Este enfoque permite mantener un almacenaje en relaciones normalizadas, así como la declaración de vistas o de reorganizaciones de los objetos complejos.
Para manipular los datos se proponen realizar extensiones a SQL, así como operaciones para el recorrido de la estructura de los objetos complejos.

Nuevos mecanismos de almacenaje

En este plano, Starburst propone un conjunto de mecanismos integrados, denominado **attachment**. Estos incluyen los métodos de acceso, las restricciones de integridad y los triggers. La idea de base consiste en asociar a cada relación un conjunto de estructuras de datos o de acciones a realizar automáticamente cuando la relación es puesta al día. Por ejemplo, a un atributo de una relación podemos asociarle un índice, un índice de join, una cierta restricción de integridad, etc.

La extensibilidad prevee la posibilidad de agragar nuevos mecanismos de attachment, integrándolos a una estructura de datos sistema donde el conjunto de esas descripciones es mantenido.

Estado actual y perspectivas

El sistema está en curso de desarrollo. Han estudiado algunas modificaciones a SQL de forma de soportar las extensiones propuestas. Estos mecanismos están siendo implementados basándose en la idea de un acoplamiento fuerte de las operaciones y las facilidades de servicio ofrecidad por el sistema. También trabajan sobre la defínción de una nueva interface usuario priorizando la facilidad de utilización y de diálogo.

III.3 PROBE.

PROBE [DAYA86] [GOLD87] es un SGBD extensible de tipo completo basado en un modelo funcional, que está siendo desarrollado por la CCA (Computer Corporation of America).

El objetivo de este proyecto es desarrollar un SGBD avanzado que pueda manejar mejor los tipos de información y el conocimiento intensional de las nuevas áreas de aplicación. Para lograr esto propone aumentar los SGBD existentes con:

- Clases de objetos, como base para definir nuevos objetos y operadores.
- Conceptos dimensionales (espacio,tiempo).
- Predicados y consultas recursivas.

Al diseñar PROBE, se estableció la división entre el núcleo del SGBD (PROBE) y las clases de objetos. PROBE provee operaciones a partir del modelos de datos; por ejemplo, selección y proyección que tratan sobre concuntos de objetos de tipo genérico. Mientras que las operaciones específicas sobre objetos específicos son implementadas con las clases de objetos definidas por los usuarios.
Para incorporar nuevas clases de objetos, se hace uso de un mecanismo llamado **adapter**. Los adapter son rutinas utilizadas para compatibilizar los dos puntos de vista con que una función en PROBE es manejada, que son : desde PROBE y desde la clase del objeto. [GOLD87] Entre las tareas que conciernen al adapter se mencionan : conversión de lista de parámetros, conversión entre tuplas/relaciones y los datos de los lenguajes de programación (y viseversa), manipulación de valores nulos, ofrecer acceso a funciones agregadas.

Con respecto al problema de la **información dimensional**, se propone proveer un soporte directo del SGBD para dicha información, con el objetivo de implementar aplicaciones espaciales con una representación adecuada, y realizar operaciones eficiententemente sobre grupos u objetos espaciales individuales. PROBE propone como solución clases de objetos generales; por ejemplo la clase PTSET (conjunto de puntos), y que la implementación de los objetos sea determinada según la aplicación.

PROBE propone la utilización de la **recursividad** en la definición de vistas y de consultas, para aumentar el poder de expresión del lenguaje de manipulación. Para esto generaliza la clausura transitiva al punto de poder resolver muchas aplicaciones importantes sin sacrificar oportunidades para el procedimiento efectivo sobre grandes bases de datos. Para estos casos especiales se explota y adapta muchos algoritmos (bien conocidos) sobre grafos.

Modelo de datos de PROBE (PDM).

Los constructores de modelación del PDM son las entidades y funciones. Una entidad es usada para representar un objeto del mundo real, y puede ser miembro de uno o más tipos, estando

los tipos organizados en una jerarquía generalizada. Las funciones son usadas para representar propiedades de las entidades. Las entidades y funciones son manipuladas por el **algebra PDM**, que se diferencia del álgebra relacional en que permite la manipulación de entidades de tipos arbitrarios, soporta funciones computadas y consultas espaciales y recursivas.

Perspectivas.

Han identificado puntos de investigación concernientes a SGBD de proposito general como ser:

- Proceso de conocimiento operacional. Este tipo de conocimiento es habitualmente expresado en la forma situación-acción. (por ejemplo triggers).
- Diseño de aplicaciones. Como se distribuye el conocimiento de una aplicación en un SGBD.
- Interfaces.
- Manejo de transacciones.

III.4 EXODUS.

El proyecto EXODUS [CARE87] es desarrollado en la Universidad de Wisconsin, Madison. Es un ´sistema para armar´ donde el objetivo es permitir el desarrollo rápido de un SGBD específico a la aplicación y con alta performance.

La idea en cuanto al diseño de EXODUS es la de proveer por un lado un conjunto de facilidades de núcleo (kernel) para uso en todas las aplicaciones y por otro un conjunto de herramientas para ayudar al DBI a generar un nuevo software de sistema de base de datos específico a la aplicación a resolver.

Entre las componentes fijas que constituyen el núcleo de EXODUS se encuentran: el manejador de objetos almacenables, y el manejador de tipos.

Manejador de objetos almacenables.

El manejador de objetos almacenables es el encargado de manipular todo lo concerniente a los objetos almacenados en disco, para lo cual provee:

- Objetos para almacenar datos y archivos para agrupar logicamente y fisicamente objetos almacenables.
Los objetos almacenados como una cadena de octetos no tipados, pueden ser tan grandes como se quiera. Un objeto se identifica por un OID, que es su dirección virtual. El archivo es un conjunto no ordenado de objetos relacionados, y es útil para proveer un mecanismo de recorrido secuencial y permitir que los objetos puedan ser agrupados físicamente en el disco.

- Manejador de ´buffers´, que maneja parte de objetos de largo variable.

- Soporte para el manejo de versiones.

- Primitivas para manejo de concurrencia y servicio de recuperación.
Para resolver el problema de concurrencia utiliza un mecanismo de bloqueo en dos fases con opción de bloquear el objeto entero o parte de él. En cuanto a recuperación se utiliza un mecanismo de archivo diario (log) de imagenes de antes de la modificación.

Manejador de tipos.

El manejador de tipos es un depósito de información relacionada con los tipos persistentes. Ejemplo: existen dependencias entre los módulos que constituyen definiciones de tipos y módulos

que usan dichas definiciones, dependencias entre archivos del usuario y consultas compiladas que acceden a ellos.Para manejar estas dependencias el manejador de tipos construye un grafo de dependencias, por medio del cual mantiene actualizada dicha información.

Entre las herramientas provistas por EXODUS se encuentran el lenguaje de programación E, una biblioteca de métodos de acceso independientes del tipo y los métodos de los operadores de los modelos de datos, generadores de componentes a partir de una especificación.

Lenguaje de programación E.

El lenguaje E es una extensión del lenguaje C++. Fue diseñado para ser utilizado como herramienta para programar los métodos de acceso, operadores del modelo de datos y funciones utilitarias, permitiendo así al DBI lograr SGBD específicos a una aplicación.

El trabajar en E elimina o resuelve parcialmente los siguientes problemas:

- El DBI debe manejar el pasaje de memoria a disco debido a que todos los datos de interés están en disco, y que lo directamente almacenado no es tipado. Por otro lado, cuando se escriben los métodos de acceso se deben resolver los problemas de concurrencia y recuperación.

- Gran parte de la información de tipos es desconocida por el DBI en el momento de implementación del SGBD. Por ejemplo, con respecto a los métodos de acceso, el DBI al escribir código para manejar índices, debe escribir el mismo de tal forma que pueda ser utilizado aplicado con claves de tipos diferentes. Este problema no se plantea sólo con los métodos de acceso. En el momento de escribir el código que implemente las operaciones a nivel de modelo de datos (selección, proyección, etc, del modelo relacional), el DBI puede asumir muy poco de los tipos de datos con los que tratará.

Para resolver dichos problemas el E provee las herramientas de generadores de clases y persistencia de clases en disco.

Clases, dbclases y generadores de clases.

Uno de los objetivos que EXODUS pretende lograr es la posibilidad de agregar nuevos ADT. El mecanismo que propone para hacer posible ésto, es hacer uso del concepto de **class** del C++ (en el sentido de clase en el enfoque objeto).

EXODUS además provee, a través del lènguaje E, una facilidad que es la de **dbclass**. Una **dbclass** es una extensión del mecanismo básico de clases diseñado para almacenamiento en memoria externa.

E divide el mundo de las clases en dos tipos: aquellos objetos que pueden existir en memoria volátil y aquellos cuyos objetos pueden existir en memoria o en disco. Las primeras son simplemente las clases del C++, y las otras son las dbclass. Las dbclass, sintácticamente, se declaran en forma similar con la restricción de que todos los datos que forman parte de la dbclass deben ser a su vez objetos de una dbclass.

A su vez el lenguaje E provee el mecanismo de **generador de clases**. Este mecanismo permite la definición de clases parametrizables en función de uno o más tipos.
Por ejemplo, para definir un nodo de un B-tree, escribimos la siguiente declaración:

dbclass BTNODE [<dbclass{} KEYTYPE; dbclass{} ENTTYPE >]

Dentro de BTNODE, KEYTYPE y ENTTYPE, son usados como cualquier otro nombre de tipo.
Dada una definición de generador de clase, se pueden crear nuevas clases dándole los parámetros adecuados. Este proceso es llamado **instanciación de una clase**.
Por ejemplo: BTNODE[<dbint, dbint >], donde dbint es una dbclass predefinida.
Además el generador de clases puede ser instanciado con funciones parametrizadas, lo cual le brinda al DBI una solución para los casos en que necesita diferentes rutinas de comparación según el tipo de la clave.

Persistencia en E.

En el lenguaje E, los archivos son introducidos como una nueva clase propia, donde, de acuerdo con el sistema de almacenamiento subyacente, un archivo es un conjunto no ordenado de objetos. Un archivo dado puede contener solo un tipo de objetos, y dicho tipo debe ser una dbclass. La forma de declarar un archivo persistente en disco es:

persistent fileof [<alguna_dbclass>] f ;

Esta declaración es sólo aplicable a objetos de clase fileof y en este ejemplo significa que f es asociado con el mismo archivo físico en toda la ejecución del programa.

Las operaciones sobre archivos están definidas simplemente como funciones sobre la clase archivo, y el acceso a los objetos en el archivo tiene la forma de operaciones sobre punteros (de dbclass). Por ejemplo, hay una operación que retorna un puntero al primer objeto del archivo, una que crea un nuevo objeto en un archivo, etc.

Métodos de acceso independientes de tipo y métodos de los operadores.

Para los métodos de acceso, EXODUS provee una biblioteca de estructuras de índice independiente del tipo, tales como B-tree, grid-files [NIEV84]y hash lineal [LITW80].
Además un DBI puede agregar nuevos métodos de acceso que considere adecuados a la aplicación específica haciendo uso del lenguaje E.

En cuanto a los métodos de los operadores, EXODUS provee los mas usuales, estando disponibles en la biblioteca de métodos independientes de tipos.
Los métodos para operadores más específicos a la aplicación deberán ser provistos por el DBI haciendo uso del lenguaje E.

Generadores de componentes a partir de una especificación del DBI.

EXODUS provee un **generador** para producir un **optimizador de consultas** para lenguajes de consultas algebraicos.[GRAE87]

Finalmente, si bien EXODUS no hace referencia a ningún modelo de datos en particular, sugiere un modelo de datos y un lenguaje de manipulación llamados EXTRA y EXCESS [CARE88].
Como resumen sus autores en [CARE 88], el modelo de datos EXTRA incluye un soporte para objetos complejos con subobjetos compartidos, una mezcla de semántica de datos orientada a valor y orientada a objeto, un soporte de ADT definido por los usuarios.Mientras que el lenguaje de manipulación EXCESS provee facilidades para consultar y actualizar objetos complejos. Además puede ser extendido agregando: funciones y operadores de los ADT.

Estado actual y perspectivas.

El estado actual[CARE87] del proyecto es el siguiente: el generador de optimizadores se encuentra operativo y bajo evaluación, la gran parte del manejador de tipos está operativo y el manejador de objetos almacenables también, el diseño del lenguaje E fue terminado y actualmente se encuentran implementando el compilador E.

III.5 GENESIS.

GENESIS [BATO 86a] [BATO 87b] es un proyecto desarrollado en la Universidad de Texas, Austin. Este proyecto puede clasificarse como ´sistema para armar´. Sus propios autores lo definen como un sistema reconfigurable.

GENESIS se apoya en una teoría sobre la construcción de SGBD. Dicha teoría identifica componentes básicos del software de SGBD, requiriendo que todos tengan la misma interface y mostrando que su composición puede ser realizada de una manera simple. A dichos componentes básicos se les llama **building blocks**. Esta teoría se describe con más detalle en [BATO 85].

La reconfiguración del SGBD es posible por la reorganización de los componentes básicos y la utilización de una biblioteca de módulos. Dicha biblioteca es extendible, aumentando las posibilidades en cuanto a la generación de SGBD.

GENESIS propone una estrutura por capas, donde las operaciones de cada capa sirven de sostén a las de la capa superior. La relación entre capa y capa está dada a través de las **interfases estandar**, las cuales son utilizadas para relacionar una operación conceptual con los algoritmos que la implementan. [BATO 87b]. Establece así una correspondencia entre el nivel abstracto y el nivel concreto tnnto en datos como en algoritmos basado en las **trasformaciones elementales**.

Para reconfigurar el sistema, el DAA(Database Architecture Administrator) tiene que escribir las correspondencias abstracto a concreto en un programa de configuración llamado **programa de arquitectura.** Luego este programa es ejecutado, creando las **tablas de arquitectura** que utilizará GENESIS para realizar las correspondencias.
La correspondencia entre las **definiciones de los datos**, es implementada mediante rutinas llamadas **transformadores de tipos** , que son las que manipulan las tablas que hacen la correspondencia entre la estructura abstracta de un archivo y su contrapartida estructura concreta. Estos transformadores de tipos, según las estructuras que se quieran utilizar, tienen que ser escritos o tomados desde una biblioteca provista por GENESIS. [BATO 86a]
A su vez la correspondencia de las **operaciones** entre el nivel conceptual y el interno es realizada a traves de la composición de **extensores de operación.** Un extensor de operación es una rutina que hace la correspondencia entre la operación sobre el archivo abstracto y una secuencia de operaciones sobre archivos concretos.
Una transacción inicia operaciones sobre archivos y links conceptuales. El **grand central** utiliza los contenidos de las tablas de arquitectura para dirigir la traducción de operaciones conceptuales en su contraparte interna. Para realizar efectivamente la correspondencia nivel a nivel son utilizados los extensores de operaciones. Un modelo de optimización de consultas basado en lo expresado es detallado en [BATO 87a}.
El nivel más bajo de estructuras y operaciones es resuelto por el **manejador de archivos** JUPITER [BATO 86a].

Extensibilidad en estructuras de almacenamiento y algoritmos.

La idea de extensibilidad en estructuras de almacenamiento no está dada sólo en poder agregar nuevas estructuras, sino también en poder construir otras a partir de la composición de las ya existentes.

La extensibilidad en algoritmos consiste en realizar de varias maneras una operación sobre una estructura dada. Por ejemplo, varias formas de recuperación en archivos invertidos o varios métodos para realizar el join de dos relaciones. Al agregar el nuevo algoritmo, se debe indicar además, la función de costo asociada al mismo (para la optimización de consultas).

La forma de poner en práctica la extensibilidad en algoritmos es similar a la de extensibilidad en estructuras de almacenamiento. La diferencia fundamental está en la forma en que se concreta la elección de un método. En el caso de estructuras de almacenamiento, la elección está dada en tiempo de compilación del SGBD, mientras que en el caso de algoritmos está dada en tiempo de ejecución de la consulta que realiza la operación.

Ejemplo: supongamos que se quiere recuperar los registros de un archivo abstracto F que satisfacen la consulta Q utilizando la operacion RET.

```
RET(F,Q)
{   case( F.implemetación ) of
    {       bplus:  RET_BPLUS(F,Q);
            isam:   RET_ISAM(F,Q);
                    .....
            index: RET_INDEX(F,Q);
    }
}
```

Mediante este ejemplo observamos que, cuando se agrega un nuevo caso, lo que se está haciendo es extendiendo las estructuras de almacenamiento.

Como un ejemplo de extensibilidad de algoritmos, supongamos que queremos realizar la operacion JOIN, que realiza el join de dos archivos abstractos A1 y A2.

```
JOIN(A1,A2)
{   case (costo_menor) of
    {       alg1:   NESTED_LOOPS(A1,A2);
            .....
            algn:   MERGE_SCAN(A1,A2);
    }
}
```

Aqui se puede observar que cuando se agrega un nuevo caso, se esta extendiendo los algoritmos que podrían realizar la operación.

Extensibiblidad en tipos de datos y operadores.

Para lograr extensibilidad en tipos de datos y operadores proponen una variante en el modelo funcional, la cual simplifica el modelo computacional. El modelo de datos esta basado en **producciones** en lugar de funciones, donde las producciones son reglas de reescritura de "streams". Un "stream" es una codificación de una **secuencia**, siendo una secuencia un conjunto de objetos entre llaves. Una producción hace la correspondencia entre un "stream" de entrada y un "stream" de salida. La implementación de una producción es un trasformador de "streams".
Por ejemplo: para listar los nombre de los departamentos que tienen empleados con edad superior a 55

```
Dept.WHERE(Empt.age .gt. 55).Dname.PRINT
```

Dept genera la secuencia de todos los departamentos. WHERE elimina aquellos departamentos que no tienen empleados cuya edad sea superior a 55. Dname reeplaza los objetos departamentos por sus nombres y PRINT los imprime.
Aplicaciones de bases de datos no tradicionales son manejadas en forma similar.

Perspectivas.

En cuanto al desarrollo sobre SGBD extensibles, este grupo se encuentra desarrollando un álgebra para el diseño de SGBD moleculares [BAT87a], con la cual pretenden que los aspectos significativos del diseño de arquitecturas de SGBD pueden ser reducidas a un formalismo simple, y reglas por las cuales se componen building blocks (o átomos) pueden ser expresadas algebraicamente.

En cuanto a perspectivas de desarrollo del prototipo, están investigando en dos campos: el de las base de datos y el del software. Con respecto al primero están incorporando al prototipo: control de concurrencia, procesamiento de consultas y control de las interfaces con usuarios finales. En particular sobre este último punto están estudiando la forma de permitir modelos de datos y lenguajes de consulta alternativos, así como la posibilidad de agregar nuevos tipos de datos.

III.6 GEODE

GEODE [PUCH 88] es un proyecto en desarrollo en el INRIA (Francia) y la Universidad de Paris VI. Es un sistema que puede ser considerado a la vez como ´sistema para armar´y como un núcleo de base para un SGBD orientado objeto.

Los principales objetivos del proyecto son :
- ofrecer un sistema abierto que permita la integración de varios modelos de datos.
- permitir la extensión de los tipos de datos por los usuarios.
- autorizar el almacenaje de objetos de gran volumen.
- tener en cuenta la evolución temporal de los datos a partir de la gestión de versiones.
- ofrecer mecanismos de almacenaje de bajo nivel para autorizar el dasarrollo de nuevos métodos de acceso a los datos.
- ofrecer mecanismos de gestión de la concurrencia adaptables a los tipos de datos y parametrables por el usuario.

Principales características

GEODE se presenta como un servidor de operaciones sobre los objetos de la base de datos. La extensibilidad del sistema se sitúa a dos grandes niveles : extensión vertical, a partir de la definición de nuevos tipos y extensión horizontal, a partir de nuevos métodos de acceso de bajo nivel.

La arquitectura de GEODE está organizada en 5 capas principales : gestión de tipos, gestión de Constructores genéricos, gestión de objetos y colecciones, gestión del almacenaje, gestión de la memoria virtual.
Esta arquitectura permite el acceso al sistema a varios niveles .

Búsqueda de buenas performances

GEODE propone técnicas adecuadas para mejorar las performances del SGBD. Ellas conciernen tres aspectos principales : contexto gran memoria, mecanismos de clustering y realización de las E/S en forma diferida.
El **contexto gran memoria** tiene por objetivo tener en cuenta la evolución tecnológica en el dominio de las memorias RAM, partiendo de la idea que para un gran número de aplicaciones se podrá disponer de una capacidad memoria suficiente como para soportar en memoria central todos los datos de una sesión.
Esto supone imaginar estructuras de datos en memoria que permitan optimizar el tiempo de cálculo CPU.

Clustering

Los objetos de un mismo tipo son organizados en Colecciones. El almacenaje de objetos en esas colecciones puede ser secuencial o ser organizados por regrupamientos en clusters .

E/S en diferido

GEODE trabaja como si la base de datos estuviera permanentemente en memoria central. El mecanismo de repercusión de las modificaciones sobre la memoria estable es realizado por un proceso independiente. Esto permite de no bloquear al sistema durante la realización de una escritura sobre disco, lo que significa una ganancia cosiderable de performance.

OIDs

Como EXODUS, GEODE ofrece el tipo de base OID, que representa una referencia única e invariable a un objeto.

Almacenaje de los objetos

El principio expuesto es basado en un almacenaje por páginas. Un objeto puede estar repartido entre varias páginas no forzosamente consecutivas. El acceso a los objetos se realiza a partir de primitivas de manipulación.

Versiones

El mecanismo de versiones permite la creación dinámica de nuevas versiones de objetos a partir de la secuencia de modificaciones realizada sobre ese objeto.
El método utilizado consiste en fabricar una nueva versión por la duplicación del artículo atómico del objeto donde fue hecha la modificación.

Concurrencia

Los algoritmos de control de la concurrencia de acceso a los datos son basados sobre el blocage en dos faces. Varios gránulos son propuestos : colecciones, objetos, sub-objetos y artículo atómico. Además del control de concurrencia clásico, es posible utilizar un mecanismo diferido aplicado únicamente a los objetos que participan del resulatado de una consulta.

Estado actual y perspectivas

GEODE es un proyecto iniciado recientemente (1987). Las capas internas del sistema son operacionales, y actualmente el desarrollo se concentra en la gestión de tipos y en la definición de un lenguaje de interface con características del enfoque objeto.

IV - CONCLUSION

Existe coincidencia en que los sistemas actuales plantean limitaciones para el desarrollo de nuevas aplicaciones, y en utilizar herramientas de otras áreas (ADT, objetos, deducción, etc) para hacer posible dicho desarrollo. Sin embargo, las propuestas existentes difieren en aspectos tales como : hacia quién está dirigido el producto (usuario o DBI), si cuenta o no con un modelo de datos definido (completo o para armar) o en que aspectos de extensibilidad pone mayor énfasis.

Las perspectivas de actividad en esta área parecen muy prometedoras. Los prototipos presentados muestran la puesta en práctica de un conjunto de ideas y soluciones originales. Estos sistemas constituyen actualemente una base importante para el prototipaje y desarrollo de nuevos modelos de datos y lenguajes de interrogación de tipos declarativo. A su vez, la confluencia con otras disciplinas, en particular con la inteligencia artificial, lenguajes funcionales y el enfoque orientado a objetos, han generado un campo importante de trabajo e investigación.

AGRADECIMIENTOS.

En su pasaje por Montevideo, el profesor Isidro Ramos de la Universidad de Valencia, tuvo la gentiliza de comentar este trabajo, por lo cual le estamos sumamente agradecidos. También han contribuido Phillipe Pucheral de la Universidad Paris VI, aportándonos ideas y preciosa documentación; así como Guillermo Calderón y Oscar Sanz, cuyas observaciones nos fueron sumamente valiosas.

BIBLIOGRAFIA

Adib 86 Adiba M., Bui N. CCollet C. ´Aspects Temporels, Historiques dans les Bases de Données. Revue TSI 1987.

Astr 76 Astrahan M. et al. ´System R: Relational Approach to Database Management´, ACM Trans. on Database Systemas, Vol 1, No 2. Junio 1976.

Banc 88 Bancilhon F. ´Object-Oriented Database Systems´. Invited Lecture. 7th ACM. Symposium on Principles of Data Base Systems. Austin. Texas. Marso 1988.

Bato85 Batory D. S. "Modeling the Storage Architectures of Commercial Database Systems". ACM - TODS 10,4 (Diciembre 1985).

Bato86a Batory D.S. "GENESIS: A reconfigurable database management system". To apear IEEE 1988.

Bato86b Batory D.S and T.Y.Leung. "Implementation Concepts for an Extensible Data Model and Data Language". TR-86-24. University of Texas, Austin. 1986.

Bato87b Batory D.S. "Principies of Database Manajement System Extensibility". IEEE Junio 1987.

Bent 75 Bentley J.L. "Multidimensional search trees user for associative searching". Communications ACM 18,9. (Setiembre 1975).

Bent 79 Bentley J.L. "Multidimensional binary search trees in database applicatios". IEEE Transactions in Software Engineering SE-5, 4. (Julio 1979).

Bern 87 Bernstein, Lomet. "CASE Requiremest for extensible database systems" IEEE Junio 1987.

Care 86 Carey M., DeWitt D. "Extensible database systems". "On Knowledge base management systems" Springer -Verlag. 1986.

Care 87 Carey M., DeWitt D. "An overview of the EXODUS Project". Database Eng. , Junio 1987.

Care 88 Carey M., DeWitt D, Vanderberg S. "A Data Model and Query Language for EXODUS". 1988.

Chou 85 Chou H. and DeWitt D. "An evaluation of buffer management strategies for relational databases". Proc. of the 11th VLDB Conf. Stockholm, Swedem, Agosto 1985.

Daya 85 Dayal U. Smith J. ´PROBE : A Knowledge-Oriented Database Management System´, Proceeding of the Islamorada Workshop on Large Sacle Knowledge Base ans Reasoning Systems, Febrero 1985.

Fag 79 R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong. "Extendible hashing-a fast access method for dynamic files". ACM Transactions on Database Systems 4,3. (Setiembre 1979), 315-344.

Gard 87a Gardarin G., Simo E. ´Les Systemes de Gestion de Bases de Données Deductives´, Revue TSI. 1987

Gard 87b Gardarin G., Pucheral P. ´Optimization of Generalized Recursive Queries Using Graph Traversal´, Informe INRIA. 1987.

Gold 87 Goldhirsch D. Orenstein J. "Extensibility in the PROBE Database System". Proc. of Extending database technology, Venesia 1987.

Grae 87 Graefe G., and DeWitt D. "The EXODUS Optimizer Generator". Proc. of the 1987 SIGMOD Conf. San Fransisco, CA. Mayo 1987.

Gutt 84 Guttman T. ´R-Tree : A Dynamic Index Structure for Spatial Searching´, Procceding the 1984 SIGMOD Conference, BOston MA Mayo 1984.

Hask 83 Haskin R., Lorie R. "On Extending the Functions of a Relational Database System". Proc. of ACM Enineering Design Applications (1983).

Katz 83 Katz R.H., Weiss S. ´Transaction Management for Design Databases´, Computer Sciences Technical Report 496, Febrero 1983.

Katz 86 Katz, Chang, Bhateja. "Version modeling concepts for computer-aided design databases". Proc 1986 ACM-SIGMOD. International Conference on Managemente of Data. Washington DC, Mayo 1986.

McPh 87 Mc Pherson J. , Pirahesh H. ´An Overvieuw of Extensibility in Starburst´, IEEE. Vol 10 no 2. Junio 1987.

Lars 80 Larson P. "Linear hashing with partial expansions" . Proceedings 6th Conference on VLDB, Montreal, Canada , 1980, pp. 224-232.

Litw 80 Litwin W. "Linear Hashing: A new tool for file and table addressing". Proc. 6th Int´l Conf. on VLDB, Montreal, 1980, pp. 212-223.

Litw 87 Litwin W and Lomet. "A new method for fasst data searches with keys". IEEE Software 4,2 (Marzo 1987).

Lome 83 Lomet D. B.. "Bounded index exponential hashing". ACM Transactions on Database Systems 88,1. (Marzo 1983), 136-165.

Lome 87 Lomet D.B. "Partial expansions for file organizations with an index". ACM Transactions on Database Systems 12,1. (Marzo 1987).

Lori 83 Lorie R., Plouffe W. "Complex Objects and their use in design transactions". Proc of ACM Engineering Design Applicatios (1983).

Niev 84 Nievergelt J., Hintenberger H., Sevcik K.C. ´The Grid File : An Adaptable, Symmetric Multikey File Structure´, ACM Transaction on Database Systems, Vol 9 no 1, Marzo 1984.

Rich 87 Richardson J., Carey M. "Programming Constructs for Database System Implementation in EXODUS". Proc. of the 1987 SIGMOD Conf. , San Fransisco, CA. Mayo 1987.

Robi 81 Robinson J.T. ´The k-d-B-tree : A Search Structure for Large Multidementional Dynamic Index´, Proccedings of the 1981 SIGMOD Conference, Junio 1981.

Puch 87 Pucheral P. ´Traitement Efficace des Regles Récursives par un Opérateur Basé sur des Parcours de Graphe´, IIIe Journées Bases de Données Avancées, Port Camargue , 1987.

Puch 88 Pucheral P., Thevenin J.M., Steffen H. ´GEODE : Un Gestionnaire d´Objetos Extensible Orinté Grande Mémoire´, Proceeding of Base de Données de Troisieme Génération. Benodet, Junio 1988.

Lori 83 Lorie R., Plouffe W. "Complex Objects and their use in design transactions". Proc of ACM Engineering Design Applicatios (1983).

Niev 84 Nievergelt J., Hintenberger H., Sevcik K.C. ´The Grid File : An Adaptable, Symmetric Multikey File Structure´, ACM Transaction on Database Systems, Vol 9 no 1, Marzo 1984.

Rich 87 Richardson J., Carey M. "Programming Constructs for Database System Implementation in EXODUS". Proc. of the 1987 SIGMOD Conf. , San Fransisco, CA. Mayo 1987.

Robi 81 Robinson J.T. ´The k-d-B-tree : A Search Structure for Large Multidemention al Dynamic Index´, Proccedings of the 1981 SIGMOD Conference, Junio 1981.

Puch 87 Pucheral P. ´Traitement Efficace des Regles Récursives par un Opérateur Basé sur des Parcours de Graphe´, IIIe Journées Bases de Données Avancées, Port Camargue , 1987.

Puch 88 Pucheral P., Thevenin J.M., Steffen H. ´GEODE : Un Gestionnaire d´Objetos Extensible Orinté Grande Mémoire´, Proceeding of Base de Données de Troisieme Génération. Benodet, Junio 1988.

Sche 82 Scheuermann P. and M. Ouksel. "Multidimensional B-trees for associative searching in database systems". Informations Systems 7,2. (1982).

Ston 84a Stonebracker et al. "QUEL as a data type". Proc. 1984 ACM-SIGMOD.

Ston 86 Stonebraker M. and Rowe L. "The design of POSTGRES". Proc. 1986 ACM-SIGMOD Conference on Management of Data. Washington D. C. Mayo 1986.

Ston 88 Stonebraker M., Anton J., Hirohama M. "Extendability in POSTGRES". To appear.

Zani 83 Zaniolo. "The database lenguaje GEM". Proc 1983 ACM-SIGMOD.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

GERENCIAMENTO DE ESQUEMAS DE DADOS DINAMICAMENTE MODIFICAVEIS EM MODELO DE BASE DE DADOS ORIENTADOS A OBJETOS.

Caetano Traina Júnior - Jan Frans Willem Slaets
ICMSC - USP - São Carlos IFQSC - USP - São Carlos
Av. Dr. Carlos Botelho, 1465 - Fone (0162) 72-4496
C.E.P. 13.560 - São Carlos - Est. São Paulo - Brasil

Resumo

Este trabalho apresenta um modelo de dados orientado a objetos, desenvolvido para suportar a construção de Sistemas de Gerenciamento de Dados para Aplicações de Projeto, permitindo a representação de grande conteúdo semântico. O modelo favorece a construção de Meta-Sistemas, onde a especificação dos requisitos da própria aplicação deve ser mantida na Base de Dados. Através de conceitos que habilitam o modelo a gerenciar a manipulação de versões e alternativas de projeto, cria-se o suporte para que a manutenção de esquemas de dados possa ser feita dinamicamente, simultaneamente com a operação normal do SGBD, sem impacto sobre os dados já armazenados.

1. - Introdução

Os Sistemas de Gerenciamento de Bases de Dados (SGBD) disponíveis atualmente foram em sua grande maioria desenvolvidos tendo por objetivo atender a aplicações comerciais, em que a tônica consiste na manipulação de um volume muito grande de dados estruturalmente idênticos. Por outro lado, aplicações não comerciais dos sistemas de computação (não convencionais para os SGBDs), com freqüência manipulam dados que não tem aquela mesma homogeneidade estrutural. Nesse último caso encontram-se os sistemas de Projeto Apoiado por Computador - PAC (CAD), Fabricação Apoiada por Computador - FAC (CAM), Sistemas de Automação de Escritório, Inteligência Artificial, etc.

Assim, têm sido pesquisados modelos para SGBDs que possam dar o suporte necessário a um amplo espectro de aplicações não convencionais [HAM_81], [FAR_85], [EAR_85] [ABI_87], e portanto que possam ser empregados em sistemas que devam ser integrados em sistemas maiores. Levantamentos de suas necessidades têm sido feitos e apresentados, tal como em [MAT_81], [HAR_85], [BIC_86] e [NAV_86], tendo-se tornado comum o termo **Sistema de Gerenciamento de Dados para Engenharia (SGDE)** para designá-los. Uma das principais necessidades corresponde à possibilidade de se poder alterar o esquema da base de dados mesmo depois desta já ter sido inicializada, uma vez que uma base de dados para projeto tem o seu esquema em constante evolução junto com o próprio projeto.

Pode entender-se um **Meta-Sistema** como um sistema que usa uma estrutura semelhante à sua própria estrutura para definir-se a si mesmo. Por exemplo, uma linguagem que ensine ou especifique uma linguagem será uma Meta-Linguagem; um programa de computador que possa ser definido, feito ou melhorado quando se usa esse mesmo programa será um Meta-Programa. Um Meta-Sistema de Gerenciamento de Dados deve poder aceitar a definição do Esquema de Dados da

mesma maneira que aceita os dados, pois o proprio esquema de dados deve ser encarado como um conjunto de dados que deve ser gerenciado. Tal enfoque implica que o próprio esquema possa ser modificado ou redefinido dinâmicamente, tal como o podem ser os dados extensionais armazenados na Base de Dados.

Neste trabalho apresenta-se um modelo de dados denominado **Modelo de Representação de Objetos - MRO** [TRA_88] que permite a construção de SGDEs que atendem a um amplo espectro de aplicações, incluindo aplicações integradas de projeto e produção de sistemas em geral, e em especial que permite a construção de Meta-sistemas de Gerenciamento de Dados [TRA_86], em que a especificação dos requisitos da própria aplicação devam ser mantidos na Base de Dados. Nas seções seguintes os conceitos principais que caracterizam o MRO são apresentados, com ênfase na maneira como tal modelo permite a operação do SGBD nele apoiado mesmo em face de um esquema de dados dinamicamente em alteração.

2. - A Fundamentação do Modelo

A fundamentação matemática de modelos comerciais mais recentes, tais como o modelo Relacional e o Modelo Entidade Relacionamento (ME-R), é a álgebra de relações [NG*_81], [PAR_85]. Esta é adequada para a manipulação de estruturas de dados que possam ser bem representadas através de tabelas. No caso de um modelo de dados voltado à representação semântica mais intensa, apoiado na modelagem de objetos, tal como o apresentado nesse trabalho, a teoria dos grafos torna-se no entanto mais atraente. A Teoria dos Grafos é mais adequada, uma vez que o reconhecimento dos objetos como nós de um digrafo corresponde exatamente à ideia de que todos os objetos, independentemente de seus tipos, tem a mesma estrutura, tal como todos os nós de um digrafo tem a mesma estrutura funcional.

O Modelo de Representação de Objetos procura modelar objetos (entes, eventos, coisas, conceitos), tratando cada um individualmente, e mantendo na representação de cada um todas as informações pertinentes sem tentar estabelecer, a priori, padrões para a representação de conjuntos de objetos, tal como no Modelo Entidade-Relacionamento, onde a representação das informações é feita na realidade através das relações de dados das entidades e dos seus relacionamentos, especificados no esquema de dados de forma imutável depois da inicialização da base de dados.

Para que se possa usar um digrafo para a representação de informações, deve-se associar as informações aos elementos que constituem o digrafo. Assim, os nós são a representação dos objetos que existem no mundo real, e os arcos representariam as associações que existem entre eles, as quais no MRO são denominadas relacionamentos. Assim, da mesma maneira que um nó corresponde aos pontos de identificação (e portanto de acesso) de um digrafo, um objeto centraliza todas as referências e informações (inclusive de identificação) que são mantidas no sistema.

A categorização dos nós de um digrafo classifica os objetos atribuindo-se um **Tipo** para cada objeto. Dessa forma, a estrutura de todos os objetos é idêntica (como o são todos os nós de um digrafo), e a classificação segundo sua categoria é um atributo obrigatoriamente associado a todos os objetos. O mesmo é feito quanto ao conceito de ponderação de arcos na teoria dos grafos,

que passa a classificar os relacionamentos entre os objetos atribuindo a todo relacionamento um "**tipo**".

Assim, no MRO os objetos e os relacionamentos são agrupados segundo seus respectivos tipos, e não em conjuntos de entidades e de relacionamentos. Isso permite que na implementação de um SGBD apoiado no MRO todos os relacionamentos e todos os objetos sejam armazenados em apenas um arquivo de relacionamentos e um arquivo de objetos.

Cada relacionamento do MRO indica que os dois objetos estão associados um ao outro, ou seja, no digrafo correspondente, devem existir dois arcos, um em cada direção. Isso não significa que o digrafo seja simétrico, pois cada um dos arcos representa uma forma diferente de associação entre entidades. Efetivamente, cada arco representa um **relacionamento oposto** ao representado pelo seu par. No MRO todo relacionamento tem o seu oposto correspondente.

Os **atributos** que são associados aos objetos e relacionamentos no mundo real são também classificados no MRO segundo seus vários tipos.

Os objetos podem ser diretamente acessados através de seu identificador, porém relacionamentos e atributos somente são acessados através das associações que mantêm com os objetos. Além disso, os atributos de relacionamento somente são acessados através dos relacionamentos aos quais estão associados, os quais por sua vez devem ser acessados através dos objetos que, esses sim, podem ser diretamente acessados (no entanto objetos podem também ser acessados através dos relacionamentos que mantém com outros).

2.1. - Um Diagrama de Representação de Objetos

Para a especificação de um esquema de dados usando um determinado modelo de representação, o uso de convenções gráficas é freqüentemente proveitoso, pois facilita a compreensão das informações. Portanto é interessante que se disponha de uma simbologia adequada, capaz de representar graficamente os recursos e conceitos do MRO. Assim, quando alguma informação é modelada usando-se o MRO, ela pode ser graficamente representada através de um **Diagrama de Representação de Objetos - DRO.**

Sendo apoiado na Teoria dos Grafos, os simbolos usados nos DROs são provenientes das representações gráficas tradicionalmente usadas para grafos, usando-se círculos para representar objetos, e arcos de circunferências para representar relacionamentos.

No entanto, em geral os diagramas elaborados não têm a intenção de representar instâncias de objetos ou de relacionamentos, mas sim de representar como os tipos de objetos e tipos de relacionamentos se associam. Portanto, é interessante que se distinguam dois tipos de diagramas: um que corresponde aos diagramas onde se representam os tipos de objetos e tipos de relacionamentos, os Diagramas de Representação de Objetos (DRO) propriamente ditos; e um outro em que se representam instâncias de objetos e relacionamentos, em geral com o propósito de servirem de exemplos, denominado **Diagrama de Representação de Instâncias - DRI.**

Na figura 1a) mostram-se os simbolos usados para a representação gráfica de Tipos de Objetos e de Tipos de Relacionamentos, e na figura 1b) mostram-se os simbolos usados para a representação de Objetos e de Relacionamentos.

Os Tipos de Objetos são representados através de um Círculo dividido ao meio. Na metade superior indica-se o Nome do Tipo de Objeto representado, e na metade inferior indica-se o Nome do Tipo de Colônia onde aquele tipo habita (Colônia e Habitação são conceitos descritos na Seção 4.).

Os Tipos de Relacionamentos são representados através de arcos de circunferências ligando os tipos de objetos que são associados por aquele tipo de relacionamento. Além disso, cada tipo de relacionamento é unido por um traço com o seu tipo de relacionamento oposto. Esse traço deve formar um ângulo agudo com cada um dos arcos, e o Nome do Tipo de cada relacionamento será escrito a partir desse ângulo, dando a indicação de qual é o nome de cada um dos dois tipos de relacionamento.

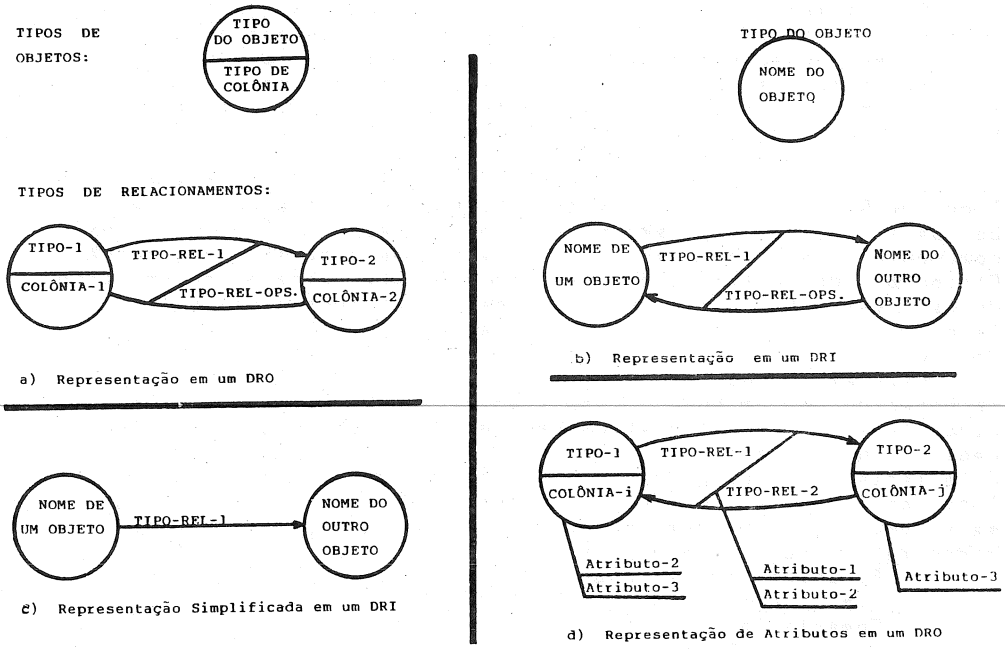


Figura 1 - Representação gráfica de peças do MRO: em a) estão os símbolos das peças em um DRO; em b) os símbolos das peças em um DRI; em c) um DRI simplificado; em d) atributos associados a objetos e relacionamentos em um DRO.

A principal diferença gráfica entre um DRI e um DRO é que os objetos são representados nos DRIs através de círculos não cortados. No interior do círculo escreve-se o nome do objeto. Se necessário, o tipo do objeto é escrito fora, acima do círculo. Em um DRI não se indica o nome da Colônia onde o objeto habita. Os relacionamentos são representados da mesma maneira que nos DROs.

Para efeito de simplificar os DRIs, os relacionamentos opostos podem ser omitidos nesses diagramas, uma vez que os tipos opostos podem ser deduzidos trivialmente através dos DROs e dos tipos dos objetos e do relacionamento envolvido. A figura 1c) mostra o mesmo relacionamento da figura 1b) representado de maneira simplificada.

Os Tipos de Atributos que podem ser associados aos tipos de objetos e de relacionamentos podem também ser expressos nos DRIs e DROs. A figura 1d) mostra um exemplo de como representam-se gráficamente associações de diversos tipos de atributos a tipos de objetos e de relacionamentos.

3. - Características de Tipos de Atributos

Para aumentar o conteúdo semântico do modelo, e facilitar o desenvolvimento dos sistemas que usem SGDEs nele apoiados, identificam-se características comuns em grupos de tipos de atributos, de forma a possibilitar que um SGDE ofereça maiores recursos de tratamento dos tipos de atributos de significado mais usados. Note-se que Tipos de Atributo e Tipo de Dado de um Tipo de Atributo são conceitos distintos.

Para isso deve-se caracterizar o significado de determinados Tipos de Atributos, que possam ser reconhecidos e individualizados para o SGDE. Por exemplo, em uma aplicação de PAC um atributo comumente usado é o que se refere às coordenadas de posição. Atributos desse tipo podem ser representados através de um tipo de dado Vetor_Real_de_ Três_Posições, porém apesar de ser descrita por três valores reais, coordenada_de_posição é mais significativo do que indicar esses três valores como apenas números.

Existem vários tipos de atributos cujo significado pode ser reconhecido por um SGDE. Cada significado que possa ser reconhecido terá uma **Característica de Tipo de Atributo**. Dessa forma, a cada Tipo de atributo estará associado um Tipo de Dado e uma Característica, os dois constituindo o Significado associado ao Tipo de Atributo. Entre as Características de Tipo de Atributo mais comuns a um Sistema de Gerenciamento de Dados de Engenharia podem ser citadas as seguintes:

- **Sinônimo.** Corresponde a um identificador alternativo para um objeto;
- **Comentário.** Corresponde a informações para o usuário, que não são analisadas pelo sistema;
- **Posição.** Corresponde à informação de localização no espaço, podendo indicar um ponto, uma orientação, ou ambos.
- **Tempo.** Indicação de tempo relativo ou absoluto associado aos objetos e relacionamentos.
- **Conjunto de Dados.** São agrupamentos de informações semelhantes, que normalmente são representados em estruturas de arranjos de dados (vetores e matrizes).
- **Procedimento.** São fórmulas ou parâmetros associados a atividades que devem ser exercidas de uma forma pré-determinada pelo sistema.
- **Regra.** São atributos que possuem uma condição e uma ação associados.

Tipos de Atributos que não tenham alguma característica especial que os distingua dos demais são considerados tipos de

atributos normais, referenciados simplesmente como **Propriedades** dos objetos ou relacionamentos a que estão associados. As características dos tipos de atributos representados em um DRO (ou dos atributos representados em um DRI) podem ser indicadas precedendo o nome do tipo do atributo por um sinal que o caracterize, tal como indicado na lista de características de atributo acima.

4. - Escopo de Abrangência dos Identificadores de Objetos

Em Modelos de dados fundamentados na Álgebra das Relações o atributo (ou os atributos) usado para identificar uma tupla deve ser capaz de identificá-la univocamente no conjunto a que pertence. Tuplas pertencentes a conjuntos distintos podem ter um identificador comum sem que isso represente um problema. O mesmo vale para os identificadores de relacionamentos. Se um atributo não é suficiente para identificar univocamente uma determinada tupla, combinam-se mais atributos, para chegar a um conjunto de atributos que constitua uma chave de acesso que permita identificar univocamente cada tupla do conjunto.

Já no MRO obrigatoriamente apenas um campo deve identificar cada objeto, e como todos os objetos são mantidos em uma única estrutura de dados, conclui-se que cada objeto deveria ter um Nome (e tantos Sinônimos quanto necessário) que fosse único em toda a Base de Dados.

Para evitar essa restrição, o modelo permite restringir o escopo de abrangência de um identificador usado para localizar um objeto.

Quando visto de maneira global, um sistema apresenta um pequeno grau de detalhe. Quando o interesse recai em uma de suas componentes, o grau de detalhe aumenta, porém toda a informação que então se apresenta refere-se diretamente àquela componente que está sendo observada.

Quando se manipula a base de dados de um ponto de vista global, são poucos os objetos acessíveis. Indicando-se ao sistema que se pretende considerar a seguir um desses objetos, indica-se uma região da Base de Dados em que se pretende obter ou manipular informações. Dessa forma tornam-se acessíveis novos objetos. Esse procedimento pode ser repetido, indicando-se para consideração um dos novos objetos tornados disponíveis, dessa forma aumentando-se sucessivamente o nível de detalhe, e correspondentemente a quantidade de objetos a que se tem acesso diretamente através de seus Nomes (ou Sinônimos).

Cada vez que um objeto é colocado em consideração, aumenta-se um nível de detalhe, o que torna disponíveis os objetos que detalham mais o objeto considerado. Note-se que com o refinamento de um determinado aspecto, os demais não deixam de estar acessíveis, apenas não são por sua vez refinados.

O MRO conta com essa capacidade devido aos conceitos de Colônia de Objetos, Objeto Considerado, Objetos Habitantes de uma Colônia, e Contexto.

Quando se procura acessar um objeto armazenado numa base de dados apoiada no MRO, se for conhecido o seu código (a indicação univoca de um nó de um digrafo) então o acesso é imediato, uma vez que o código identifica univocamente um objeto dentre todos os objetos da base de dados. No entanto, quando a identificação é

feita pelo nome (ou por um sinônimo), essa identificação não é unívoca, uma vez que pode (a princípio) existir mais de um objeto com um mesmo nome. Para que se possa identificar um objeto pelo seu nome univocamente, é necessário que se especifique o **Contexto** onde esse nome está definido. A caracterização da existência de contextos é feita através da partição dos identificadores existentes na base de dados em **Colônias**, de maneira que um nome esteja definido somente no domínio de uma Colônia.

Cada **Colônia de Objetos** é constituída por um conjunto de objetos, e se refere não somente a um nível específico de detalhe de informação, mas também a um determinado aspecto de interesse, associado ao objeto do qual se deseja maiores detalhes.

Para isso, os objetos de uma Base de Dados devem ser classificados segundo diferentes aspectos, sendo essa classificação feita usando os diferentes Tipos de Objetos do Esquema de dados. Dessa forma, cada objeto pertence a apenas uma Colônia de Objetos, a qual é determinada através do seu tipo de objeto, e é constrita pelo objeto que caracteriza o aspecto particular do qual esse objeto é um detalhe.

O conceito de Colônias de Objetos divide os objetos de uma Base de Dados em vários aspectos de interesse, que são definidos pelos Tipos de Objetos que compõem o Esquema de Dados daquela base. O conceito estabelece portanto uma partição também no conjunto de Tipos de Objetos, sendo cada conjunto de Tipos de Objetos denominado **Tipo de Colônia**. Os objetos da base de dados estarão também particionados segundo os seus tipos, em correspondência com a partição que os Tipos de Colônias estabelecem sobre o conjunto de todos os Tipos de Objetos definidos na base. No entanto, além de estarem particionados segundo seus tipos (através dos Tipos de Colônias), os objetos da base de dados estarão separados também pelas várias Colônias de Objetos, uma vez que podem existir várias Colônias de Objetos correspondendo a um mesmo Tipo de Colônia. Os objetos que pertencem a uma colônia são ditos **habitantes** dessa colônia. Cada objeto habita exatamente uma colônia.

Os objetos que habitam uma colônia, tornam-se acessíveis quando a colônia a que pertencem é **aberta para acesso**. Num dado instante somente uma colônia de cada tipo pode estar acessível de cada vez. Assim, das várias colônias de objetos que correspondem a cada tipo de colônia apenas uma pode estar aberta num dado instante. Pode-se abrir uma colônia para permitir que os objetos se tornem acessíveis para consulta e modificação ou apenas para consulta.

Uma das colônias terá que estar permanentemente disponível, e será denominada **Colônia Global**. Os objetos da Colônia Global são os que estão disponíveis quando se está no nível menos detalhado de acesso ao sistema, e permanecem sempre disponíveis. Correspondentemente existe um **Tipo de Colônia Global**, o qual contém os tipos de objetos que existem na Colônia Global. Dessa forma, todos os objetos dos tipos que pertencem à Colônia Global estarão permanentemente disponíveis.

Objeto Considerado é um objeto que se tem presentemente disponível, e que indica-se para consideração, o que faz com que os objetos que o especificam com maior grau de detalhe se tornem também disponíveis. Um objeto está disponível quando a Colônia de Objetos em que ele habita está aberta.

Indicar um objeto para consideração caracteriza uma operação

de Abrir uma Colônia de Objetos para acesso, tornando disponíveis os objetos que habitam aquela colônia. Os objetos que habitam essa colônia constituem-se na **Colônia de Objetos Constrita pelo Objeto Considerado**. Quando considera-se um objeto, indica-se implicitamente uma ou mais colônias de objetos. A indicação de um objeto para consideração tem por objetivo abrir novas colônias de objetos para acesso, e a indicação de um objeto pode abrir mais de uma colônia. Dessa forma, o Conjunto de Objetos Constritos por um objeto pode envolver uma ou mais Colônias de Objetos.

Quando indica-se para consideração um objeto que constringe uma Colônia de Objetos correspodente à um Tipo de Colônia já aberto, este é **fechado**, o que faz com que se feche a Colônia de Objetos correspondente que estava aberta; a seguir o mesmo Tipo de Colônia é novamente aberto, agora tornando disponíveis os objetos da Colônia de Objetos constrita pelo Objeto Considerado agora. Outros objetos, pertencentes à colônias que não aquela focalizada implicitamente pela indicação de um novo objeto para consideração, e a correspondente operação de abertura de colônia, não são afetados por essa operação.

Se o usuário desejar referir-se a qualquer dos objetos que habitam qualquer uma das colônias abertas, o Nome do Objeto e a indicação de seu Tipo será suficiente para identificá-lo. No entanto, se for necessário referir-se a algum objeto que habita uma colônia não aberta, ou a um objeto que habita uma colônia de objetos constrita por um objeto que não o presentemente considerado, então será necessário que se identifique todo o caminho de acesso, desde o ponto de interesse atual na base de dados até o ponto onde está o objeto a que se quer se referir.

Com esses conceitos, o Modelo de Representação de Objetos passa a permitir que objetos que pertencem a Colônias de objetos Constritas por objetos distintos tenham nomes iguais, e mesmo assim possa manter-se a sua identificação inequívoca.

5. - O Controle de Versões e Alternativas de Projeto

O projeto de um produto de engenharia é um processo de refinamento sucessivo, em que os conceitos imaginados são transformados em objetos reais, as especificações genéricas são detalhadas, o todo é analisado em partes até os menores detalhes. No processo de projeto, o engenheiro parte de especificações genéricas ou de produtos realizados anteriormente, e vai sucessivamente incorporando novos dados, ou modificando partes já definidas anteriormente. Uma ferramenta de apoio à projetos deve verificar a consistência das novas informações e das mudanças efetuadas, comparar com as restrições impostas ao projeto, usando todas as informações presentes na Base de Dados que sejam correlatas a cada passo do processo.

Muitas versões de um projeto podem existir simultaneamente, e essa característica impõe ao SGDE a capacidade de poder armazenar e gerenciar múltiplas versões e as alternativas propostas para um projeto. Katz [KAT_84] [KAT_86] caracteriza "Versão" e "Alternativa" de um projeto considerando que um projeto está inicialmente em progresso, e que sobre esse projeto os projetistas efetuam mudanças e incorporação de dados. As alterações feitas para experimentar outras formas de implementação são **Alternativas**, e as alterações que são oficialmente aprovadas para

fazer parte do produto final são chamadas **versões**. Note-se que um projeto pode ter muitas versões e muitas alternativas, sendo que as versões produzidas são armazenadas para sempre.

O problema de controle das versões produzidas por um processo de desenvolvimento de software tem sido estudado sob a denominação de Gerenciamento de Configuração de Software [BER_84]. Sob esse enfoque, a menos de não se cuidar do conceito de "Alternativa", trata-se do mesmo problema, restrito ao desenvolvimento de software. O Gerenciamento de Configuração de Software, ataca problemas semelhantes aos de gerenciamento de configurações em geral, os quais foram re-estudados sob o enfoque das necessidades de configurações (e versões) de projeto de software. No entanto, os conceitos desenvolvidos podem ser aplicados ao Gerenciamento de Configurações de Projetos em geral, e serão aplicados neste trabalho como um recurso de auxílio ao gerenciamento e controle de desenvolvimento de projetos em geral, armazenados em uma base de Dados suportada pelo MRO.

Note-se que o Gerenciamento de configuração de projetos é muito mais semelhante ao Gerenciamento de Configuração de Software do que ao Gerenciamento de Configuração em geral, uma disciplina bastante estruturada que se aplica ao estudo do gerenciamento de bens em geral que possam ser produzidos. Isso porque um projeto - quando pensado em termos de sua concepção, motivos, decisões, etc, e em especial quando é constituído dos dados armazenados em uma base de dados - é muito mais semelhante ao software, no que diz respeito a maneira como deve ser gerenciado. Assim, as alterações nas regras de Gerenciamento de Configuração em geral, que tem sido estudadas para adequá-las ao Gerenciamento de Configuração de Software, continuam válidas para o Gerenciamento de Configuração de Projetos, mesmo que o bem final produzido por um dado projeto seja mais adequadamente controlado pelas regras de Gerenciamento de Configuração em geral.

As diversas perspectivas dos usuários são contempladas pelo MRO, implementado-se o esquema de proteção e/ou ocultamento de informações pelo controle de acesso às Colônias de Objetos. O acesso a Colônias de Objeto pode ser controlado de forma a que quando uma colônia é aberta, permite-se abrí-la apenas para consulta ou também para atualização, permitindo assim que a proteção seja feita a nível de colônias. Para obter acesso aos dados de interesse, os programas de aplicação precisam informar quais colônias devem ser abertas e quando devem ser fechadas.

Esse esquema de proteção tem uma correspondência direta com as operações de alto nível dos programas de aplicação, uma vez que se reflete nas operações solicitadas pelos usuários do programa de aplicação. Por exemplo, se um usuário estiver usando um editor para modificar o conteúdo da base de dados, ele provavelmente estará trabalhando com apenas parte da base toda, restrita a alguma colônia de objetos local ao seu ponto de interesse num dado instante. Com a proteção atuando a nível de colônias, a colônia onde ele estiver trabalhando estará protegida do acesso por qualquer outro usuário, ao mesmo tempo em que não poderá ter acesso a uma área de interesse sendo acessada por outro usuário.

Em operação normal, qualquer usuário tem acesso para leitura na colônia global, e indicando sucessivamente vários objetos para consideração, várias colônias vão se tornando acessíveis. Ao se indicar cada objeto para consideração, indica-se também se a colônia que o objeto considerado abre deve ser aberta para apenas

leitura ou para leitura e escrita.

Necessariamente, todos os usuários deverão ter acesso à colônia global para que qualquer outra colônia possa ser acessada. Porém, se for considerado que em uma base correspondente a um projeto já adiantado, envolvendo vários usuários, a colônia global já deverá estar definida e razoavelmente estática, então qualquer usuário necessitará ter acesso a ela apenas para consulta, permitindo que muitos usuários compartilhem a consulta à colônia global, e apenas em colônias em nível hierárquico mais baixo cada usuário obterá permissão individual para atualizações. A atualização da colônia global será permitida apenas a usuários privilegiados, tal como o gerente do projeto, o qual para atualizá-la poderá requerer que naquele instante ninguém mais tenha acesso à qualquer porção da Base de Dados.

O MRO Suporta a armazenagem das várias versões e alternativas de um projeto permitindo a geração de **Instâncias de Colônias de Objetos** dentro da mesma base de dados, de maneira que uma colônia constrita por um objeto possa ter várias versões e alternativas de projeto. Cada instância é chamada de uma **Variante**, e pode corresponder à uma Versão ou à uma Alternativa, e terá atributos de acesso e proteção independentes das demais instâncias do mesmo objeto, de maneira que cada uma poderá ser acessível por um usuário de maneira independente das demais.

Considere-se o desenvolvimento de um determinado aspecto de um projeto. Isso normalmente é efetuado por um projetista usando algum tipo de editor interativo, que lhe permite ir colocando novos dados, alterando os já existentes, até que o projetista fica satisfeito, e aquele aspecto do projeto é considerado terminado. O fato de aquele aspecto do projeto requerer atenção especial por parte de um projetista caracteriza que esse aspecto deve ser um **item de Configuração de Projeto (ICP)**, que deve ter atenção por parte do **Gerenciador de Configuração do Projeto**. Esse aspecto irá então constituir um ICP, e poderá ou não envolver outros ICPs.

Para ser um ICP, aquele determinado aspecto foi identificado como tal, e lhe foi atribuído identificação e um **registro de histórico** para acompanhamento, o qual indica continuamente seu estado para acesso. A identificação corresponde no MRO à existência de um objeto, que constringe a colônia de objetos que o descreve. Para que pudesse ser acessado pelo projetista, seu registro de histórico deveria estar indicando que ele estava em desenvolvimento, e a partir do instante em que o projetista passasse à edição indicando aquele objeto para consideração, seu registro de histórico indicaria o fato, evitando que outros projetistas pudessem acessá-lo ao mesmo tempo.

Cada ICP é armazenado no MRO como uma colônia. As instâncias de colônias que devem ser consideradas versões de um ICP será "Congelada", significando que qualquer usuário somente poderá ter acesso para leitura a essa colônia. Uma versão não poderá nunca ser modificada, e se alguma alteração for necessária àquele ICP, então será necessário que se crie uma nova alternativa que, essa sim será modificada, e uma vez completada a alteração, poderá vir a ser uma nova versão, se assim desejado. Em termos de operações na base de dados, isso será efetuado com a instanciação da colônia constrita pelo ICP a ser alterado, criando-se assim duas instâncias para aquela colônia. A instância que corresponde à versão original permanecerá protegida contra escrita, e a outra

instância, que corresponde à nova alternativa criada (inicialmente idêntica à versão original) estará liberada para alterações, e será então editada. Se a alteração deve levar à uma nova versão, ela será por sua vez a seu tempo também congelada, e protegida contra escrita.

Cada usuário pode ter apenas uma das instâncias de cada colônia disponível num dado instante, a qual é chamada de **Variante Corrente** da colônia. Dessa maneira cada usuário terá sempre disponível uma versão - uma Configuração Básica segundo Bersoff - do projeto, composto pela coleção de versões ou alternativas correntes de cada ICP do projeto.

6. - Meta-Sistemas de Gerenciamento de Dados

A armazenagem dos dados em uma base Base que utiliza o MRO pode ser logicamente feita utilizando-se vários arquivos. Além dos arquivos lógicos de objetos e de relacionamentos já mencionados, devem existir os vários arquivos de atributos, para armazenar atributos das várias características e os arquivos de históricos das colônias.

Deve ser notado que, ao contrário dos demais modelos de dados, cada arquivo lógico do MRO não armazena os dados de um mesmo item de especificação do esquema, e sim armazena dados que são estruturalmente semelhantes, mesmo que correspondam logicamente a ítens diferentes, e até de estruturas diferentes. Isso deve-se ao fato de que cada item de dado tem existência própria e individual, seja esse item um objeto, um relacionamento ou um atributo. Isso torna possível modelar o próprio MRO segundo um modelo de base de dados.

Considerando que cada objeto, relacionamento, colônia, atributo, etc, são por sua vez objetos de tipo respectivamente OBJETO, RELACIONAMENTO, COLÔNIA, ATRIBUTO, etc, pode-se descrever um Esquema genérico do MRO através de um **Meta-Esquema** de dados para o modelo. Isso pode ser feito considerando-se que cada arquivo lógico pode ser encarado como um Meta-Tipo de objeto. Existirão os seguintes Meta-Tipos de Objetos:

- Identificadores: correspondem a todos os identificadores de objetos da Base de Dados, sejam eles os identificadores principais (nomes) ou sinônimos;
- Objetos: correspondem a todas as informações de cada objeto da base de dados;
- Colônias: armazenam os conjuntos de objetos constritos em cada classe;
- Relacionamentos: correspondem a todos os relacionamentos existentes na base de dados;
- Atributos: armazenam o conjunto de todos os atributos de cada objeto ou relacionamento da base de dados;
- Atributos das várias características: existirá um Tipo de Atributo específico para cada Característica de Atributo (Propriedade, Regra, Posição, Comentário, etc).

As várias Características de Atributos são tratadas sempre da mesma maneira, permitindo que qualquer característica de atributo seja incorporada a um SGDE. Note-se que os atributos com característica de Identificadores, sejam eles os identificadores principais ou qualquer tipo de sinônimo, são tratados de forma diferen-

ciada dos demais atributos.

Uma conseqüência imediata da existência de um Meta-Esquema de Dados que permite definir o esquema de dados de uma dada aplicação, é que o Sistema de Gerenciamento de Dados para Engenharia permitirá a existência de um Meta-Sistema de Gerenciamento de Base de Dados (Meta-SGBD), o qual por sua vez manterá uma Meta-Base de Dados.

Um SGDE que abrange um Meta-SGBD é denominado neste trabalho **Meta-Sistema de Gerenciamento de Dados**, ou **Meta-SGD**. Num sistema assim, co-existem uma Meta-Base de Dados e uma Base de Dados. A Meta-Base de Dados armazena as informações **léxicas**, **sintáticas** e **semânticas** que constituem o Esquema de dados das informações manipuladas pela Base de Dados, ou seja, as informações que especificam como os dados se relacionam e qual o seu significado na Base de Dados.

O Meta-Sistema de Gerenciamento de Dados atua como um Gerenciador do Esquema de Dados armazenado na Meta-Base de Dados. Esse Esquema de Dados incorpora as informações de Sintaxe da Linguagem de Definição de Dados da Base de Dados, e nesse sentido é um Esquema de Dados semelhante aos usados nos demais modelos. No entanto, sendo definido pelo Meta-Esquema, o esquema de uma aplicação armazena também as informações sobre os próprios dados, e permite que essas informações sejam representadas em uma estrutura sintática idêntica à que é usada para representar os próprios dados. Isso permite que o Meta-Esquema do modelo MRO seja representado usando o próprio MRO.

A Meta-Base de Dados é usada pelo SGDE para manipular os dados da Base de Dados, e tendo a mesma estrutura de uma Base de Dados, a Meta-Base de Dados pode ser manipulada usando as mesmas operações que permitem a Manipulação de uma Base de Dados. Com esse enfoque, uma aplicação pode tanto manipular uma Base de Dados quanto manipular a definição dessa Base. Um ponto a ser analisado é o quanto uma alteração na Meta-Base de Dados afeta o conteúdo da Base de Dados. Isso será feito na seção 7.

Sem considerar-se efeitos de alterações da Meta-Base de Dados e as conseqüências semânticas de operações de inicialização de Bases de Dados e de Meta-Bases de Dados, pode-se usar a dualidade de estruturas e operações da Base e da Meta-Base de Dados para incorporar ambas em apenas uma **Base Unificada.** Nessa base são então armazenados o próprio esquema de dados e os dados da base. Ou seja, usando-se a terminologia de bases de dados relacionais, a mesma base armazena tanto a base extencional quanto a base intencional.

Isso pode ser feito aproveitando-se a disponibilidade permanente dos objetos da Colônia Global. Sempre que uma Base Unificada é criada, inicializa-se automaticamente a Colônia Global com um objeto de tipo controlado pelo sistema, que também deve ser único, o qual pode ser denominado de tipo ESQUEMA (esse nome não precisa estar entre os nomes de tipos de objetos definidos pelo usuário). Esse objeto passa a constringir a colônia "Esquema", a qual irá armazenar os objetos de tipos (também controlados pelo sistema - na realidade Meta-tipos) OBJETOS, IDENTIFICADORES, COLÔNIAS, RELACIONAMENTOS e ATRIBUTOS. A figura 2 corresponde ao DRO de um esboço de como essa colônia é esquematizada.

O usuário define e manipula o esquema de sua aplicação através da definição e manipulação de objetos dessa colônia. Atributos com características de Regra ou Procedimento são espe-

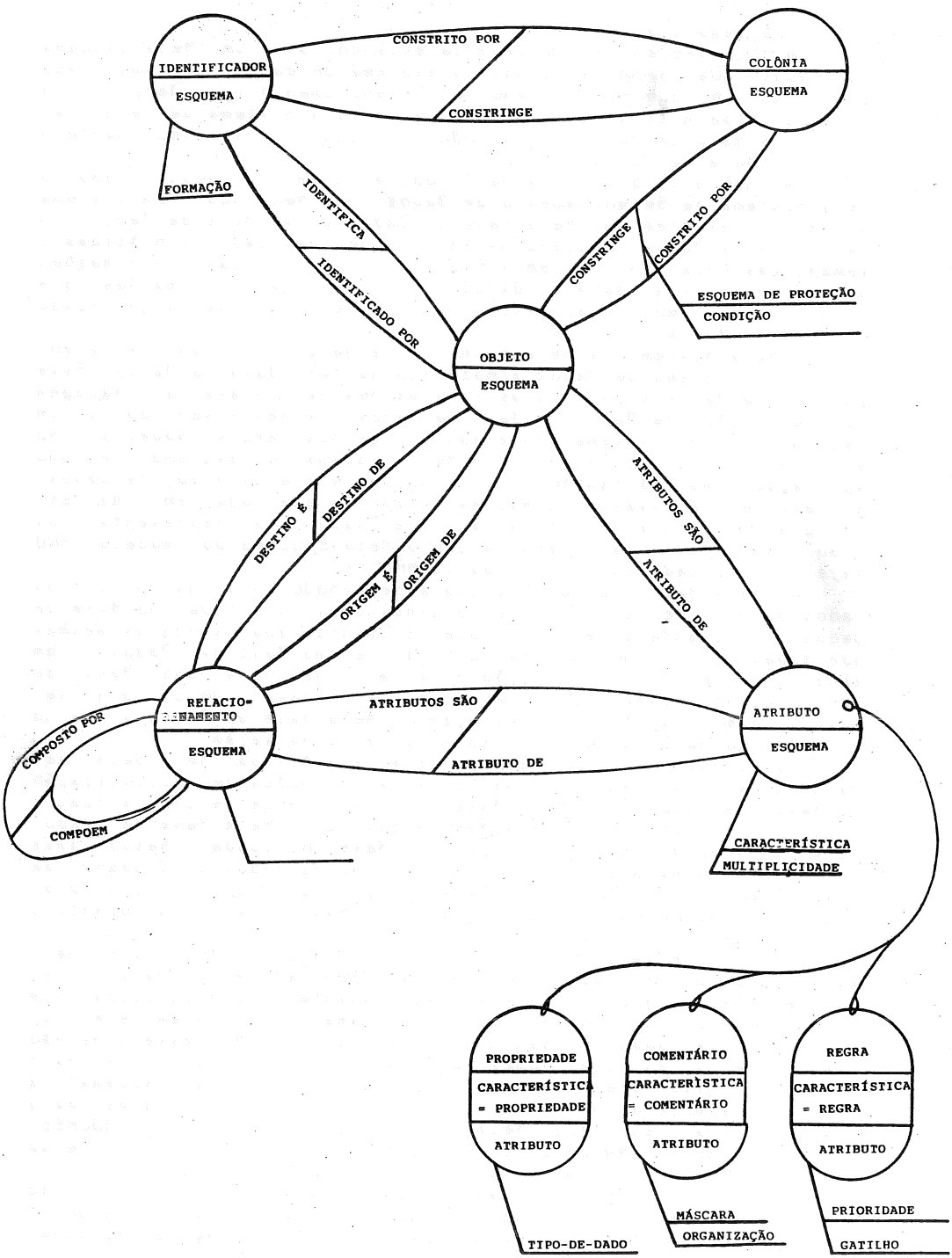


Figura 2 - Uma versão simplificada do esquema da Colônia βEsquema

cialmente adequados para essa colônia. No entanto, como o conteúdo da colônia "Esquema" corresponde ao esquema do restante da base, as operações semânticas da base de dados devem ser feitas usando-se os dados armazenados nessa colônia. Decorre daí que as operações que podem ser executadas na colônia "Esquema" são as mesmas que podem ser executadas em qualquer outra colônia, porém o sistema deve controlar essa colônia como um recurso próprio, uma vez que depende de seu conteúdo para reconhecer as operações solicitadas sobre as demais colônias. Portanto, o MRO não tem a Linguagem de Definição de Dados distinta da Linguagem de Manipulação de dados, sendo ambas as operações realizadas pela Linguagem de Manipulação de Dados. Quando esta se refere à Meta-Base de Dados, ela atua como uma Linguagem de Definição de Dados de um sistema convencional.

Neste trabalho, o termo Meta-Base de Dados corresponde ao esquema armazenado na Colônia "Esquema", ao passo que os dados dos usuários são armazenados nas demais colônias, e chamados simplesmente de Base de Dados. Ao conjunto da base de dados mais a colônia "Esquema" (a Meta-Base de Dados) denomina-se Base Unificada.

7. - O Controle de Versões do Esquema de Dados

Sendo armazenado em uma Colônia de Objetos, o esquema de uma aplicação está sujeito às mesmas regras de operações que governam as demais colônias. Isso vale também para os critérios de instanciação de colônias, o que significa que podem existir várias versões de esquemas. No entanto, a colônia "Esquema" é reconnecida pelo SGDE e tratada de maneira especial quando as operações se referem à Meta-Base de Dados. Portanto, para que um esquema de dados possa ser usado como tal, é necessário que a colônia "Esquema" esteja protegida contra escrita, ou seja, somente podem ser usadas variantes que sejam versões.

Assim, para que uma base de dados possa ser usada, ela deve ser inicializada, e sua Colônia "Global" e Colônia "Esquema" criadas. A Colônia "Global" passará a ter inicialmente um único objeto que constringe a Colônia "Esquema", e o esquema da aplicação deve ser definido. Tendo completado o esquema, ele deve ser "Congelado", e a partir daí poderá ser usado para que a base de dados propriamente dita seja usada. A Base de Dados estará então inicializada. A figura 3 mostra como estará uma base de dados recém-inicializada. Note-se que os objetos tipo βESQUEMA e βUSUÁRIO são controlados pelo sistema (O β do nome indica isso), e portanto não precisam ser explicitamente colocados pelo usuário. Os objetos constritos pelo objeto βESQUEMA são os Tipos de Objetos, de Relacionamentos, etc. que definem a primeira versão do esquema da aplicação. Objetos do tipo βUSUÁRIO são usados para o controle de acesso à base de dados, e podem existir quantos forem necessários. Em uma base de dados existe sempre exatamente um objeto do tipo βESQUEMA, do qual podem existir muitas variantes.

Se alguma alteração for necessária, a colônia "Esquema" deve ser instanciada, e a nova alternativa para o esquema deve ser fornecida ao sistema. No entanto, para que possa ser usada para continuar o acesso à base de dados, esta terá que ser por sua vez congelada. Numa situação assim, existirão duas versões para o

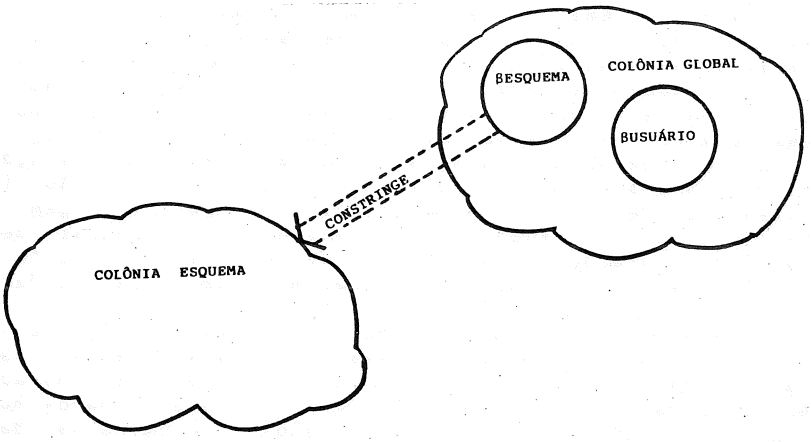


Figura 3 - DRI de uma base de dados recém-inicializada

esquema de uma base de dados. Será usada para um determinado projetista a versão que corresponde à sua variante corrente. Podem existir várias alternativas de um esquema, porém para que possam ser usadas como o Esquema da base de dados elas terão necessáriamente que ser congeladas, transformando-se em versões.

Note-se que dessa maneira, alterações no esquema de uma aplicação não causam impactos na estrutura da base de dados, uma vez que as informações da base estarão sempre associadas à versão do esquema que a criou, às quais sempre estarão disponíveis. Quando um usuário acessar as informações no entanto, ele terá acesso apenas às informações que ele tem esquematizadas em sua versão corrente da colônia "Esquema".

Feita essa caracterização, pode-se analisar agora como cada tipo de alteração do esquema pode ser tratado pelo MRO. Alterações que incluam novos tipos de objetos ou de associações (relacionamentos ou atributos) entre tipos de objetos já existentes não causam problemas, pois apenas irá ocorrer que na parte da base criada anteriormente à alteração do esquema não existem objetos e associações dos tipos novos.

Alterações que retirem tipos de objetos ou de associações do esquema poderão causar a existência de dados na Base de Dados que não estariam definidos no novo esquema. Nesse caso, três alternativas podem ser escolhidas pelo projetista de um SGBD apoiado no MRO: pode-se ignorar dados que não estão definidos na versão corrente do esquema; pode-se indicar que o dado somente pode ser acessado quando o esquema corrente é o mesmo usado para a definição daquele dado; ou finalmente, pode haver a possibilidade de que o dado possa ser acessado com a indicação que as informações da Meta-Base de dados devem sempre vir acompanhadas da indicação da versão do esquema a que elas se referem.

8. - Conclusões

Uma consideração deve ser feita aqui quanto aos nomes dados aos conceitos do MRO. A partir da popularização da linguagem de programação SmallTalk-80 [REN 82] desenvolvida pela Xerox Parc no inicio dos anos 80, o conceito de objetos desenvolveu-se paralelamente também como um paradigma de programação. Apesar de serem conceitos semelhantes, como um paradigma de programação o conceito de objeto apresenta características diferentes, e as vezes conflitantes, em relação às características que lhe são atribuídas como um modelo de dados para o desenvolvimento de bases de dados orientadas a objetos.

O Paradigma de Programação orientada à objetos designa os vários objetos de um mesmo tipo como uma "Classe de Objetos", e cada ocorrência de um objeto como uma "Instância de Objeto". As associações entre objetos são feitas através das propriedades de cada classe de objetos, e raramente as associações são explicitadas. No MRO, a associação entre objetos é sempre feita explicitamente, e dessa maneira, uma uniformidade dos termos usados para designar objetos e relacionamentos deve ser seguida. Por isso, o termo "Objeto" é usado para designar uma instância de um objeto, da mesma maneira que o termo "Relacionamento" é usado para designar um relacionamento único entre dois dados objetos. Analogamente, o termo "Tipo de Objeto" é usado para designar as características importantes que devem ser reconhecidas em objetos, as quais permitem identificar determinado objeto como sendo daquele determinado tipo; e o termo "Tipo de Relacionamento" é usado para designar as características importantes que devem ser reconhecidas em relacionamentos, aquelas que permitam identificar cada relacionamento como sendo daquele tipo.

Um conceito importante no MRo corresponde às Colônias de Objetos. Esse conceito permite não só o acompanhamento do foco de interesse de um projetista na Base de Dados, mas também a instanciação e controle de versões e alternativas dentro da Base de Dados.

Também importante é a fundamentação do modelo na Teoria dos Grafos porque, permitindo a homogeneidade da representação estrutural dos dados, propicia a construção do SGD como um Meta-Sistema. A fundamentação do MRO na Teoria dos Grafos e o conceito de Colônias de Objetos se aliam para criar o suporte onde a manutenção de um esquema de dados pode ser feita dinamicamente, em conjunto com a operação normal do SGD sem impacto sobre os dados já armazenados.

Por fim é importante ressaltar que o objetivo principal que norteou o desenvolvimento do MRO é o seu emprego em Ambientes de Desenvolvimento de Software. Duas implementações de SGDEs para esse fim apoiadas nos conceitos apresentados estão em fase adiantada, uma no ICMSC-USP em São Carlos, feita para o Sistema SACDS [TRA_82], e outra no CTI em Campinas para o Projeto SIPS [TSU_85].

REFERÊNCIAS BIBLIOGRÁFICAS

[ABI_87] Abiteboul, S.; Hull, R. - "IFO: A Formal Semantic Database Model", ACM TODS, Vol. 12, Nro. 4, December 1987, pp. 525-565.

[BER_84] Bersoff, E. - "Elements of Software Configuration Management", IEEE Trans. Software Engineering, Vol. SE-10, Nro. 1, January 84, pp. 79-87.

[BIC_86] Bic, L.; Gilbert, J. P. - "Learning from AI: New Trends in Database Technology", IEEE Computer, Vol. 19, Nro. 1, January 86, pp. 44-54.

[EAR_85] Earl, A. N.; Whittingtom, R. P. - "Capturing the Semantics of an IPSE Database" Data Processing, Vol.27, Nro. 9, November 1985, pp. 33-43.

[FAR_85] Farmer, D. B.; King, R.; Myers, D. A. - "The Semantics Database Constructor" IEEE Trans. Software Engineering, Vol. SE-11 Nro. 7, July 85, pp. 583-591.

[HAM_81] Hammmer, M.; McLeod, D. - "Database Description with SDM: A Semantic Database Model", ACM Trans. Database Systems, Vol 6, Nro. 3, September 81, pp. 351-386.

[HAR_85] Hartzband, D. J.; Maryanski, F. J. - "Enhancing Knowledge Representation in Engineering Databases" IEEE Computer, Vol. 18, Nro. 9, Septenber 85, pp. 39-48.

[KAT_84] Katz, R. H.; Lehman, T. J. - "Database Support for Versions and Alternatives of Large Design Files", IEEE Trans. Computers, Vol. SE-10, Nro. 2, march 84, pp. 191-200.

[KAT_86] Katz, R. H.; Chang, E.; Bhateja, R. - "Version Modeling Concepts for Computer-Aided Design Databases", SIGMOD 86 Conference, in ACM SIGPLAN Vol. 21, Nro. 12, pp. 379-386.

[KEL_87] Kelter, U. - "Concurrency Control for Design Objects With Versions in Cad Databases", Information Systems, Vol. 12, Nro. 2 pp. 137-143.

[MAT_81] Matsuka, H.; Uno, S.; Sibuya, M. - "Specific Requirements in Engineering Data Base", Lecture Notes in Computing Science, Vol. 133, New York: Springer-Verlag, 1982, pp. 345-356.

[NAV_86] Navathe, S. et al. "Integrating User Views in Database Design", IEEE Computer, Vol. 19, Nro. 1, January 86, pp. 50-62.

[NG*_81] Ng, A. P. - "Further analysis of the Entity-Relationship approach to Database Design", IEEE Trans. Software Engineering Vol. SE-7 Nro. 1, January 81, pp. 85-99.

[PAR_85] Parent, C.; Spaccapietra, S. - "An Algebra for a General Entity-Relationship Model", IEEE Trans. Software Engineering, Vol. SE-11 Nro. 7, July 85, pp. 634-643.

[REN_82] Rentsh, T. - "Object-Oriented Programming", ACM SIGPLAN Notices, Vol. 17, Nro. 9, September 1982, pp. 76-87.

[TRA_82] Traina Jr., C. - "Sistema de Apoio por Computador 1 à Documentação de Sistemas", Dissertação apresentada ao ICMSC-USP para obtenção do Título de Mestre, Dezembro 82.

[TRA_86] Traina Jr., C. - "Máquina e Modelo de Dados Dedicados para Aplicações de Engenharia", Tese apresentada ao IFQSC-USP para obtenção do Título de Doutor, Dezembro 86.

[TRA_88] Traina Jr., C.; Slaets, J. F. W. - "Um Modelo de Representação de Objetos", in Anais do 3º Simpósio Brasileiro de Banco de Dados, Recife, Março de 1988, pp. 227-242.

[TSU_85] Tsukumo, A. N. et alli - "Um Sistema Expansível para Produção de Software", in Anais do 2º Congresso Nacional de Automação Industrial, São Paulo, Novembro 85, pp. 379-386.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

REVER: Uma Interface Semântica para Bancos de Dados Relacionais

Mônica de Lima Nogueira*
Claudia Bauzer Medeiros†
DCC - IMECC - UNICAMP - CP 6065
CEP 13.081 Campinas, SP - BRASIL
Tel (0192) 391301, ramal 3442

Sumário

Este artigo descreve as características de uma interface E-R para um banco de dados relacional - o sistema REVER. O usuário do sistema pode utilizá-lo tanto para fazer o projeto do banco de dados quanto para consultas e atualizações. A visão do usuário é a de criação de elementos de um diagrama E-R estendido, e de manipulação de suas ocorrências. O sistema se encarrega de mapear as solicitações do usuário para operações sobre o esquema e instâncias do banco de dados relacional correspondente.

Abstract

This paper describes the features of an E-R interface to a relacional database - the REVER system. The system allows the user to design the database using only the E-R specification, as well as to query and update it. The user's view is always that of an E-R (extended) diagram, in the design and in the operational phases. The system maps the user's requests on the E-R scheme and instances to operations on the corresponding relational database.

*Tecnóloga em Eletrônica (UTAM 1982) e Eng. Elétrica/Eletrônica (FUA 1983). Mestranda em Ciência da Computação (UNICAMP 1988). Área de interesse: Banco de Dados.

†Eng. Eletrônica (PUC/RJ 1976), Mestre em Informática (PUC/RJ 1979), PhD. in Computer Science (Waterloo 1985). Áreas de interesse: Banco de Dados, Eng. Software.

1 Introdução

Até o surgimento da proposta de [CHEN76] para modelagem do mundo real em termos de entidades, relacionamentos e atributos, os bancos de dados existentes não eram capazes de expressar em correpondência direta os objetos de uma aplicação. Na realidade, o ambiente era moldado para um modelo de estrutura mais ou menos rígida.

Algumas das vantagens do Modelo Entidade-Relacionamento (E-R) segundo [CHUN83] são: permitir capturar e preservar algumas das importantes informações semânticas do mundo real, possibilitar linguagens de computador baseadas em E-R mais naturais para usuários não-técnicos. A facilidade de tradução entre o modelo E-R e os modelos de dados convencionais indicam que o modelo deveria ser padronizado para modelagem de esquema conceitual nas arquiteturas ANSI/SPARC e para manipulação da tradução de dados em ambientes de banco de dados heterogêneos.

Desde a proposta de Chen [CHEN76] inúmeras extensões têm sido defendidas ([ELMA85], [FELD86], [PARE86], [SCHE80], [SMIT77], [TEOR86], [WAGN87]) para o Modelo Entidade-Relacionamento. No entanto, não existe um consenso a respeito da Linguagem de Manipulação de Dados (DML) mais adequada para este modelo, que englobe todas essas extensões. Outro problema apontado é a fraqueza de definição semântica e a falta de poder para expressar propriedades temporais.

Muitos dos modelos propostos são distintos entre si nas definições e uso dos conceitos de relacionamento e atributo. Em [CHEN83b] é apresentado um quadro para classificar as extensões E-R existentes e é discutido como estas podem ser traduzidas de uma para a outra.

[TEIC83] apresenta uma retrospectiva dos trabalhos baseados no E-R-Approach (ERA) e resultados obtidos na sua aplicação. O enfoque do modelo ERA surge como o mais geral, capaz de fornecer uma visão mais natural dos sistemas e de preencher a lacuna entre análise de sistemas, modelagem de dados e análise funcional.

Outro modelo com o mesmo tipo de enfoque é EAS e a linguagem EAS-E, sistemas experimentais de desenvolvimento de aplicações baseados no formalismo de entidades, atributos, conjuntos e eventos. Segundo [MARK83] as visões E-R e EAS são similares no aspecto que possibilitam ao usuário modelar o mundo em vez de obrigá-lo a especificar como o computador deve representar este mundo e ambas fornecem um conjunto simples mas geral de conceitos de modelagem. O poder de cada visão é também sua fraqueza. Na visão EAS é necessário que o usuário declare antecipadamente os relacionamentos, o que leva à produção de um programa que executa mais eficientemente. Ainda comparado ao modelo E-R, o sistema EAS normalmente envolve mais problemas quando são efetuadas mudanças nas definições do banco de dados.

Há, além disso, o problema de que o modelo E-R é um modelo semântico, que deve depois ser analisado por gerentes de banco de dados para especificar o banco de dados que o represente. Assim, embora ao leigo seja facilitado expressar suas especificações no modelo semântico, a implementação fica a cargo de especialistas. Além disto, uma vez gerado o banco de dados, o usuário não mais dispõe das abstrações iniciais.

Dentre trabalhos que abordam este problema situam-se [SHAV81], que discute uma ferramenta para modelagem conceitual baseada em elementos E-R; o Projeto GAMBIT [BRAE85], que dentre outros pontos facilita ao usuário a definição de restrições de integridade; e o Sistema CHRIS [FURT87], uma implementação em PROLOG de uma interface E-R para o banco de dados relacional.

O presente trabalho enfoca alguns dos aspectos mencionados nos parágrafos anteriores, tentando solucionar o problema através de uma interface E-R para manipulação de um banco de dados relacional. REVER (RElacional Via Entidade-Relacionamento) é a interface proposta, que amplia

o universo de usuários que virão a usufruir das qualidades do Banco de Dados Relacional sem que para isso lhes seja impingida a necessidade de se especializarem nessa área. Como resultado, o usuário poderá desfrutar das vantagens da modelagem semântica E-R aliadas ao rigor teórico existente para o Modelo Relacional.

Pressupõe-se que o usuário da interface já haja procedido à análise do ambiente e suas necessidades, havendo elaborado a especificação do sistema, ou esquema funcional [ROUS84]. Uma vez estabelecido um modelo funcional, ou seja, quando identificadas as principais atividades ou funções e as associações existentes entre elas, pode-se gerar um esquema conceitual E-R usando a interface.

Um dos objetivos deste trabalho é integrar técnicas de projeto de banco de dados e de projeto de programas, dado que um sistema de BD não é só um esquema (componente estrutural) ou só um conjunto de transações (componente procedural), mas sim um esquema com transações sendo aplicadas a este esquema [ROUS84].

Elmasri [ELMA83], em sua proposta de uma linguagem (GORDAS) para o modelo E-R, classifica os diferentes tipos de interfaces em: procedurais, dedicadas a profissionais para definição dos programas de aplicação que serão usados pelos usuários; não-procedurais, para interação de usuários técnicos e semi-técnicos com o DBMS a partir de um terminal; e amigáveis, que inclui menus para orientar usuários não-técnicos e uso de linguagens naturais. O sistema REVER se enquadra nesta última categoria. Ele está sendo implementado como parte de um projeto de tese de mestrado no DCC/UNICAMP [NOGU88].

Este artigo possui 5 seções adicionais. A primeira delas trata dos conceitos e primitivas do modelo E-R estendido utilizado e seu mapeamento para o modelo Relacional; a seguir tem-se a especificação da interface, sendo discutidas as linguagens de definição dos objetos do modelo e de consulta/atualização. A seção 4 trata de alguns detalhes da implementação e a seção 5 discute alguns problemas de atualização. A seção final apresenta problemas e conclusões.

2 Conceitos e Nomenclatura

2.1 Modelo Entidade-Relacionamento

Esta seção pressupõe que o leitor já conhece as características básicas do modelo E-R. Serão apenas ressaltados alguns termos utilizados neste artigo. Para maiores detalhes, sugere-se a leitura de [CHEN76] e de [MAIE85].

Uma *"entidade"* representa um conjunto de ocorrências de informações de uma mesma espécie, com características: *"atributos"*. As diferentes ocorrências de uma entidade são distinguidas através de um ou mais atributos identificadores (*"chave"*).

Os atributos podem ser: 1. compostos - formados por mais de um atributo, identificadores ou não (descritores); 2. multivalorados - quando para um único atributo identificador ou chave correspondem mais de um atributo descritor ou não-chave.

Um *"relacionamento"* é a representação das associações que ligam uma (auto-relacionamento) ou mais entidades, podendo ter atributos próprios.

Um dos aspectos mais importantes que diferenciam o modelo original de Chen de suas extensões é a definição dos relacionamentos. Um exemplo são os conceitos de 'grau'(unário,binário.ternário), 'conectividade' (1:1, 1;n, m:n) e 'classe-de-membro' (mandatório ou opcional) dos relacionamentos conforme adotados por [TEOR86].

Um relacionamento (entre entidades A e B) é dito *"total"* de B em relação a A quando a

existência de cada ocorrência da entidade B *"depende"* da existência de uma ocorrência correspondente da entidade A. Essa entidade com dependência de existência é chamada de entidade *"fraca"* no modelo original de Chen. Na extensão E-R adotada, não se identifica uma entidade fraca, mas sim o relacionamento total, indicado graficamente através de um círculo escuro conforme exemplifica a figura 1. A modificação de representação (em função do relacionamento) é particularmente interessante, pois marca o relacionamento que estabelece a relação de dependência de existência, evitando ambiguidades quando essa mesma entidade participa de outros relacionamentos onde tal dependência não existe.

Caso a existência de uma entidade seja completamente independente da existência de outra entidade, o relacionamento que as interliga é dito *"parcial"*.

Um relacionamento total é representado por uma *"dependência de inclusão"* do modelo relacional e implica que a inclusão de uma tupla em uma relação depende da existência de uma tupla correspondente em outra relação (aquela à qual está ligada através do relacionamento total).

2.2 Modelo E-R Estendido

Das inúmeras contribuições existentes para o modelo E-R, a de Wagner e Medeiros reuniu um grupo de proposições cujo objetivo principal é ampliar a riqueza semântica do modelo através do aumento dos níveis de abstração.

O modelo E-R estendido aqui utilizado é o apresentado em [WAGN87], composto de blocos adicionais mais complexos baseados nas primitivas do modelo original, aqui chamados objetos complexos: 'Agregação', 'Generalização' e 'Hierarquia de Tipos'. A figura 2 mostra a simbologia usada para tais elementos conforme foi proposta por Navathe e Cheng [NAVA83]. Maiores detalhes e exemplos da extensão adotada podem ser encontrados em [WAGN87].

Entende-se *"agregação"* como o encapsulamento de uma ou mais entidades e um relacionamento que as interliga (figura 3). Na realidade, as ocorrências do agregado correspondem às associações expressas pelo relacionamento que encapsula.

A entidade complexa *"generalização"* pode ser caracterizada por uma árvore de 2 níveis onde a raiz engloba os argumentos comuns a um conjunto de elementos (as folhas). Cada elemento-folha tem como atributos aqueles herdados da raiz, bem como atributos que o distinguem das demais folhas (figura 4). A cada ocorrência de um elemento-raiz corresponde uma ocorrência de alguma folha; as folhas correspondem a conjuntos mutuamente exclusivos. Diz-se que a raiz é uma *"generalização"* das folhas e que estas são uma *"especialização"* da raiz. Esta caracterização é recursiva, ou seja, cada folha pode ser por sua vez a raiz de uma nova árvore: o grafo correspondente adquire assim as características de uma árvore de nós heterogêneos.

Esta definição de entidades de uma generalização é semelhante àquela aplicada por [TEOR86] e [BRAE85]. Em [WELD80] a generalização é representada por um objeto geral com as características comuns de uma coleção de objetos (mais específicos e mutuamente exclusivos entre si). A existência de categorias não exclusivas implica em diferentes 'agrupamentos' de generalizações, sendo que em cada agrupamento as categorias são distintas.

A generalização tem uma variante denominada *"hierarquia de tipos"*. Se em generalização ocorre exclusão mútua entre os elementos-filho participantes, o mesmo não é necessariamente verdade em *"hierarquia de tipos"*, daqui por diante referenciada simplesmente como *"hierarquia"*, figura 5. O trabalho de [FURT87] implementa o conceito de hierarquia de tipos que também é conhecida por 'hierarquia is-a'. Em [TEOR86] temos o mesmo conceito indicado como 'hierarquia de subconjuntos' onde as entidades-folha não possuem exclusão-mútua ocorrendo sobreposição de papéis sem que

haja qualquer prejuízo da estrutura; os únicos atributos repetidos no caso são aqueles identificadores das entidades, ou chaves.

A princípio nada impede a existência de árvores cujos nós são relacionamentos (vide Wagner e Medeiros [WAGN87]), ou mesmo árvores heterogêneas (onde os nós correspondem a elementos de natureza diversa, p.ex., folhas compostas por hierarquias e agregações).

2.3 Modelo Relacional

Simplificadamente, um banco de dados relacional é composto por um conjunto de relações ou tabelas. Um *"esquema"* R(I) é formado pelo conjunto de (nomes de) atributos {A(1)...A(N)} onde cada um destes é definido sobre um domínio, DOM{A(I)}. Um *"esquema relacional"* RR é composto de um conjunto de esquemas R(I). Por analogia a esquema de relação este artigo denomina esquema E-R à descrição dos objetos no projeto conceitual no modelo E-R estendido, onde se especificam os objetos do projeto e suas interligações, sem preocupação com ocorrências.

Uma *"relação"* R(I) é uma ocorrência de um esquema R(I). Diz-se também que uma relação é uma tabela formada por um conjunto de tuplas T. Uma *"tupla"* T(I) é um conjunto de valores ordenados segundo os atributos da relação, com significação válida dentro do domínio dos atributos correspondentes. De novo, por analogia, o artigo referencia ocorrências de um objeto E-R (por exemplo, dada uma entidade MÉDICO, cada ocorrência é o conjunto de dados relativo a um médico). Um *"banco de dados relacional"* é um conjunto de relações {R(I)}. Um estado do banco de dados é uma ocorrência do esquema relacional RR, sobre o qual está definido.

Supõe-se que o leitor conheça as definições de *"chave"* e *"dependências funcionais e multivaloradas e 1NF"*.

Um relacionamento total é representado por uma *"dependência de inclusão"* do modelo relacional e implica que a inclusão de uma tupla em uma relação depende da existência de uma tupla correspondente em outra relação (aquela à qual está ligada através do relacionamento total).

2.4 Mapeamento E-R ↔ Relacional

A interface REVER pressupõe regras de mapeamento entre o modelo E-R estendido adotado e o modelo relacional. A interface tem dois níveis de mapeamento: para geração do esquema e para consultas/atualizações de relações. No primeiro caso, o usuário especifica seu projeto semântico em termos de objetos E-R e a interface gera o esquema relacional correspondente segundo regras pré-definidas de mapeamento. No segundo caso, o usuário consulta/atualiza objetos E-R, o que é traduzido pela interface em consultas e atualizações às relações base. Há dois aspectos a se considerar: 1. encontrar a correspondência entre os dois modelos; e 2. sintetizar algumas propriedades dos dois modelos em uma forma compreensível [CHEN83a].

A cada atributo do modelo E-R corresponderá um único atributo relacional. Seguindo o proposto por [TEOR86] os atributos multivalorados devem ser transformados em entidades ainda na fase de projeto E-R. Portanto, não há necessidade de mapeá-los para o modelo relacional.

A fase de mapeamento englobará a transformação de entidades, relacionamentos e dos objetos complexos: generalização, hierarquia e agregação. A geração do esquema relacional segue as seguintes regras, não pressupondo otimização ou minimização de esquema.

- Cada entidade simples é convertida em uma relação com esquema igual ao da entidade original, identificando atributos chave e não-chave.

- Cada relacionamento é transformado em um esquema composto das chaves das entidades que liga acrescido dos atributos específicos do relacionamento. Esse mapeamento é adotado por que ressalta as seguintes vantagens: 1. o uso de uma relação separada para relacionamentos permite armazenar relacionamentos sem modificar outras relações do BD relacional; 2. esta implementação uniforme não precisa ser modificada para relacionamentos de cardinalidades diferentes (1:1, 1:n, m:n); e 3. possibilita de forma simples a criação e remoção de relacionamentos sem modificação das tabelas das entidades.

- As entidades complexas: Generalização e Hierarquia, seguem o padrão, adotado por [WELD80]:

  - o nó 'pai' é transformado em um esquema com a chave e atributos (comuns a todas as entidades 'filho') desse nó;
  - os nós 'filhos' são transformados em esquemas contendo cada um a chave da entidade 'pai' mais os atributos específicos de cada filho.

- A entidade complexa Agregação é convertida obedecendo os mesmos critérios usados para relacionamentos.

O modelo semântico proposto considera dependências de chave e de existência. Estas dependências são mapeadas, no modelo relacional, para dependências funcionais (dentro de uma relação) e dependências de inclusão (correspondentes a relacionamentos totais e manutenção de árvores com heranças de propriedades).

A herança de propriedades em uma árvore será garantida pela inclusão da chave do pai entre os atributos dos filhos, sendo mantida portanto através dos mecanismos de integridade associados a chaves. (A chave de cada filho englobará obrigatoriamente a chave do pai).

A manutenção de dependências de inclusão e de propriedades como exclusão mútua (de ocorrências de filhos em uma generalização) apresentam problemas teóricos complexos. A análise teórica destes problemas escapa ao âmbito deste trabalho. A seção 2.5 apresenta algumas simplificações do trabalho que permitem eliminar alguns destes problemas.

2.5 Simplificações para o Presente Trabalho

A interface considera apenas relacionamentos binários. As relações criadas a partir da especificação E-R estarão, pelo menos, na Primeira Forma Normal. Atributos chave compostos não são considerados nesta versão do REVER.

Este trabalho se restringirá ao manuseio de árvores homogêneas (compostas do mesmo tipo de nó). Árvores de relacionamentos serão igualmente ignoradas, embora se aceite árvores de agregados.

Para minimizar os problemas advindos de manutenção de integridade para dependência de inclusão, temos:

- Uma dada entidade só pode estar associada a no máximo um relacionamento total (ou seja, só pode participar de no máximo uma dependência de inclusão), figura 6.

- Agregados que participem de árvores não podem englobar relacionamentos totais (a participação em uma hierarquia já implica uma dependência de inclusão), figura 7.

- Agregados que englobem relacionamentos totais não podem por sua vez estar associados a outros elementos através de relacionamentos totais, figura 8.

- Folhas de uma árvore não podem estar associadas por relacionamentos totais, figura 9.
- Nenhum elemento pode pertencer a mais de uma árvore, figura 10.

3 Especificação da Interface

A interface tem dois tipos de operação: operações sobre o esquema e sobre os dados. Em ambos os casos, são permitidas consultas e atualizações. Saliente-se que é possível modificar o esquema mesmo após criado o banco de dados. No estágio atual do REVER, as modificações permitidas são ainda limitadas (por exemplo, não é possível mudar o tipo de um atributo uma vez criada a relação).

As operações de geração e consulta permitidas sobre o esquema do modelo E-R estendido aqui utilizado, nos leva à discussão da linguagem de consulta apropriada.

Inúmeras linguagens de consulta E-R foram propostas sobre uma base de dados relacional (por ex. [MALH86]). Todas tentam aliar a riqueza semântica do modelo E-R ao formalismo relacional.

Chen [CHEN83b], por exemplo, baseia-se no cálculo relacional para mostrar que linguagens de consulta para o modelo E-R podem ser 'completas', que é um requisito fundamental das linguagens relacionais. Fornece duas definições formais: linguagens E-R completas e completas simplificadas.

Segundo [DATE77], uma linguagem completa *"significa para o usuário que, se uma informação desejada está num banco de dados, então ela pode ser recuperada através de uma simples consulta"*. As linguagens completas, entretanto, não possuem função de agregação e capacidade aritmética implícitas, que são propriedades desejadas.

A interface REVER se comunica com o usuário através de menus, de forma interativa. Quaisquer operações (consultas ou atualizações) sobre dados são especificadas pelo usuário em termos do projeto E-R. Desta forma, o sistema precisa primeiro verificar que elemento(s) do esquema relacional corresponde(m) a um objeto E-R para depois efetuar a operação desejada sobre as relações correspondentes (ocorrências do esquema E-R) e mostrar o resultado ao usuário.

Existem vários problemas no que diz respeito a operação sobre ocorrências de objetos E-R. Por exemplo, uma consulta a um objeto complexo E-R pode corresponder a consultas a várias relações. Optou-se, neste caso, por informar ao usuário sobre o esquema consultado para que ele possa optar sobre que dados deseja realmente ver. Assim, por exemplo, se consultada uma generalização, o usuário recebe como resposta quais os nós componentes, podendo então escolher de qual nó examinar as ocorrências.

Da mesma forma, atualizações sobre um objeto podem ser propagadas a outros (razão pela qual se optou pelas restrições mostadas na seção anterior). A seção 5 discute alguns problemas neste sentido. Uma maior discussão sobre o tópico e soluções apresentadas pode ser encontrada em [NOGU88], escapando ao âmbito deste artigo por sua complexidade.

As operações de atualização sobre o esquema E-R são, na verdade, de dois tipos: operações permitidas sobre um esquema vazio e operações permitidas quando o banco de dados já contém dados. Muitas atualizações permitidas sobre o esquema de um banco de dados vazio deixam de sê-lo a partir do momento em que dados são carregados. Isto parte do pressuposto que a definição do esquema E-R pode sofrer alteração (por exemplo, atributos de uma entidade, restrições de integridade ou interrelacionamentos entre objetos), enquanto que modificações de esquema se tornam mais difíceis após a geração de dados. Assim, as únicas alterações de esquema permitidas após a carga do banco de dados são as que implicam geração ou remoção de relações inteiras.

A formação do esquema relacional é processada como um conjunto de listas encadeadas. Apenas quando o usuário se considera satisfeito com o projeto do esquema E-R é que são gerados os comandos para criação das relações correspondentes.

4 Implementação da Definição do Esquema

O esquema relacional é armazenado em um arquivo com 4 registros diferentes, ordenados via um ÍNDICE. Cada conjunto de registros é identificado por um campo comum chamado TIPO. Os formatos de registro se encontram a seguir, onde IND é um identificador para o índice, T é um identificador de tipo, CAD(ATR) é um apontador para uma cadeia de enumeração de atributos do objeto, CAD(REL) é um apontador para uma cadeia que armazenará todos os índices dos relacionamentos aos quais o objeto está ligado e CAD(COMPLEX) é um apontador para uma cadeia que conterá todos os índices dos objetos complexos dos quais o objeto participa.

Ind	E	Nome	Chave	Cad(atr)	Cad(rel)	Cad(complex)

O registro referente a relacionamentos abaixo engloba também a indicação se é um relacionamento TOTAL/PARCIAL, a DIREÇÃO para relacionamentos totais e os índices IE1 e IE2 dos objetos que liga.

Ind	R	Nome	Chave	T/P	Dir	IE1	IE2	Cad(atr)	Cad(complex)

O registro de agregações abaixo indica o relacionamento que identifica a agregação - IREL.

Ind	AG	Nome	IRel	Cad(rel)	Cad(complex)

As estruturas de árvores tem registros semelhantes para armazenar, nó a nó, os seguintes campos: NOME da árvore (generalização/hierarquia), IND(PAI) - índice do nó-pai; CAD(IND-FILHOS) - um apontador para uma cadeia que engloba os índices dos nós-filho do nó em questão.

Ind	T	Nome	Ind(pai)	Cad(Ind_filhos)

Com as informações constantes desses registros é possível navegar em todos os sentidos para a recuperação/alteração de dados. Optou-se por dar às relações geradas os mesmos nomes dos elementos E-R.

5 Problemas de Atualização

São consideradas apenas atualizações de dados do tipo inserção, remoção e modificação de valores não-chave. Qualquer pedido de atualização é suposto como sendo requisitado em cima dos seguintes elementos: a) entidades simples; b) relacionamentos; c) agregados; d) nó de uma árvore (que pode ser de um dos tipos anteriores). De agora em diante, qualquer atualização em um elemento é entendida como mapeada para a relação correspondente.

O caminho de propagação de atualizações é obtido através de navegação dentro da estrutura de dados do esquema, acompanhado, quando necessário, de consultas ao usuário sobre opções a serem tomadas.

Para melhor entendimento do que se segue, serão definidos os seguintes termos:

- Caminho em uma árvore - mesmo conceito utilizado em estruturas de dados do tipo árvore.
- Caminho ancestral de um nó - caminho que sai da raiz de uma árvore até atingir o nó em questão.
- Irmãos de um nó - dado o ancestral (pai) do nó em uma árvore, são os outros filhos daquele pai, no mesmo nível.

Para inserção em nós de árvores, é necessário ao usuário indicar valores para todos os atributos no caminho na árvore ao qual o nó pertence.

Para remoção de nó em generalizações, serão eliminados todos os elementos a ele relacionados no caminho na árvore; para remoção em hierarquias, serão eliminados todos os elementos a ele relacionados no caminho da raiz até o nó em questão, se o elemento a remover não tiver correspondente em seus irmãos na árvore; e serão eliminados todos os elementos da sub-árvore da qual ele é raiz.

Modificações de valores de atributos não acarretam problemas.

Tendo em vista estas definições, e considerando-se as atualizações permitidas, observa-se o seguinte:

- modificações não exigem propagação, restringindo-se ao elemento considerado (pois não se permite mudar conteúdo de chaves).
- as inserções e remoções podem requerer propagação, em casos específicos. A discussão destes casos se encontra em [NOGU88].

Casos de propagação de atualizações:

a) Quando há dependência de inclusão, figura 11:

- se inserir em B e/ou R → inserir em A,
- se excluir de A e/ou R → excluir de B;

b) Em entidades, figura 12:

- se excluir de A/B → excluir de R,
- se incluir em R → incluir em A e B;

c) Em agregações:

- obedece as regras de (a) e (b);

d) Em hierarquias de tipo, figura 13:

- se inserir em EL:
  - Propagação no caminho ancestral:
    - se já existe irmãos com mesma chave do pai, não fazer nada;
    - senão, propagar pelo caminho ancestral até RZ.

- Propagar por um caminho em SA.

- se remover em EL:
  - Propagação no caminho ancestral:
    - se existe irmão com mesma chave, não fazer nada;
    - senão, propagar pelo caminho ancestral, respeitando a existência de irmãos no caminho.
  - Propagar por todos os caminhos necessários em SA.

e) Em generalizações, figura 13:

- se inserir em EL:
  - Propagar pelo caminho ancestral;
  - Propagar para algum caminho em SA.
- se excluir em EL:
  - Propagar pelo caminho ancestral;
  - Propagar para o caminho (de mesma chave) em SA.

6 Considerações Finais

Este artigo apresentou uma interface E-R para o modelo relacional, que está sendo implementada. No momento em que este artigo foi escrito, o manuseio do esquema se encontra pronto, restando associá-lo ao manuseio de ocorrências.

Motro [MOTR80] defende a idéia de que estruturas rígidas que possuem arquitetura estruturada, baseada em registros, facilitam a recuperação de informações, e no entanto, impedem que o BD acompanhe a evolução constante do ambiente que retrata. Propõe como solução uma arquitetura binária baseada em coleções desestruturadas de fatos que podem ser consultados através de uma linguagem 'padrão' fundamentada em predicados. A recuperação de um fato, definido da mesma forma que um agregado - duas entidades associadas via um relacionamento binário, é realizada através de dois mecanismos: pesquisa, exame da vizinhança do fato; e tentativa, consulta seguida de falha seguida de consulta.

O armazenamento do esquema via listas ligadas impõe um caminho a ser percorrido para localizar o dado, o que implica em problemas com o tempo de acesso. Outro ponto crucial é a necessidade de se trabalhar num ambiente heterogêneo, formado pelos dois modelos envolvidos, e extrair os aspectos mais favoráveis e fundamentais de cada um sem perdas profundas no outro lado.

Uma deficiência da interface aqui proposta é que implica num mapeamento único para implementação do banco de dados. Além disto, não há preocupação em otimizar o BD relacional gerado.

Como é finalidade deste trabalho usufruir da semântica do modelo E-R para proporcionar maior compreensão e documentação do sistema, e como as fases de planejamento e análise do desenvolvimento do sistema levam a representações extensas do modelo que requerem sucessivos refinamentos, acreditamos que um futuro melhoramento seria a implementação dos conceitos descritos no *"Modelo Entidade por Agrupamento"* de Feldman e Miller [FELD86].

Acredita-se que após a aplicação desse modelo por agrupamento de entidades, outro melhoramento seria a implementação de uma linguagem gráfica para a interface, pois um dos objetivos do agrupamento é reduzir o número de entidades num contexto (tela) para facilitar sua compreensão (representação visual).

O que julgamos mais adequado seria a existência de 3 linguagens a serem suportadas pela interface: a. gráfica - dirigida aos usuários não-técnicos; b. orientada por menus - para atender usuários mais familiarizados com a área de BD, capazes de estabelecer restrições de integridade, p.ex.; c. formal, semelhante no formalismo do cálculo ou álgebra relacional - para servir aos profissionais que administrem o banco de dados.

A inexistência de formalismos para o modelo E-R nos obriga a refletir sobre as questões de normalização e otimização das relações geradas pelo mapeamento imposto. O trabalho de Chung [CHUN83] define a noção de Relações E-R e introduz formas normais E-R para decompor o diagrama - esquema E-R até normalizá-lo e obter uma correspondência de 1:1 com o esquema relacional.

Agradecimentos A primeira autora desenvolveu este trabalho como bolsista da CAPES e posteriormente da IBM - Brasil.

Referências

[ATZE83] Atzeni, P. e Chen, P.P. *Completeness of Query Languages for the Entity-Relationship Model* in [CHEN83a].

[BRAE85] Braegger,R.P.; Dudler,A.M.;Rebsamen,J. e Zehnder,C.A. *Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints, and Transactions.* IEEE Transactions on Software Engineering, SE-11(7), 1985.

[CHEN76] Chen, P.P. *The Entity-Relationship Model: Toward a Unified View of Data.* ACM TODS 1(1), 1976.

[CHEN83a] Chen,P.P. *Entity-Relationship Approach to Information Modeling and Analysis.* North-Holland, 1983.

[CHEN83b] Chen, P.P. *A Preliminary Framework for Entity-Relationship Models* in [CHEN83a].

[CODD79] Codd, E.F. *Extending the Database Relational Model to Capture More Meaning.* ACM Transations on Database Systems 4(4), 1979.

[CHUN83] Chung,I.; Nakamura,F. e Chen,P.P. *A Decomposition of Relations Using the Entity-Relationship Approach* in [CHEN83a].

[DATE77] Date, C.J. *An Introduction to Data Base Systems.* 2nd Ed., Addison Wesley, 1977.

[DELG86] Delgado, A.L.N. e Magalhães, L.P. *Assimilando a Semântica de PAC através de um Modelo de Dados.* Anais do VI Congresso da SBC, Recife, 1986.

[ELMA83] Elmasri,R. e Wiederhold,G. *GORDAS: A Formal High-Level Query Language for the Entity-Relationship Model* in [CHEN83a].

[ELMA85] Elmasri,R., Weeldreyer,J. e Hevner,A. *The Category Concept: An Extension to the Entity-Relationship Model.* North-Holland, 1985.

[FURT87] Furtado,A.L., Casanova,M.A. e Tucherman,L. *The CHRIS Consultant.* 6th International Conference on Entity-Relationship Approach, 1987.

[FELD86] Feldman, P. e Miller, D. *Entity Model Clustering: Structuring a Data Model by Abstraction.* The Computer Journal 29(1), 1986.

[MAIE85] Maier, D. *Theory of Relational Databases.* Computer Science Press, 1985.

[MALH86] Malhotra,A., Tsalalikhin,Y., Markowitz,H., Pazel,D. e Burns,L.M. *Implementing an Entity-Relationship Language on a Relational Data Base.* Relatório Técnico IBM, 1986.

[MARK83] Markowitz,H.M., Malhotra,A. e Pazel,D.P. *The ER and EAS Formalisms for System Modeling, and the EAS-E Language* in [CHEN83a].

[MOTR84] Motro,A. *Browsing in a Loosely Structured Database.* ACM SIGMOD, 1984.

[NAVA83] Navathe,S. e Cheng,A. *Database Schema Mapping* in [CHEN83a].

[PARE86] Parent, C. e Spaccapietra, S. *An Algebra for a General Entity-Relationship Model.* IEEE Transations on Software Engineering SE-11(7), 1986.

[NOGU88] Nogueira, Mônica de Lima - Tese de Mestrado, em preparação, 1988.

[ROUS84] Roussopoulos,N. e Yeh,R.T. *An Adaptable Methodology for Database Design.* Computer, May, 1984.

[SCHE80] Scheuermann,P., Schiffner,G. e Weber,H. *Abstraction Capabilities and Invariant Properties Modelling Within the Entity-Relationship Approach.* North-Holland, 1980.

[SHAV81] Shave, M.J.R. *Entities, Functions and Binary Relations: Steps to a Conceptual Schema.* The Computer Journal 24 (1), 1981.

[SMIT77] Smith,J.M. e Smith,D.C.P. *Database Abstractions: Aggregation and Generalization.* ACM Transations on Database Systems 2(2), 1977.

[TABO83] Taborier, Y. e Nànci, D. *The Occurence Structure Concept: An Approach to Structural Integrity Constraints in the Entity-Relationship Model* in [CHEN83a].

[TEOR86] Teorey,T.J., Yang,D. e Fry,J.P. *A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model.* Computing Surveys 28(2), 1986.

[WAGN87] Wagner, C.F. e Medeiros, C.B. *Um Modelo Binário Estendido para Entidade-Relacionamento.* Anais do VII Congresso da SBC, Salvador, 1987.

[WELD80] Weldon, J.L. *Using Data Base Abstractions for Logical Design.* The Computer Journal 23(1) 1980.

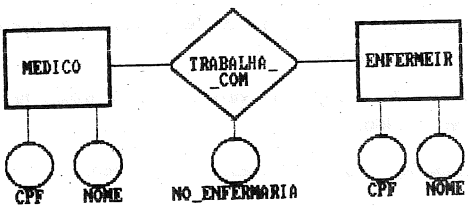


FIG. 1



FIG. 2

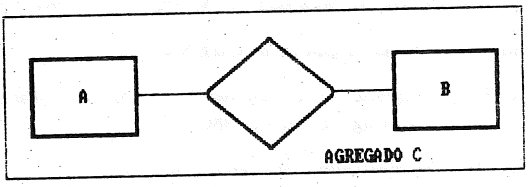


FIG. 3: AGREGACAO

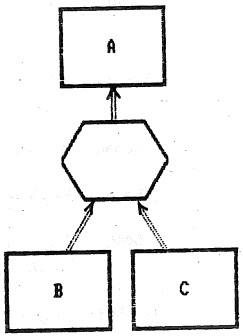


FIG.4: GENERALIZACAO

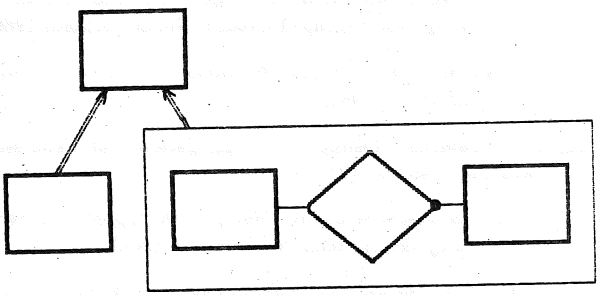


FIG. 7

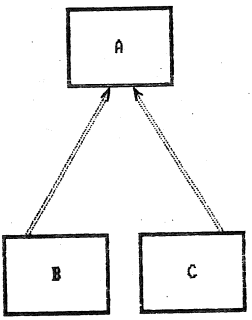


FIG.5:HIERARQUIA DE TIPOS

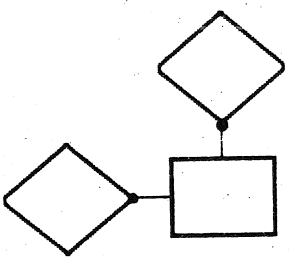


FIG.6(a)

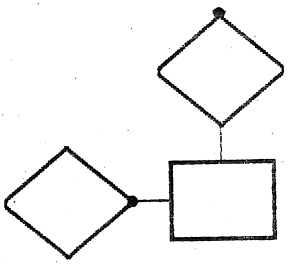


FIG.6(b)

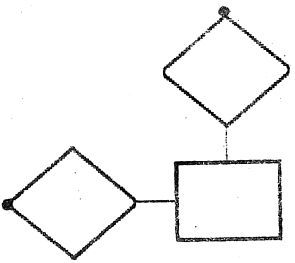


FIG.6(c)

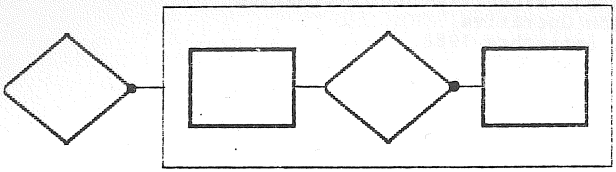


FIG. 8

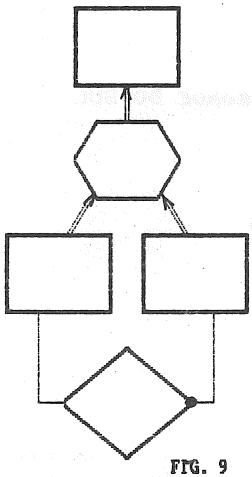


FIG. 9

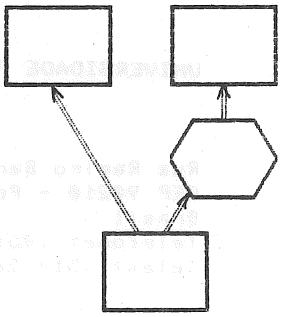


FIG. 10



FIG. 11



FIG. 12

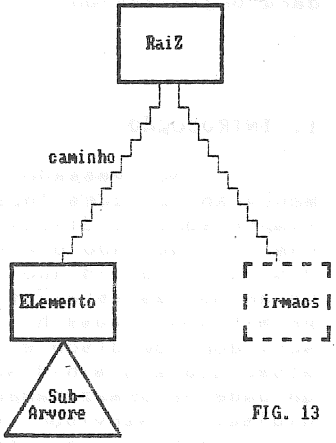


FIG. 13

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

SERVIDORES DE ARQUIVOS EM REDES DE COMPUTADORES

Liane Margarida Rockenbach Tarouco
Maria Janilce Almeida Wilkens

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Rua Ramiro Barcelos, 2574
CEP 90210 - Porto Alegre - RS
Brasil
Telefone: (0512) 31-2355
Telex: (51) 2680 - CCUF - BR

RESUMO:

O objetivo deste trabalho é apresentar um estudo sobre Servidores de Arquivos em Redes, descrevendo a implementação realizada na Universidade Federal, baseada no padrão FTAM - File Transfer, Access and Management da ISO - International Standard Organization.

1. INTRODUÇÃO

No passado, uma das maiores motivações para a implementação de redes locais foi o compartilhamento de periféricos (impressoras e discos). As primeiras implementações que surgiram no mercado ofereciam soluções para compartilhamento dos discos subdividindo-os em minidiscos, que eram alocados aos usuários nas estações de trabalho. Os mecanismos de acesso eram primitivos, possibilitando estabelecer uma associação com o servidor de disco e atuar sobre algum minidisco. A maneira de atuar sobre o minidisco era mediante o artifício de definir uma unidade de armazenamento na estação associando-a com algum minidisco no servidor. A partir daí, todos os comandos do sistema operacional, para manuseio de discos, podiam ser usados. Alguns comandos adicionais eram oferecidos para acesso a outros serviços que o servidor podia oferecer (provimento da data, relacionamento de outros usuários ativos etc.).

Com a evolução tecnológica, o preço das unidades de armazenamento diminuiu e o compartilhamento de discos deixou de

ser um fator decisivo no estabelecimento de redes locais. A própria evolução dos sistemas de processamento de dados construidos usando as redes mostrou uma necessidade de compartilhamento de informações. Os servidores tiveram que evoluir e passaram a se tornar servidores de arquivo e não mais apenas servidores de disco. A diferença principal estava no fato de que era possível um compartilhamento ordenado de arquivos, que permitisse a várias aplicações, rodando em várias estações de trabalho, acessarem concorrentemente os mesmos dados.

O Servidor de Arquivo passou a ser definido como um gerenciador físico e lógico de uma unidade de memória de massa a ser compartilhado entre os usuários, através da rede.

As principais vantagens na utilização dos Servidores de Arquivos são: compartilhamento lógico da informação, redução dos custos pelo compartilhamento do dispositivo de armazenamento, provimento de um sistema de segurança de acesso e manutenção dos arquivos, e ainda serviços de "spool" para impressoras. Mas a principal vantagem é a viabilização da troca de informções entre sistemas cooperantes. Os sistemas que interagem numa rede podem possuir características diversas, especialmente no que concerne à manipulação de arquivos. Para que se torne possível o acesso de cada um a todos os servidores de arquivo existentes na rede, é necessário que sejam resolvidas estas diferenças.

Os servidores de arquivos podem oferecer às demais estações da rede, serviços que envolvam a organização estrutural do armazenamento e um conjunto de operações que atuarão sobre a estrutura. Tais operações serão utilizadas em funções de transferência, acesso e gerência de arquivos armazenados ou trocados pelos sistemas da rede. Estas operações são providas de forma equivalente aos serviços oferecidos por um Sistema Operacional para acesso a um disco tais como leitura, escrita e informação sobre status, devendo existir ainda funções mais sofisticadas para garantir a integridade e segurança dos arquivos.

O sistema de gerenciamento realiza as funções de proteção dos arquivos contra: o acesso indevido de usuários a executar qualquer ou determinada operação, sobre um arquivo ou um conjunto de arquivos; proteção contra o acesso não autorizado ao servidor de arquivos; provimento de um sistema de controle de direitos de acesso por arquivo armazenado; mecanismos de proteção contra falhas através de recuperação automática; proteção contra acesso simultâneo a uma mesma porção de dados. Para isso, é definida uma política dinâmica de controle de acesso aos dados. Mesmo com a utilização de uma política, podem ocorrer situações de inconsistência devendo ser empregado um mecanismo de sincronização para que a proteção dos dados compartilhados seja confiável.

Em um ambiente distribuído, quando existe a necessidade de acesso a arquivos localizados em diferentes equipamentos as diferenças apontadas dificultam a implantação de aplicações distribuidas, mesmo que os diversos sistemas sejam capazes de serem interconectados em redes. É preciso que o acesso ao armazenamento tambem seja padronizado, pois, em caso contrário, ca-

da sistema teria que ter software de apoio ao acesso a cada diferente tipo de servidor de arquivo a ser usado na rede.

A norma ISO 8571 define um serviço de arquivo e especifica um protocolo que é capaz de prover um serviço de acesso, transferência e gerenciamento de arquivos. O serviço definido enquadra-se na categoria dos SASE-Specific Application Service Elements, isto é, trata-se de uma aplicação distribuída, num contexto OSI e será descrita na seção seguinte.

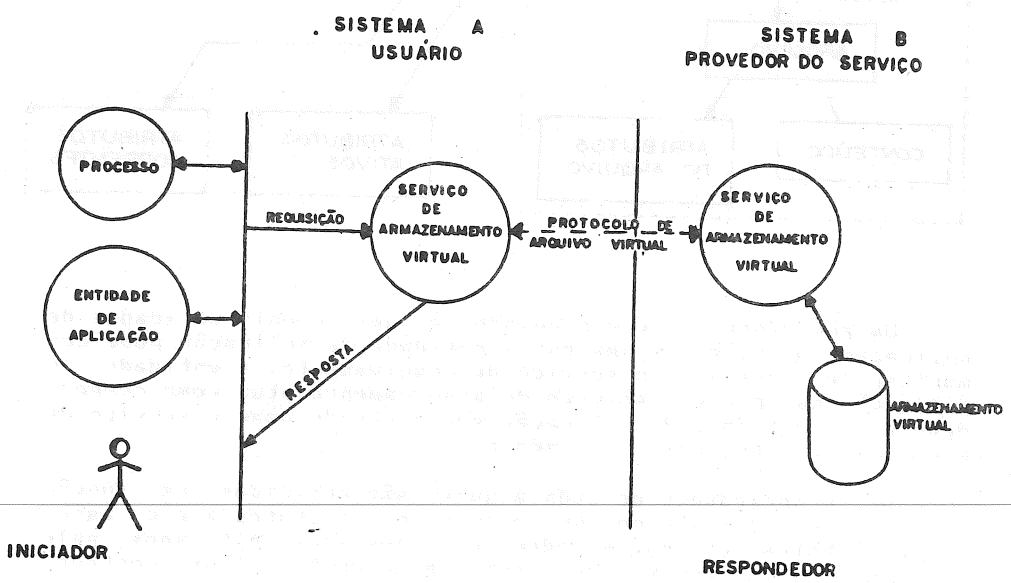
2. PADRONIZAÇÃO DO SERVIÇO DE ARQUIVO

O objetivo da padronização de um serviço de arquivo é possibilitar que sistemas abertos (no sentido definido no modelo de referência OSI) possam transferir, acessar e gerenciar arquivos remotos, independente do sistema real em que estejam armazenados. A norma ISO 8571 define este padrão denominado FTAM-File Transfer Access and Management. As incompatibilidades existentes, na forma de tratamento dos arquivos entre os diferentes tipos de sistemas presentes em uma rede heterogênea são resolvidas na padronização mediante a definição de um Serviço de Armazenamento Virtual (SAV). O SAV tem como função fazer a interface entre a entidade de serviço de arquivo e o seu sistema real de tratamento de arquivos.

Conforme ilustrado na figura 1, cada processo que atuar sobre o Serviço de Armazenamento Virtual apresentará requisições de serviço e receberá respostas contendo a informação requerida ou alguma indicação sobre o não atendimento. Estas requisições e respostas são definidas na padronização através de primitivas de serviço. Ao requerer algum serviço, um processo pode provocar a interação entre dois sistemas abertos, aquele no qual está sendo rodado o processo requerente e o outro no qual existe o recurso (dado, arquivo, diretório, ...) que deve ser acessado para atendimento da requisição. A interação entre os dois sistemas ocorre segundo um protocolo, também definido pela ISO.

O sistema que requer um serviço é definido como iniciador e o que atende é o respondedor. O iniciador vê o armazenamento gerenciado pelo respondedor como um armazenamento virtual, com todas as propriedades e atributos definidas na norma ISO. O mapeamento feito no sistema respondedor entre o armazenamento virtual e o armazenamento real (com sua estrutura e propriedades possivelmente diferentes daquelas definidas para o armazenamento virtual), é transparente para o iniciador. Desta maneira, em cada sistema aberto qualquer processo poderá acessar, de forma única e padronizada, qualquer armazenamento virtual disponível na rede.

Figura 1: Serviço de Armazenamento Virtual.

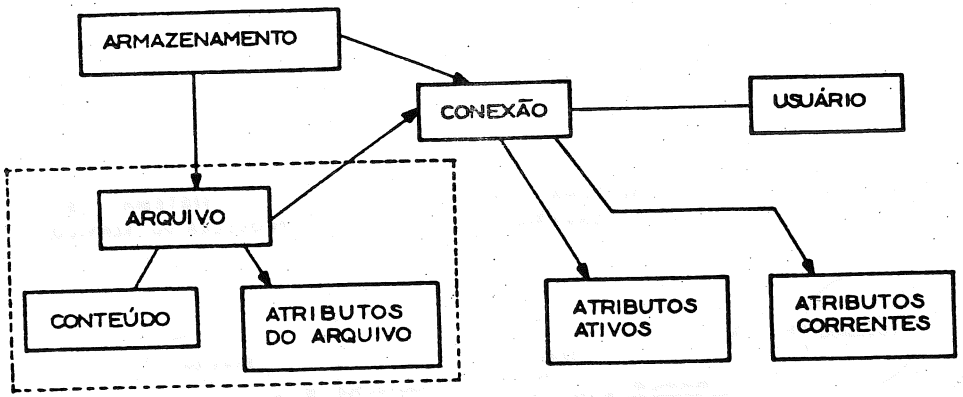


2.1 CARACTERÍSTICAS DO ARMAZENAMENTO VIRTUAL

A padronização proposta pela ISO define um modelo abstrato para um armazenamento virtual, o conjunto de ações disponíveis para manipular os elementos do modelo, as propriedades dos arquivos individuais e sua forma de representação com estruturas hierárquicas.

Um armazenamento virtual pode conter zero ou mais arquivos, tal como ilustrado na figura 2.

Figura 2: Esquema Básico do Armazenamento Virtual.

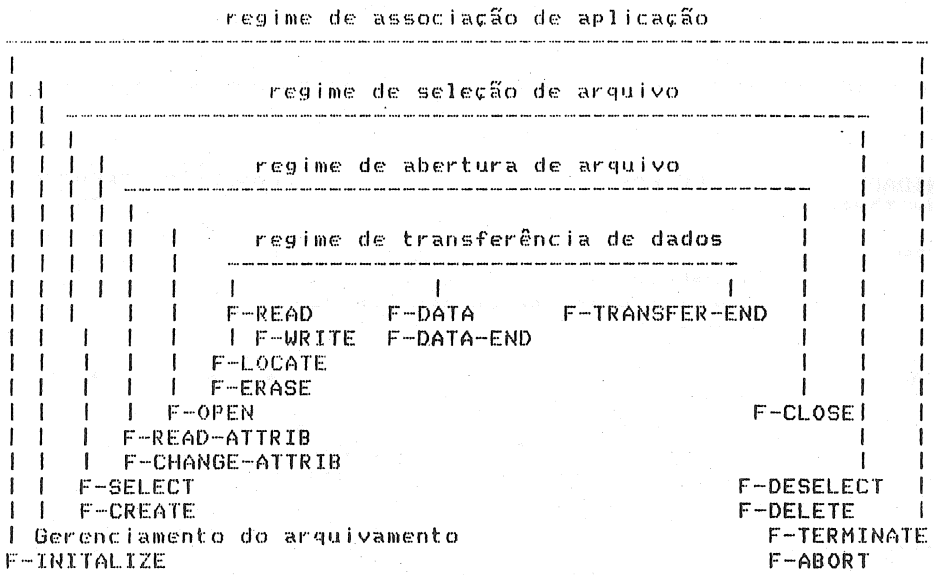


Um armazenamento é manipulado por uma ou mais entidades de aplicação com as quais uma outra entidade de aplicação pode comunicar-se para usar o serviço de arquivamento. A entidade de aplicação que provê o serviço de arquivamento atua como respondedora e a entidade de aplicação que pretende usar o serviço de arquivamento atua como iniciadora.

As propriedades de cada arquivo são definidas em função dos valores assumidos por um conjunto de atributos a ele associados. Estes atributos podem ser lidos e/ou alterados pelo iniciador. Cada arquivo pode estar vazio ou ter algum conteúdo e uma estrutura.

O iniciador pode selecionar um arquivo qualquer do armazenamento mas previamente deve estabelecer um regime de associação com a entidade aplicação que manipula o armazenamento virtual. Esta entidade é responsável pelo mapeamento do serviço de arquivo virtual definido pela ISO, num serviço de arquivamento real disponível na máquina na qual está implementado. Para que os serviços do arquivamento virtual possam ser usados é preciso que a entidade de aplicação iniciadora passe ordenadamente pelos regimes de trabalho previstos que são os esquematizados na figura 3.

Figura 3: Regimes de Trabalho.



Os quatro possíveis regimes de trabalho do servidor de arquivamento virtual definido pela ISO são:

a - regime FTAM - que existe enquanto a associação de aplicação é usada para o protocolo FTAM;
b - regime de seleção de arquivo - quando um arquivo particular está associado com o regime FTAM;
c - regime de arquivo aberto - enquanto um particular conjunto de contextos de apresentação, controles de concorrência etc estão em operação;
d - regime de transferência de dados - durante um particular contexto de acesso e direção de transferência.

Os regimes estão embutidos uns nos outros conforme mostra a figura 3. Pode haver uma série de ocorrências de um regime. Por exemplo, um regime de arquivo aberto pode conter uma sequência de regimes de transferência de dados. O encerramento de um regime implica no término de todos os regimes aninhados nele.

Em cada regime existe uma série de serviços que podem ser acionados. Estes serviços lêem/acessam/modificam algum atributo ou o conteúdo do armazenamento virtual. Os serviços são classificados em unidades funcionais e classes de serviços de arquivo. Foram definidas cinco classes de serviço:

T: Transferência de arquivo
A: Acesso a arquivo

G: Gerenciamento de arquivo
TG: Transferência e gerenciamento de arquivo
I: Irrestrita

As unidades funcionais e as classes de serviços em que são mandatórias (M) ou opcionais (O) são mostradas na tabela 2.

Tabela 2: Unidades funcionais e classes de serviços

UNIDADES FUNCIONAIS	SERVIÇOS	CLASSES DE SERVIÇO T	A	G	TG	I
KERNEL	estabelecimento do regime FTAM término ordenado ou abrupto do regime FTAM	m	m	m	m	m
READ	ler dados em bloco transferência unitária de dados fim de transferência dos dados cancelar transferência dos dados abrir e fechar arquivo	*	m		*	o
WRITE	escrever blocos de dados transferência unitária de dados fim de transferência de dados cancelar tansferência de dados abrir e fechar arquivo	*	m		*	o
ACESSO A ARQUIVO	localizar e apagar FADU		m			o
GERENCIAMENTO LIMITADO	criar arquivo deletar arquivo ler atributos	o	o	m	o	o
GERENCIAMENTO AMPLIADO	alterar atributos (requer o anterior)	o	o	m	o	o
GRUPAMENTO	início e fim de grupamento	m	o	m	m	o
RECUPERAÇÃO	regime de recuperação checkpoint cancelar transferência de dados	o	o		o	o
REINÍCIO DE TRANSFERÊNCIA DE DADOS	reinício de transferência de dados checkpoint cancelar transferência de dados	o	o		o	o

2.2 AÇÕES SOBRE O ARMAZENAMENTO VIRTUAL

São definidas uma série de ações que manipulam unidades de dados no armazenamento virtual. As ações são as seguintes:

- **Ações sobre o arquivo inteiro**

a - **CRIAR ARQUIVO:** cria um novo arquivo ou seleciona um arquivo existente e estabelece os atributos do novo arquivo; o novo arquivo passa a ter o status de selecionado e é estabelecido o regime de arquivo selecionado.

b - **SELECIONAR:** cria um relacionamento entre um iniciador e um particular arquivo; esta ação também estabelece o regime de seleção de arquivo que é pré-requisito para as ações c a h

c - **ALTERAR ATRIBUTO:** altera o valor atual dos atributos do arquivo; pode modificar o valor do atributo, no caso de ser um atributo escalar; pode substituir uma lista de elementos no caso do atributo ser vetorial; pode adicionar ou remover um elemento no caso do atributo ser do tipo conjunto.

d - **LER:** interroga o valor do atributo.

e - **ABRIR ARQUIVO:** estabelece um regime apropriado para a execução de ações de acesso ao arquivo selecionado; um arquivo pode ser aberto para ler (e neste caso somente são válidas operações de ler e localizar) ou para escrever (quando todos os acessos são válidos).

f - **FECHAR ARQUIVO:** encerra de forma ordenada o regime de arquivo aberto previamente estabelecido.

g - **DELETAR ARQUIVO:** deletar e desselecionar o arquivo, encerrando o regime corrente de seleção de arquivo.

h - **DESSELECIONAR:** encerrar de forma ordenada o regime de seleção de arquivo.

- **Ações para acesso ao arquivo**

As operações de acesso operam no regime de trabalho estabelecido pela ação de abertura de arquivo. As ações que são passíveis de execução dependem do conjunto de restrições que se aplicam ao arquivo. Os arquivos são considerados como tendo uma estrutura hierárquica e são acessados em termos de FADU-File Access Data Units, que são o mesmo que sub-árvores num modelo genérico hierárquico. A menor quantidade de dados que pode ser especificada para acesso é um DU-Data Unit. Para fins de transferência com checkpoint, as Unidades de Dados (DU) podem ser

divididas em pedaços menores, denominados DE-Data Elements. Contudo, não é possível acessar um Data Element com as ações definidas pela ISO 8571 para o armazenamento virtual. Somente DUs como um todo são acessados.

a - **LOCALIZAR**: localiza um específico FADU-File Access. Os FADUs passíveis de obtenção são: primeiro, último, atual, próximo, anterior, por nome (é especificada a identidade do FADU), por número, por nível

b - **LER**: localiza e le um FADU

c - **INSERIR**: cria um novo FADU e o insere na posição apropriada no arquivo

d - **SUBSTITUIR**: atualiza o conteúdo de um FADU existente

e - **EXTENDER**: adiciona dados ao final do arquivo

f - **APAGAR**: elimina um FADU

2.3 ATRIBUTOS DO ARMAZENAMENTO VIRTUAL

Cada atributo de arquivo no armazenamento virtual é global, no sentido de que tem somente um valor ou conjunto de valores, num particular instante de tempo. Todos os iniciadores de ações sobre o arquivo verão o mesmo valor, conjunto de valores ou obterão a resposta "nenhum valor disponível" para um atributo do arquivo. Os atributos definidos são:

FILENAME: atributo vetorial contendo uma sequência de componentes do nome
AÇÕES PERMITIDAS: atributo vetorial que indica o conjunto de ações que podem ser efetivadas sobre o arquivo e o conjunto de tipos de FADUs que podem ser usados numa ação de localização
CONTROLE DE ACESSO: atributo do tipo "conjunto" que define condições sob as quais é válido o acesso ao arquivo
CONTABILIZAÇÃO : identifica a autoridade contábil responsável pela contabilização do uso do arquivo
DATA E HORA DECRIAÇÃO
DATA E HORA DA ÚLTIMA MODIFICAÇÃO
DATA E HORA DO ULTIMO ACESSO PARA LEITURA
DATA E HORA DO ULTIMO ACESSO PARA MODIFICAÇÃO
IDENTIDADE DO CRIADOR DO ARQUIVO
IDENTIDADE DO ÚLTIMO MODIFICADOR
IDENTIDADE DO ÚLTIMO LEITOR
IDENTIDADE DO ULTIMO MODIFICADOR DE ATRIBUTO
DISPONIBILIDADE DO ARQUIVO
TIPO DE CONTEÚDO: indica os tipos abstratos de dados do conteúdo do arquivo
NOME CRIPTOGRÁFICO: determina o algorítimo usado para criptografar o arquivo
TAMANHO DO ARQUIVO (em octetos)

TAMANHO FUTURO :valor máximo em octetos que o tamanho do arquivo pode assumir
QUALIFICAÇÃO LEGAL: status legal do arquivo e seu uso (depende de legislação sobre proteção de dados)
USO PRIVADO: atributo sem significado definido pela ISO 8571, pode assumir qualquer forma desejada pelo usuário

Existem ainda outros atributos, denominados de atividade que refletem o estado do regime em andamento:

PEDIDO DE ACESSO EM ANDAMENTO
IDENTIDADE DO INICIADOR ATUAL
PASSWORD DE ACESSO ATUAL
NOME DA APLICAÇÃO CHAMANTE ATUAL
NOME DA APLICAÇÃO RESPONDEDORA ATUAL
AUTORIDADE RESPONSÁVEL PELO USO (CONTABILIZAÇÃO)
MODO DE PROCESSAMENTO ATUAL (ações permitidas)
CONTEXTO DE ACESSO CORRENTE (hierárquico, plano, unitário, não estruturado)
CONTROLE DE CONCORRENCIA ATUAL: restrições de acesso paralelo.
LOCALIZAÇÃO CORRENTE
QUALIFICAÇÃO LEGAL ATIVA
TIPO DE CONTEÚDO ATIVO

Uma implementação do armazenamento virtual pode implementar apenas uma parcela destes atributos. A norma especifica os grupos de atributos que podem ser implementados e os nomes que tais grupos recebem. O conjunto mínimo de atributos a serem implementados é denominado KERNEL e prevê o uso dos seguintes atributos:

ATRIBUTOS DE ARQUIVO: Nome do arquivo e tipo de conteúdo
ATRIBUTOS DE ATIVIDADE: Tipo de conteúdo ativo, pedido de acesso, localização, modo de processamento e nome da entidade de aplicação atuais

Outros grupos de atributos são:

GRUPO DE ARMAZENAMENTO: permite a uma atividade referenciar informação estabelecida por alguma atividade anterior

GRUPO DE SEGURANÇA: prevê mecanismos de segurança e controle de acesso e somente pode ser usado se o GRUPO DE ARMAZENAMENTO tambem for usado.

Pode haver ainda um outro grupo denominado PRIVADO, que não foi definido na padronização OSI que poderá ser livremente definido por grupos privados de usuários.

2.4 PROTOCOLOS USADOS

O serviço de armazenamento virtual é provido por uma aplicação que é executada usando os serviços previstos no modelo OSI para as camadas de alto nível. Os protocolos especificados em ISO 8571/4 suportam os serviços anteriormente. O protocolo de arquivo especificado em ISO 8571/4 assume o uso dos seguintes serviços nos níveis de sessão e apresentação:

Unidade Funcional do FTAM	Unidade Funcional de SESSÃO	Unidade Funcional de APRESENTAÇÃO
Kernel	Kernel Duplex Ressincronização (opcional)	Kernel Ressincronização (opcional) Gerência de Contexto (opcional)
Recuperação	Sincronização Menor	
Reinício	Sincronização Menor Ressincronização	

Em princípio, a operação do protocolo é modelada pela interação entre duas FPM - File Protocol Machine. As duas máquinas comunicam-se por meio dos serviços disponíveis nos níveis inferiores.

Os PDUs - Protocol Data Units do protocolo de arquivo são tipos de dados complexos definidos usando ASN.1. A definição inclui campos com nome, e outros dados correspondentes aos parâmetros necessários para a operação do protocolo. A seguir é apresentada a definição parcial de alguns PDUs, a título de exemplo:

```
IS08571-FTAM DEFINITIONS ::=

BEGIN

PDU ::= CHOICE { FTAM-Regime-PDU, File-PDU, Bulk-Data-PDU }

File-PDU ::= CHOICE {
     [ 6] IMPLICIT F-SELECT-request,
     [ 7] IMPLICIT F-SELECT-response,
     [ 8] IMPLICIT F-DESELECT-request,
     [ 9] IMPLICIT F-DESELECT-response,
     [10] IMPLICIT F-CREATE-request,
     [11] IMPLICIT F-CREATE-response,
     [12] IMPLICIT F-DELETE-request,
     [13] IMPLICIT F-DELETE-response,
     [14] IMPLICIT F-READ-ATRIB-request,
     [15] IMPLICIT F-READ-ATRIB-response,
     [16] IMPLICIT F-CHANGE-ATRIB-request,
     [17] IMPLICIT F-CHANGE-ATRIB-response,
     [18] IMPLICIT F-OPEN-request,
     [19] IMPLICIT F-OPEN-response,
     [20] IMPLICIT F-CLOSE-request,
     [21] IMPLICIT F-CLOSE-response,
     [22] IMPLICIT F-BEGIN-GROUP-request,
     [23] IMPLICIT F-BEGIN-GROUP-response,
     [24] IMPLICIT F-END-GROUP-request,
     [25] IMPLICIT F-END-GROUP-response,
```

```
      [26] IMPLICIT F-RECOVER-request,
      [27] IMPLICIT F-RECOVER-response,
      [28] IMPLICIT F-LOCATE-request,
      [29] IMPLICIT F-LOCATE-response,
      [30] IMPLICIT F-ERASE-request,
      [31] IMPLICIT F-ERASE-response,
      }

F-SELECT-request ::= SEQUENCE {
  attributes   Attributes,
   --  Somente  o atributo filename deve estar  presente  neste
campo
  requested-access     Access-Request,
  access-passwords     Access-Passwords OPTIONAL,
  concurrency-control   Concurrency-Control OPTIONAL,
  commitment-control   Commitment-Control OPTIONAL,
  account    Account OPTIONAL
  }

Attributes ::= [APPLICATION 11] IMPLICIT SEQUENCE OF CHOICE {
--Kernel - group
  filename [0] IMPLICIT SEQUENCE OF GraphString,
  contents-type [1] Contents-Type-Attribute
  }

Contents-Type-Attribute ::= [APPLICATIONS 12] CHOICE {
   document-type-name  Document-Type-Name,
   constraint-set-and-abstract-syntax [0] IMPLICIT SEQUENCE {
        constraint-set-name Constraint-Set-Name,
        abstract-syntax-name  Abstract-Syntax-Name
        }
   }

Access-Request ::=  [APPLICATION 13] IMPLICIT BITSTRING
                {   read (0),
                    insert (1),
                    replace (2),
                    erase (3),
                    extend (4),
                    read-attribute (5),
                    change-Attribute (6)
                    delete-file (7)  }

Access-Password ::= [APPLICATION 14] IMPLICIT SEQUENCE {
      read-password [0] IMPLICIT Password,
      insert-password [1] IMPLICIT Password,
      replace-password [2] IMPLICIT Password,
      erase-password [3] IMPLICIT Password,
      extend-password [4] IMPLICIT Password,
      read-attribute-password [5] IMPLICIT Password,
      change-attribute-password [6] IMPLICIT Password,
      delete-password [7] IMPLICIT Password,
      }

Password ::= [APPLICATION 6] CHOICE  { Graphstring, OCTETSTRING
}

Account ::= [APPLICATION 5] IMPLICIT GraphicString
```

3. UM PROTÓTIPO DE FTAM NO PROJETO REDURGS

O Projeto REDURGS - Rede da Universidade Federal do Rio Grande do Sul está sendo desenvolvido com recursos da FINEP e tem por objetivo integrar os diversos sistemas computacionais da Universidade. Esta integração está sendo baseada no modelo OSI da ISO. Um dos serviços selecionado para ser oferecido aos usuários da rede, foi o Serviço de Transferência de Arquivos. Este serviço foi projetado e está sendo implantado segundo o padrão FTAM definido pela ISO. O protótipo inicial contempla a transferência de arquivos entre microcomputadores e o computador A10 (da UNISYS) na UFRGS. No protótipo sendo implementado, os microcomputadores (tipo PC) atuam como iniciador e o A10 como respondedor.

O software implementado no A10 foi escrito em DCALGOL. Foram implementados três níveis do modelo OSI: Transporte, Sessão e Aplicação (FTAM). O Nível de Transporte foi implementado de forma restrita, realizando somente as funções exigidas para assegurar a transferência confiável de TSDUs (Transport Service Data Units) num enlace onde é usado o protocolo Poll-Select. Na implementação feita, para uso interno, não está sendo usado o serviço de nível 3, sendo as conexões ponto-a-ponto ou multiponto com o A10. A função básica do nível de transporte implementado é assegurar que erros não sinalizados (perda de mensagens) sejam detectados. Os TPDUs são numerados e confirmados um a um (trabalha-se com janela de tamanho 1) /TAR 86/.

Outro nível implementado foi o de Sessão. O Nível de Sessão tem como objetivo fornecer os meios necessários para organizar a sinalização e o diálogo entre os sistemas. Atende pedidos pedidos de estabelecimento de conexão, transferência de dados e término de conexão recebidos do nível superior e envia os SPDU apropriados. Foi implementada a unidade funcional Kernel do nível de sessão com capacidade de segmentação de SSDUs (Session Service Data Units).

O último nível implementado foi o da Aplicação FTAM. Não foi implementado o nível de apresentação porque nenhuma seleção de contexto ou transformação de sintaxe é necessária nesta fase do projeto.

A comunicação entre as duas máquinas via FTAM dá-se através de duas entidades, conforme descrito na seção 3:

- Iniciador, que na implementação feita é um microcomputador PC

- Respondedor que é o mainframe A10.

A comunicação entre o Iniciador e o Respondedor é feita através do envio de FPDU's - FTAM Protocol Data Units. Estas FPDU's possuem parâmetros que identificam os serviços que devem ser realizados ou que respondem a esses serviços. Os serviços que são definidos por meio de primitivas descritas a seguir:

- SELECT - utilizada para selecionar um arquivo, para que este seja transferido do A10 para o PC;
- CREATE - utilizada para criar um arquivo no A10. Ocorrerá a transferência de um arquivo do PC para o A10;
- OPEN - utilizada para abrir um arquivo no A10. Essa primitiva é necessária tanto após um SELECT quanto um CREATE;
- CLOSE - utilizada para fechar um arquivo no A10;
- DESELECT - utilizada para se retornar ao estado inicial "CONNECT", para que seja feito um novo SELECT ou CREATE;
- READ - utilizada quando se dá um SELECT, pois será necessário haver a leitura do arquivo;
- WRITE - utilizada quando se dá um CREATE, pois será necessário haver uma gravação do arquivo que o PC está transferindo;
- DATA-END - indica fim de arquivo;
- TRANSFER-END - indica fim de transferência.

A iniciativa de conectar/desconectar é sempre tomada pelo Iniciador (PC). Após ser feita a conexão PC-A10, o processo entra no estado CONNECTED, sendo este o estado considerado inicial do FTAM implementado. O diagrama de estados a seguir mostra as interações realizadas entre o Iniciador (PC) e o Respondedor (A10).

Definição dos ESTADOS do Diagrama de Estados.

ESTADOS:

CONNECTED - é o estado INICIAL e indica que o PC e o A10 estão conectados.

SELECT_PD_R - indica que há um pedido de SELECT (feito pelo PC) pendente e aguarda-se a resposta do receptor (A10).

CREATE_PD_R - indica que há um pedido de CREATE (feito pelo PC) pendente e aguarda-se a resposta do receptor (A10).

DESELECT_PD_R - indica que há um pedido de DESELECT (feito pelo PC) pendente e aguarda-se a resposta do receptor (A10).

SELECT_R - indica que um pedido de SELECT, CREATE ou CLOSE foi aceito pelo receptor (A10), ou ainda pode indicar que um pedido pendente de OPEN em um arquivo não foi aceito pelo A10.

OPEN_D_R - indica que há um pedido de OPEN (vindo do PC) pendente e aguarda-se a resposta do receptor (A10).

DXFRIDLE_R - indica que o A10 está pronto para fazer um READ ou WRITE num arquivo.

READ_R - indica que está ocorrendo a leitura de um arquivo no A10.

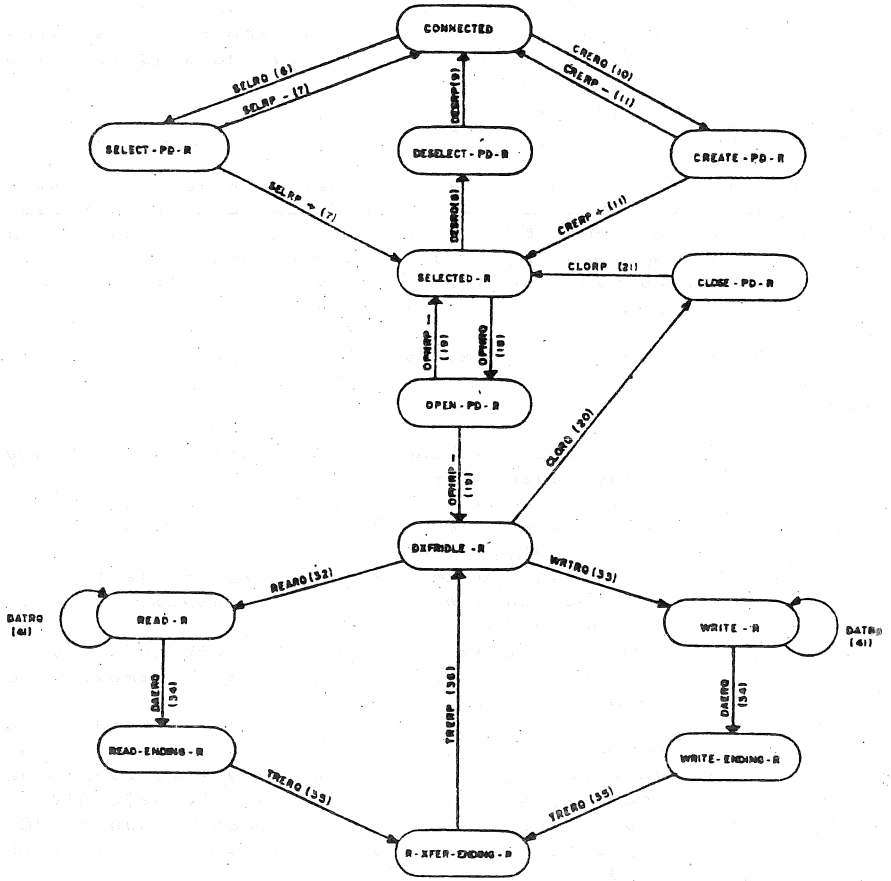
WRITE_R - indica que está ocorrendo a gravação de um arquivo no A10.

READ_ENDING_R - indica que ocorreu o fim da leitura de um arquivo no A10.

WRITE_ENDING_R - indica que ocorreu o fim da gravação de um arquivo no A10.

R_XFER_ENDING_R - indica que foi feito um pedido de TRANSFER_END pelo PC e é aguardada uma resposta do receptor

Figura 4: Diagrama de Estados.

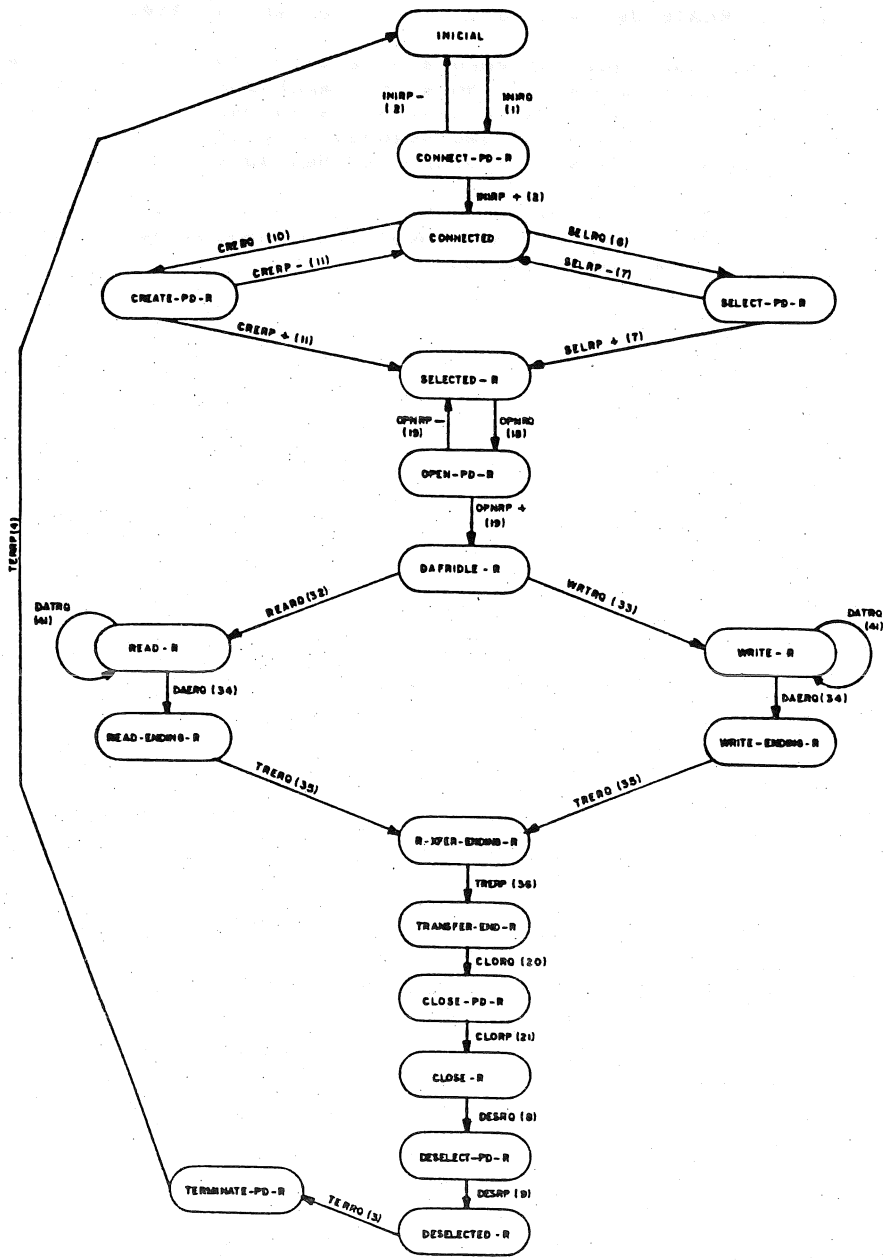


Convém salientar que o que determina se uma resposta, enviada pelo A10, é positiva ou negativa é um diagnóstico indicando se houve falha (negativa) ou sucesso (positiva). As falhas podem ser detectadas no momento em que o PC requisitar um SELECT de um arquivo que não existe no A10 ou quando o PC requisitar um CREATE de um arquivo que já existe no A10.

O software que implementa o serviço FTAM no microcomputador foi escrito em "C". No microcomputador, o sistema é composto de quatro módulos ou níveis distintos. O nível mais básico interage com o mainframe segundo o protocolo Poll-select. Os demais níveis são: Transporte, Sessão e FTAM.

O diagrama da figura 5 define as interações a nível de aplicação, entre o PC e o A10, mostrando também os serviços possíveis de serem requisitados.

Figura 5: Interações entre o PC e o A10.



4. CONCLUSÃO

A decisão de implementar um protótipo do serviço FTAM na UFRGS foi tomada porque se acredita na importância da padronização do serviço de arquivos em redes de computadores. A padronização proposta pela ISO é bastante complexa e julgou-se que somente mediante uma implementação experimental se teria condições de avaliar com melhor discernimento as características importantes e as facultativas num serviço de arquivo realmente útil e prático. Não é aconselhável que sejam sacrificadas funcionalidades essenciais sem que uma análise custo-benefício seja efetuada. A partir do uso do protótipo, esta sendo buscada a realimentação necessária para definir um serviço de arquivo para redes mais completo e eficiente. Outros serviços, baseados no FTAM tambem estão sendo projetados.

Cabe destacar que a implementação ora apresentada está baseada no resultado de trabalhos anteriores que visaram a implementação de outros sistemas experimentais baseados em padrões internacionais / MUS 85/ e /WIL 86/. Pretende-se posteriormente transferir o know-how adquirido para a indústria, a exemplo do que já ocorreu com protótipos anteriormente desenvolvidos na UFRGS na área de comunicação de dados.

5. BIBLIOGRAFIA

/ISO 85a/ ISO. File transfer, access and management - Part 1: general description. s.l., Apr. 1985 (ISO/TC96/SC21 N 516).

/ISO 85b/ ISO. File transfer, access and management - Part 1: the virtual filestore. s.l., Apr. 1985 (ISO/TC97/SC21 N 517).

/ISO 85c/ ISO. File transfer, access and management - Part 3: the file service Dc function. s.l. Apri. 1985 (ISO/TC97/SC21 N 518).

/ISO 85d/ ISO. File transfer access and management - Part 4: the file protocol specification. s.l., Jul. 1986 (ISO/TC97/SC21 N 1218).

/MUS 85/ MUSSE,J & Branco, R & TAROUCO,L. Uma análise do serviço de transporte. 3º SBRC, Rio de Janeiro, 1985.

/TAN 81/ TANENBAUM, A.S. "Computer Networks", Prentice Hall, 1981, pp. 512.

/TAR 86/ TAROUCO, Liane. Redes de Computadores: locais e de longa distância. McGraw-Hill, São Paulo, 1986.

/WIL 86/ WILKENS, Maria J. A. Um Sistema de Mensagens Distribuído. PGCC-UFRGS, Junho, 1986.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

EM BUSCA DE UM SISTEMA GENERALIZADO DE GERENCIA DE DADOS, MODELOS, E INTERFACES COM O USUÁRIO.

MARIA HELENA BRAZ*

RUBENS NASCIMENTO MELO**

Dep. de Informática - PUC/RJ
R. Marquês de S. Vicente 225
22453 Gávea - Rio de Janeiro
BRASIL
Telef. (021) 5299385

SUMARIO

Com o objetivo de definir as caracteristicas de um sistema genérico para Gerência integrada de Dados, de Modelos e de Interfaces com o Usuário, considerado importante para dar suporte ao desenvolvimento de aplicações como Sistemas de Apoio à Decisão (SAD) onde estes três aspectos assumem igual relevância, procurou-se estabelecer uma analogia de conceitos entre Sistemas de Gerência de Dados, de Modelos e de Interfaces com o Usuário, e propõe-e o paradigma de objetos como forma de integrar estes três sistemas.

* Eng. Eletrotécnica (I.S.T, 1979), Mestre em Pesquisa Operacional e Engenharia de Sistemas (I.S.T., 1984); Prof. Ext. Dept. de Inf. PUC-RJ; Engenharia de Software, Sistemas de Apoio à Decisão, Interfaces com o Usuário.

** Dr.Sc ITA 1976; Prof. Associado Dept. de Inf. PUC-RJ; Banco de Dados, Engenharia de Software, Sistemas Gráficos Interativos.

I - INTRODUÇÃO.

Até há alguns anos, um programador de aplicação tinha que incluir na "lógica" de seu programa, além dos procedimentos específicos para resolver o problema em si, também o código correspondente à gerência dos dados e todos os comandos de entrada e saída (E/S) necessários para a comunicação com o usuário.

A primeira evolução no sentido de diminuir a complexidade das tarefas de programação correspondeu à criação de bibliotecas de subrotinas que ofereciam funções previamente implementadas sendo posteriormente ligadas ao código desenvolvido. Foi o caso das bibliotecas com funções matemáticas que apoiavam a linguagem FORTRAN.

A segunda evolução correspondeu à criação de Sistemas de Gerência de Banco de Dados (SGBD), retirando da aplicação a necessidade de gerenciar os dados. A integração com os procedimentos computacionais da aplicação passou a ser feita através das linguagens de manipulação de dados oferecidas pelo SGBD.

Em paralelo, e com o desenvolvimento da tecnologia, surgiram os sistemas interativos, baseados em novos dispositivos de E/S, para os quais a interface tomou uma nova importância pois passou a ser possível ao usuário final interagir diretamente com o sistema e até controlar a execução das aplicações. Também nesta situação a primeira evolução para simplificar o uso destes novos dispositivos consistiu na criação de pacotes gráficos oferecendo funções de alto nível e aliviando a programação de detalhes de controle desses dispositivos.

Posteriormente, a necessidade de desenvolver aplicações com interfaces de qualidade recorrendo a prototipação sugeriu que, à semelhança do que acontecera para os dados, se deveria procurar gerenciar as interfaces exteriormente às aplicações de tal modo que se pudessem adaptar livremente as interface sem necessariamente alterar os módulos da aplicação.

Por outro lado o aparecimento de aplicações tipo Sistemas de Apoio à Decisão (SAD), veio salientar a importância dos Modelos como recurso que, à semelança dos dados e interfaces poderia ser também gerenciado com vantagens do ponto de vista da consistência, segurança, etc.. e garantindo um

desenvolvimento mais simples destas aplicações.

O conceito de Modelo, familiar na área de SAD, corresponde a uma abstração do mundo real que se caracteriza por depender do domínio do problema que se pretende resolver. Por exemplo, um Modelo num SAD poderia ser um módulo de programa que permite realizar projeções de vendas.

Repare-se que geralmente os Modelos assentam em ferramentas computacionais do tipo das já anteriormente armazenadas em bibliotecas mas destinguem-se destas por terem associada uma lógica deduzida da realidade modelada. Como exemplo poderemos dizer que o algoritmo Simplex é a ferramenta sobre a qual se implementa um Modelo para maximização de lucro atendendo a determinadas restrições espressas por expressões lineares.

Verifica-se então, que uma forma de simplificar o processo de desenvolvimento de aplicações será recorrer a Sistemas de Gerência de Dados, Modelos e Interfaces, capazes de isolar dentro do possível, os detalhes de controle de cada um destes aspectos.

O desenvolvimento de aplicações como SAD onde estes três aspectos - Dados, Modelos e Interfaces - são fundamentais aponta ainda a necessidade da sua integração. Neste sentido, este trabalho procura analisar as características de Dados, Modelos e Interfaces de forma a encontrar analogias permitindo definir um Sistema Generalizado para Gerência comum destes aspectos.

II - ANALOGIA ENTRE GERENCIA DE DADOS, GERENCIA DE MODELOS E GERENCIA DE INTERFACES.

A criação dos SGBD foi, como se sabe, fortemente motivada por questões como: (1) Redundância e Consistência; (2) Integridade e Segurança; (3) Uso compartilhado; (4) Normalização; (5) Independência lógica e física.

Estas questões motivaram igualmente o desenvolvimento de Sistemas de Gerência para Modelos e Interfaces.

No caso dos Modelos, redundância significa a existência de múltiplas versões do mesmo Modelo desenvolvidas independentemente. Inconsistência surge devido a problemas de validade e precisão que podem fazer com que dois Modelos

aparentemente idênticos apresentem resultados distintos. Será necessário ter acesso às condições de aplicação do Modelo garantindo que ele é usado corretamente. Além disso, será necessário criar mecanismos garantindo a integridade e segurança dos Modelos gerenciados, que consistirá fundamentalmente em assegurar o uso correto e proteger contra eventuais alterações de parâmetros ou outros elementos que possam invalidar o Modelo.

O uso compartilhado será ainda um benefício pois atualmente a criação de Modelos é geralmente feita numa base pessoal e após o uso do Modelo este é descartado o que traz problemas de produtividade e custo para a organização. São exemplos desta situação os Modelos criados recorrendo a planilhas de cálculo.

O problema da normalização está fortemente ligado com redundância e inconsistência. Na realidade a normalização dos Modelos evita a existência de diferentes Modelos para uma mesma situação garantindo que todos os usuários obtêm resultados semelhantes no seu uso.

A independência lógica e física, que foi um objetivo fundamental alcançado pelos SGBD, corresponderá, neste ambiente, à definição de Modelos de forma lógica e independente da descrição da sua implementação física.

A descrição de um Modelo e a sua solução devem ser funções separadas permitindo que todos os usuários possam descrever Modelos com uma linguagem única e que a sua representação seja transformada para o nível apropriado à sua solução.

Do ponto de vista das organizações será necessário prever a existência de novas funções ligadas à Gerência de Modelos. Tal como surgiu o Administrador de Dados e o Administrador de Base de Dados para gerenciar dados, também para os Modelos se poderá pensar na existência de um Administrador de Modelos e um Administrador de Base de Modelos.

Naturalmente será de prever uma grande cooperação entre as Administrações de Dados e Modelos, sendo possível considerar pelo menos na fase inicial que a administração de dados já existente englobe os Modelos. No entanto com a crescente aceitação dos Modelos como recurso exigindo conhecimento

profundo em áreas como Estatística, Pesquisa Operacional, Otimização, etc, deverá necessariamente se dissociar alcançando uma colocação equivalente à administração de dados na estrutura da empresa.

Tal como a Administração de Dados, a Administração de Modelos deverá ter, à sua disposição, ferramentas de Software para lhe dar suporte. Em [Dolk 85] são indicados como fundamentais um Sistema Dicionário/Diretório (SDD) e um SGM correspondentes ao Dicionário de Dados e ao SGBD.

O SDD será a ferramenta fundamental para o AM pois conterá informações sobre os Modelos existentes e fornecerá documentação básica sobre eles. O SGM será importante para o ABM pois permitirá armazenar, manipular e controlar os Modelos. Ambos poderão compor um SGM genérico dependendo do grau de integração alcançado.

A compatibilidade com o SGBD será fundamental pois os Modelos necessariamente utilizarão os dados do BD devendo o SGM ser desenvolvido atendendo ao SGBD disponível.

Para conseguir independência lógica, o usuário do SGM deverá poder especificar os Modelos numa Linguagem de Descrição de Modelos (LDM). Ao usar esta linguagem o usuário deverá preocupar-se com a descrição do Modelo esquecendo a representação necessária à respectiva solução.

Para operar com os Modelos o usuário terá à sua disposição uma Linguagem para Manipulação de Modelos (LMM) que permita atualização, resolução, consulta, ligação, podendo ser uma linguagem específica ou estar embutida numa linguagem de programação já existente.

Outro componente do SGM será o Sistema de Controle de Modelos (SCM) que gerenciará o acesso, o armazenamento e consulta ao nível físico de implementação.

Qualquer destes componentes, LDM, LMM e SCM têm equivalência direta com os componentes de um SGBD (Linguagem de Descrição de dados, Linguagem de Manipulação de Dados e Sistema de Controle de Base de Dados), no entanto, no caso dos Modelos, será ainda necessário considerar um novo componente que é a Biblioteca de Soluções contendo os algoritmos utilizáveis na solução dos Modelos, por exemplo, o algoritmo do Simplex para resolução de Modelos de programação linear.

Esta biblioteca deverá conter um conjunto de algoritmos adequado às aplicações pretendidas pela organização e servindo as classes de Modelos existentes.

Na fig.1 ilustramos as ligações entre os componentes de SGM associando-os com os diferentes níveis de usuários.

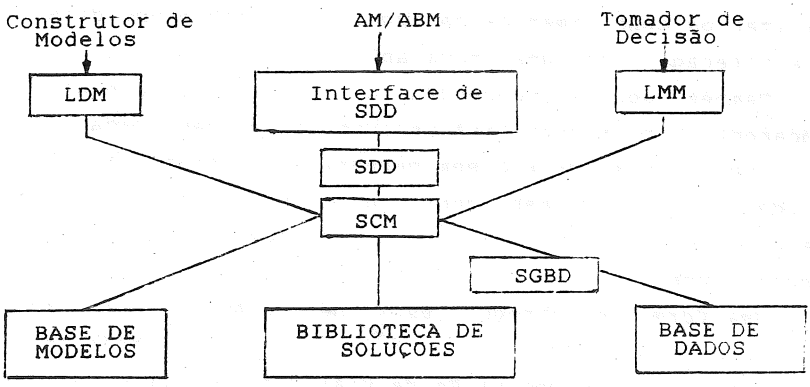


fig.1 - Os Componentes de um SGM.

Apesar da analogia estabelecida entre Dados e Modelos para ilustrar o que será um SGM é conveniente notar que os Modelos são, na realidade, entidades bem mais complexas que os Dados envolvendo parâmetros e relações sofisticadas entre eles, geralmente expressas de forma algébrica.

Uma interface, por sua vez, é descrita por diálogos apoiados em técnicas de interação. Também neste caso poderemos seguir uma abordagem identica á anterior e verificar que os Sistemas de Gerência de Interfaces com o Usuário (SGIU), atualmente propostos, procuram resolver o mesmo tipo de questões que motivaram quer os SGBD quer os SGM.

Para as Interfaces também será importante evitar redundância e garantir consistência. Dispondo de um SGIU o projetista de interfaces poderá recorrer a técnicas de interação, ícones, janelas, etc... previamente definidos para montar as suas interfaces aproveitando rotinas já existentes. Quanto á consistência, ela significa que a interface criada deverá apresentar uma uniformidade nas formas de interação que terá como vantagem a possibilidade dos usuários recorrerem á analogia para exacutarem novas situações de interação.

Isolando a interface das funções da aplicação garante-se também a possibilidade de usar partes de uma interface em outras aplicações semelhantes, isto é, o uso compartilhado de componentes das interfaces.

A normalização neste caso também será importante pois garantirá consistência entre diferentes interfaces. A existência de interfaces normalizadas torna possível ao usuário utilizar os conhecimentos já adquiridos, sobre uso de sistemas, para interagir com uma nova aplicação.

Também aqui a independência lógica e física é um objetivo fundamental pois permitirá recorrer a uma gama cada vez maior de dispositivos de E/S sem necessitar alterar as interfaces criadas. Estas corresponderão, no nível lógico, à definição de diálogos e de estilos de interação independentes de dispositivos.

Uma forma de conseguir esta independência lógica será, de forma semelhante ao que foi criado para dados e Modelos, criar uma Linguagem de Descrição de Diálogos LDDg cujo uso permite especificar o diálogo correspondente à interface pretendida, ignorando a forma como as técnicas de interação irão implementar tal diálogo.

Para manipular diálogos o projetista de interfaces necesssitará também de uma Linguagem de Manipulação de Diálogos LMDg que irá permitir a atualização, execução, ligação, consulta dos diálogos criados.

Outro componente do SGIU será o Sistema de Controle de Diálogos SCD que, a partir da especificação de um diálogo na LDDg, é capaz de gerenciar a execução dos módulos de aplicação e o acesso e consulta ao nível interno de implementação das técnicas de interação.

Tal como no caso dos Modelos, também aqui é necessário prever a existência de uma biblioteca de técnicas de interação que corresponderá à biblioteca de soluções.

Na fig.2 apresentamos os componentes do SGIU associando-os com diferentes níveis de usuário.

Qualquer dos componentes LDDg, LMDg e SCD têm eqivalência direta com os componentes já descritos anteriormente para os SGBD e SGM. Para completar tal equivalência será necessario considerar também a existência de um Sistema Dicionário de

Dialogos e Módulos de Estilo (SDDME) que, de forma semelhante aos seus congêneres, conterá informação sobre todos os diálogos e técnicas de interação organizadas em módulos de estilo, fornecendo a documentação básica do SGIU. A responsabilidade deste componente caberá ao Administrador de Diálogos (AD) que deverá estabelecer as normas de construção garantindo a segurança, consistência e normalização do SDDME.

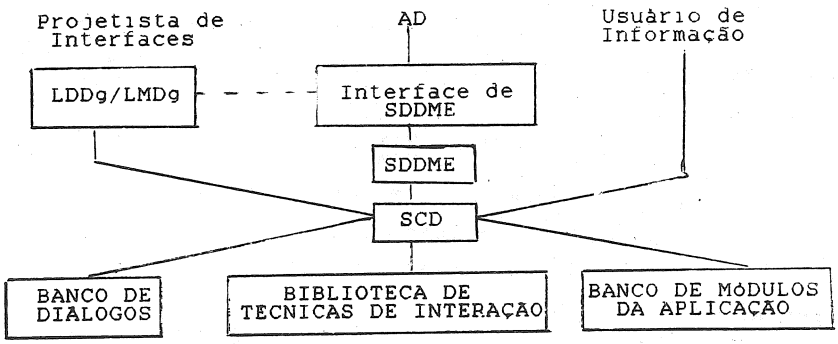


fig.2 - Os Componentes de um SGIU.

III - INTEGRAÇÃO DE DADOS, MODELOS E INTERFACES EM SAD.

Como vimos anteriormente a principal motivação para o desenvolvimento de Sistemas de Gerência consiste essencialmente em procurar simplificar a complexidade das tarefas ligadas ao desenvolvimento de aplicações.

Em aplicações dirigidas para níveis mais altos de gerência na empresa, que necessitam recorrer a Modelos para satisfazer as suas necessidades em termos de tratamento de informação, o desenvolvimento será beneficiado pela existência de um SGM. Como estas aplicações necessitam igualmente manipular dados é necessário integrar o SGM com o SGBD disponível para permitir que um Modelo possa acessar ou gerar dados cuja gerência é da responsabilidade do SGBD. Outro aspecto importante será a integração em termos de uso isto é, permitir que o acesso a Dados e Modelos seja feito de forma uniforme através de interface generalizada e flexível (SGIU).

A necessidade de integração de Dados e Modelos tem sido notada recentemente e certas propostas [Liang85] visam integrar

Dados e Modelos ao nivel das consultas e uso do sistema.

Seguindo a analogia com a área de BD, os SGM podem ser divididos também em três niveis: externo, lógico e físico, correspondentes respectivamente aos três niveis de usuários previstos, a saber: Os Usuários de Informação, os Construtores de Modelos e os Construtores de Ferramentas.

No nivel externo os Modelos são vistos de forma a refletir a visão do usuário de informação a quem interessa fundamentalmente as saidas que atendem às entradas disponiveis e às necessidades existentes. Essa visão externa pode ser representada, no modelo relacional, como duas relações - INPUT e OUTPUT - do tipo:

```
INPUT( Nome do Modelo, Nome da Entrada),
OUTPUT(Nome do Modelo, Nome da Saida).
```

Considerando, por exemplo, o Modelo Vendas dado por:

Faturação = Preço * Quantidade

o seu esquema externo seria, usando estas relações, representado por:

INPUT		OUTPUT	
Nome do Modelo	Nome da Entrada	Nome do Modelo	Nome da Saida
Vendas	Preço	Vendas	Faturação
Vendas	Quantidade		

que é possivel armazenar numa base de dados relacional facilitando a sua manipulação e integrando-a com os dados que poderão pertencer a essa base. Outra vantagem é isolar este nivel dos niveis lógico e físico pois qualquer mudança terá um impacto muito limitado neste nivel.

No nivel lógico os Modelos não são mais vistos como dados mas sim como subrotinas e tratados como programas escritos em linguagem de muito alto nivel. Neste nivel devem ser definidas duas linguagens, uma para definição (LDM) e outra para manipulaçao (LMM). A especificação de um Modelo com essas linguagens terá a seguinte configuração lógica conforme sugerido em [Liang85]:

```
NAME: <Nome do Modelo>
OUTPUT: <Saidas,..>
INPUT: <Entradas,...>
TOOL: <Opcional,..>
MODEL: <Opcional...>
BEGIN
USE <Tool1>
  IF <Condição> THEN
     LINK <Modelo>
  ELSE
     USE <Tool2>
END
```

que no caso do exemplo anterior (Modelo Vendas) seria:

```
NAME: Vendas
OUTPUT: Faturação
INPUT: Preço, Quantidade
BEGIN
   Faturação = Preço * Quantidade
END.
```

Analisando esta forma de organização verifica-se que quaisquer mudanças a nivel de ferramentas não afetam este nivel bem como mudanças na sequência lógica de computação de ferramentas ou Modelos não afeta o esquema externo já apresentado.

No nivel fisico é mantido um banco de subrotinas que podem ser usadas através de um único comando USE. As ferramentas não podem se comunicar entre si e não devem conter informação relativa à formulação lógica dos Modelos.

As principais vantagens desta integração residem na possibilidade de adaptação às caracteristicas dos diferentes usuários e na independência e modularidade que é proporcionada pelos niveis considerados. Por outro lado garante também a integração, ao nivel externo, com o SGBD, pois, os usuários de informação farão consultas de forma semelhante nos dois casos.

Por outro lado para que os sistemas sejam adaptados adequadamente aos usuários finais, com perfis possivelmente diferentes, é preciso que as interfaces possam ser definidas, prototipadas, avaliadas e ajustadas. São as interfaces que controlam as aplicações que por sua vez são responsáveis pela execução de modelos e manipulação de dados verificando-se ser necessário não só integrar as interfaces com os modelos como também com os dados.

Um caso tipico de aplicação onde tal é necessário são os SAD, estes sistemas procuram apoiar tomadores de decisão na resolução de problemas pouco estruturados e são fundamentalmente compostos por (1) Banco de Dados, (2) Base de Modelos e (3) Interface de usuário que, combinados adequadamente, formam um SAD especifico para certa classe de tomadores de decisão. Um SAD é, em geral, obtido de um ambiente Gerador de SAD (GSAD) que contèm facilidades como os SGBD, SGM e SGIU e que permite configurar diferentes SAD especificos para diferentes classes de usuários e aplicações.

Na fig.3 ilustramos um ambiente GSAD onde os vários

Sistemas de Gerência estão representados.

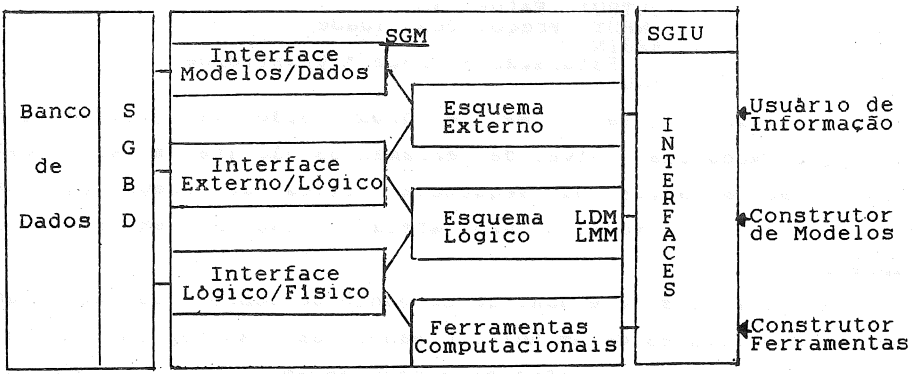


fig. 3 - Um ambiente GSAD integrado.

Para que SAD específicos sejam configurados a partir de um GSAD, este deve ter as qualidades necessárias para tornar disponíveis a diferentes Tomadores de Decisão, possivelmente com perfis diferentes, os vários modelos e dados existentes na Empresa. O ambiente GSAD deve portanto ter possibilidade de definição, armazenamento e manipulação de Modelos e Dados bem como de Diálogos que podem ser ajustados aos perfis dos usuários em questão. Se disposermos de um SGBD, um SGIU e um SGM o desenvolvimento deste tipo de aplicações só será facilitado se estes sistemas estiverem integrados, pois, no caso contrário, será necessário desenvolver interfaces para comunicação entre eles e perder-se-á a vantagem de simplificar o processo de desenvolvimento.

IV - UMA REPRESENTAÇÃO INTEGRADORA.

Para conseguir uma integração real entre SGM, SGBD e SGIU não basta, no entanto, olhar para as semelhanças entre conceitos que foram estabelecida anteriormente. Apesar dessas semelhanças não é evidente como poderão ser tratados de forma uniforme entidades tão distintas como são Dados, Modelos e Diálogos.

Vimos, na proposta de T.Liang [Liang85], que os modelos poderiam ser vistos, pelo menos ao nível externo, como relações e através de uma representação única (relações) era feita a integração destes com os Dados. Esta proposta não contempla, no

entanto, o caso dos diálogos, pois limita-se a considerar apenas uma linguagem para consultas aos dados e modelos sem atender à necessidade de integração com o SGIU.

Para estender esta proposta, promovendo a integração de Dados, Modelos e Diálogos será necessário encontrar uma forma de representação suficientemente flexível que torne viável gerenciar estes três componentes num único ambiente, facilitanto a posterior contrução de ambientes específicos de desenvolvimento de aplicações como, por exemplo, um GSAD.

Esse ambiente genérico para gerência de Dados, de Modelos e de Interfaces com o Usuário seria um Sistema de Gerência de Banco de Dados Não Convencional que teria funções para armazenamento e acesso a entidades dos três tipos.

Um paradigma que se mostra adequado à construção de tal sistema é a orientação para objetos. Neste paradigma as entidades são vistas como objetos capazes de evoluir ao longo do tempo e interagir entre si.

Um objeto é então uma forma de representação capaz de modelar simultaneamente os aspectos estáticos e dimâmicos das entidades a representar, através da definição, respectivamente, dos seus atributos e do seu comportamento face aos eventos que podem ocorrer durante a sua existência.

A criação de um sistema de Banco de Dados orientado para Objetos parece a melhor resposta para a construção de um Sistema Generalizado de Gerência de Dados, de Modelos e de Interfaces com o Usuário e dará suporte ao posterior desenvolvimento de aplicações onde estes três aspectos estão presentes simultaneamente. Um sistema de gerência deste tipo está sendo desenvolvido no projeto EITIS do Departamento de Informática da PUC-RJ [Melo 87, 88].

V - CONCLUSOES.

Este trabalho abordou o problema de integração de aspectos de gerência de Dados, Modelos e Interfaces, em áreas de aplicação onde os três aspectos podem ser vistos como recursos que podem e devem ser gerênciados de maneira centralizada como é o caso dos SAD.

A analogia com a área de Gerência de Banco de Dados foi

considerada como base para o estabelecimento de um enfoque comum tanto nos aspectos de Administração como nos referentes à Arquitetura para estes Sistemas de Gerência, tendo-se mostrado a equivalência que é possível estabelecer do ponto de vista dos conceitos básicos entre sistemas de gerência de Dados, de Modelos e de Interfaces com o usuário. No entanto, se estas semelhanças indicam ser possível a construção de um Sistema Generalizado de Gerência de Dados, de Modelos e de Interfaces com o Usuário, nada adiantam sobre qual a melhor representação que garante a construção de tal sistema.

Foi proposto então, o paradigma de objetos como forma de integrar, do ponto de vista da representação, entidades tão diversas como são os Dados, os Modelos e os Diálogos.

BIBLIOGRAFIA

[Blann86] - R.W.Blanning, "An Entity-Relationship Approach to Model Management", Decision Support Systems, N.2, 1986.

[Braz 88] - Maria Braz, "Sistemas de Gerência de Modelos", Monografia do Departamento de Informática da Puc-RJ, 1988.

[Dolk 84] - D.R.Dolk, B.R.Konsynski - "Knowledg Representation for Model Management Systems", IEEE Transations on Software Engineering, Vol.SE-10, N.6, 1984.

[Dolk 85] - Daniel Dolk, Benn Konsynski -"Model Management in Organizations", Information & Managemente, N.9, 1985.

[Dolk 86] - Daniel R. Dolk -"Data as Models: An Approach to Implementing Model Management", Decision Support Systems, N.2 1986.

[Liang85] - T.Liang - "Integrating Model Management with Data Management in Decision Support Systems", Decision Support Systems, N.1, 1985.

[Melo 87] - Rubens Melo - "EITIS - Um ambiente de ferramentas integradas para desenvolvimento de sistemas interativos", I Simpósio de Engenharia de Software, Petrópolis, 1987.

[Melo 88] - R.Melo, R.Lanzelotte, M.A.Rezende, P.Gualandi - "OMS-EITIS - Um sistema de Gerenciamento de Objetos para o projeto EITIS" , 1988, a ser publicado.

[Rozo 88] - M.A.Rozo - "Ambiente integrado para Sistemas Interativos: Criação de Sistemas de Apoio à Decisão", Tese de Mestrado PUC-RJ, 1988.

Capítulo 5

Ingeniería de Software

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

UM ESTUDO SOBRE O TRATAMENTO DA ESTRUTURA LOGICA DE DOCUMENTOS

José Valdeni De Lima[1]

Université Joseph Fourier-IMAG-LGI

ABSTRACT

Este trabalho tem como objetivo principal a especificação de alguns critérios para o armazenamento e o acesso de documentos, visando sua estrutura lógica, dentro de um Sistema de Gerência de Banco de Dados. Os documentos aqui considerados são documentos generalizados: compostos de imagens, textos, gráficos, voz, etc. Na bibliografia americana, este tipo de documento é conhecido pelo nome de "Multimedia". O tratamento da estrutura lógica de documentos facilita a troca de documentos entre duas estações de trabalho, facilita a tarefa de edição e, finalmente, permite consultas mais inteligentes aos documentos armazenados. Por exemplo:<<Qual o nome do remetente da carta que trata sobre o aumento dos preços dos produtos plásticos?>>. O tratamento da estrutura lógica aqui apresentado é função de dois parâmetros: as restrições declaradas e o mecanismo adotado de conformidade a estas restrições. O mecanismo de conformidade dita a aceitação ou não aceitação do documento pelo Banco de Dados.

1 Professor da Universidade Federal do Rio Grande do Sul (Brasil) e bolsista CAPES, afastado para Doutoramento na Université Joseph Fourier.

1. Introdução

Os documentos desempenham um papel importante na troca de informações. Por esta razão, muito esforço foi dispensado pelos comitês internacionais de padronização para criar "standards" capazes de trocar documentos entre sistemas abertos. Como exemplos destes esforços podem ser citados: CCITT T.6 (Photographic Information Facsimile Group 4), CCITT T.73 (Restricted Layout Structure - T.61 et T.6) et ECMA 101 (ODA-Office Document Architecture). Este último documento, criado pelo grupo TC29 da ECMA (European Computers Manafacturers Association) [ECM85], define o padrão ODA/ODIF, que permite a troca de documentos considerando-os como unidades contendo suas estruturas lógicas e/ou de formatos.

Segundo Horak [Hor85] a compreensão de uma estrutura comum é fundamental quando da troca de documentos entre estações de trabalho cooperantes dentro de um ambiente de escritório. Para certas aplicações de escritório, é marcante a necessidade de tratamento da estrutura lógica dos documentos. Por exemplo, oconsideremos a manipulação de cartas compostas semanticamente de três partes: cabeçalho, corpo e conclusão e, o cabeçalho contendo os nomes de remetente e destinatário. E evidente que este aspecto é indispensável para responder a consultas tais como: <<Qual é o nome do remetente da carta que trata do preço dos computadores?>>.

Além disso, na área de pesquisa inteligente de informações em documentos bibliográficos, é perfeitamente desejável o conhecimento de pelo menos alguns elementos da estrutura lógica. Por exemplo, a pesquisa de um artigo, por autor, dentro dos anais de uma conferência, necessita conhecer pelo menos a lista destes autores para que seja possível responder consultas do tipo: <<Quais são os artigos escritos por Jose?>>.

Tendo en vista estes motivos, nosso trabalho começa citando algumas propriedades que podem influenciar no armazenamento e na recuperação dos documentos. Estas propriedades são, na verdade, restrições sobre a estrutura lógica, que são declaradas na especificação do tipo de um documento. Finalmente são identificados vários mecanismos de tratamento de estrutura lógica, em função das restrições declaradas e

estratégia utilizada para garantir a conformidade destas restrições.

2. Restrições sobre a estrutura lógica dos documentos

Com relação às restrições sobre a estrutura lógica de um documento, são identificados três casos:

- Restrições de existência de elementos;
- Restrições restritas sobre a hierarquia dos elementos;
- Ausência de restrição.

2.1 Restrições de existência de elementos

Na verdade, as restrições quanto à exitência (facultatividade, obligatoriedade...) podem ajudar na escolha de uma estrutura de armazenamento, garantir a integridade dos elementos correspondentes aos diferentes tipos de dados e facilitar a recuperação dos documentos armazenados.

Existem muitas restrições de existência e agora serão descritas algumas delas:

a. Lista de elementos possíveis de pertencer à estrutura lógica. Esta restrição é contraditória com qualquer outra restrição sobre a hierarquia dos elementos. Ela simplesmente informa a possibilidade de existência dos elementos, sem nenhuma informação sobre a posição hierárquica dos mesmos. Ela deve ser usada conjuntamente com outras restrições de existência. Por exemplo, as propriedades dos elementos tais como: obrigatório, facultativo, tipo de base, cardinalidade, etc são consideradas restrições de existência.

b. Restrição de existência obrigatória. Esta propriedade é importante pois ela é utilizada como restrição de aceitação dos documentos assegurando uma certa padronização dos documentos armazenados. Com relação à estrutura interna de armazenamento das ocorrências dos documentos contendo elementos com esta restrição, é facilitada a alocação de espaço. Esta maneira flexível de se declarar restrições é extremamente interessante.

Normalmente trabalhamos com documentos onde sua estrutura não é totalmente conhecida e, além disso, a recuperação dos documentos armazenados é feita levando em conta somente certos elementos. Rabitti em [Rab85] afirma que uma das maiores ênfases que se poderia dar aos modelos de documentos generalizados é a definição de tipo flexível. Esta definição flexível é possível através de especificação de restrições como as descritas neste trabalho.

Todos os exemplos de declarações de restrições sobre a estrutura lógica apresentados no presente trabalho utilizarão uma sintaxe próxima à sintaxe da linguagem FMQL especificada em [OIS87a]. Por exemplo a declaração de obrigatoriedade de presença do elemento "Destinatário" de uma "Carta" pode ser expressa da seguinte maneira:

[Carta,Content:ODA(*Destinatário)]

c. Restrição de existência facultativa. Esta restrição torna a declaração de um tipo de documento mais flexível. Por exemplo, a declaração do fato de uma carta possuir facultativamente anexos, ou que seu remetente seja facultativo, permite a aceitação de cartas anônimas com ou sem anexo. A declaração deste fato pode ser expressa da seguinte maneira:

[Carta,Content:ODA(remetente,anexo)]

d. Restrição do tipo de base de um elemento. Este tipo de restrição é importante no sentido de que pode manter a integridade semântica dos elementos correspondentes aos diferentes tipos de dados que podem compor o documento. Na realidade, parece interessante manter a possibilidade de especificar que um elemento é de um tipo determinado. Por exemplo, "abstract" é do tipo Texto ou "foto" é do tipo Imagem. Outro exemplo, mais simples, é a declaração de cartas, contendo o elemento "remetente" do tipo "string". A restrição de tipo "string" é importante pelas seguintes razões:

- A estrutura interna de armazenamento deve conter campos, identificados por "remetente", capazes de armazenarem valores do tipo "string".
- Todos os valores armazenados devem ser obrigatoriamente do tipo "string".
- A recuperação do documento pode utilizar explicitamente o elemento "remetente".

Os tipos de dados dos elementos de um documento são os tipos de base convencionais mais os dados do tipo Texto, Imagem, Gráfico e Voz.

e. Restrição de cardinalidade. Esta restrição normalmente permite informar ao sistema que o elemento tem um número mínimo e um número máximo de valores. A especificação desta restrição pode ser feita através de dois limites: inferior e superior. Um exemplo, utilizando a sintaxe próxima da linguagem FMQL, é dado abaixo e especifica que um documento pode ter zero ou vários autores:

[Document:Document,Content:ODA(autor(0,*))]

A indicação de zero como o número mínimo de valores para um elemento é interpretada pelo servidor OIS que este elemento pode não existir em certas ocorrências de um tipo de documento. Informações sobre o servidor OIS podem ser encontradas em [OIS86a,OIS86b,OIS86c,OIS87b]. Esta interpretação pode ser contraditória com a restrição de obligatoriedade. Na verdade, as diferentes interpretações existentes apresentam uma analogia com os valores nulos referenciados por [AG87,GZC87]. A interpretação de zero valores, para um elemento que existe, não é contraditória com a restrição de obrigatoriedade. Desta forma, se o valor do elemento não existe, é possível modelá-lo em 3 (três) situações distintas:

- O elemento lógico existe sem nenhum valor. Por exemplo, um livro de um autor desconhecido tem um autor, mas ninguém conhece seu nome.

- O elemento lógico existe, mas seu valor não é conhecido neste momento. Por exemplo, o livro cujo título é "Le Parfum" sabe-se que tem um autor, mas no momento não se conhece seu nome.

- O elemento lógico não existe. Por exemplo, um manual técnico não tem autor.

No caso de existência do valor do elemento, são duas as possibilidades de modelagem:

- O elemento lógico existe com um único valor. Por exemplo, o livro "Introdução à Linguagem PASCAL" tem uma autora: V.L.S. de LIMA.

- O elemento lógico existe com vários valores. Por exemplo, o livro "Bases de Données et Systèmes Relationnels" tem dois autores: C. DELOBEL e M. ADIBA.

f. Restrição de proibição. Esta restrição pode, talvez, ser desconsiderada. Ela poderia ser útil quando da especialização de um documento. Por exemplo, "cartas-sem-assinaturas" contendo todas as cartas sem o elemento "assinaturas". Na verdade, este problema poderia ser resolvido de outra maneira, via programação.

g. Restrição de elementos alternativos. Esta restrição é útil se utilizada para especificar a estrutura lógica dos documentos do tipo formulário, onde são conhecidos os elementos alternativos segundo o seu preenchimento. Por exemplo, o elemento "nome-de-solteira" existe para as pessoas do sexo feminino não casadas; jamais existe para as pessoas do sexo masculino. Em compensação, o elemento "situação militar" só existe para as pessoas do sexo masculino. A restrição de elementos alternativos também é útil no caso de sua utilização juntamente com as restrições sobre a hierarquia dos elementos. Isto implica a possibilidade de especificar estruturas genéricas mais gerais, que podem aceitar documentos com estruturas variadas. Por exemplo, o artigo de um congresso pode ser um artigo completo ou um resumo e, como consequência, com estruturas diferentes.

2.2 Restrições sobre a hierarquia dos elementos

a. Restrição restrita sobre a hierarquia dos elementos. Esta restrição é mais rica no sentido de que ela reflete um conhecimento hierárquico da estrutura lógica do documento. Por exemplo, o elemento autor é composto de nome, prenome e endereço, que por sua vez é composto de nome da rua, número e bairro.

b. Restrição sobre o nível dos elementos. Esta restrição é importante para o aumento de performance quando da pesquisa do elemento dentro da estrutura hierarquica. Entretanto, na prática, é difícil de dispor desta informação, pois as estruturas dos documentos variam muito. Por exemplo, o capítulo pode estar no primeiro nível de uma estrutura de um livro mas, dentro de um outro, organizado em partes I e II, os capítulos estão no segundo nível.

c. Restrição de recursividade. Um elemento recursivo acarreta um ciclo sobre o modelo conceitual. Isto significa que o elemento é composto dele mesmo ou de um outro que o compõe. Por exemplo, seção composta de seção que pode, por sua vez, ser composta de novas seções. Para o tratamento de documentos dentro de um servidor de informações, a restrição de recursividade é vista como dispensável, principalmente se levado em conta o modelo ODA. Esta constatação se deve ao fato do modelo ODA quando da representação de um elemento recursivo, notar explicitamente as ocorrêcias do mesmo através de numeração progressiva. Por exemplo: seção 1, seção 1.1,... Tanto o servidor de informação MULTOS [CYT86] como o OIS [OIS87c] levam en conta o modelo ODA.

2.3 Ausência de restrição

A ausência de restrição implica a estratégia de sistema aberto. A ausência de restrição será descrita no decorrer deste trabalho, mais precisamente, na seção sobre a ausência de mecanismo de tipo.

3. Mecanismos de conformidade às restrições

São identificados três mecanismos de conformidade às restrições:

- Mecanismo de tipo forte.
- Mecanismo de tipo fraco.
- Ausência de mecanismo de tipo.

3.1 Mecanismo de tipo forte

A utilização deste mecanismo de tipo acarreta uma estratégia de sistema fechado. Isto significa que o sistema só aceita documentos onde todos os elementos que compõem a estrutura lógica sejam declarados. Este mecanismo é interessante quando se trabalha com documentos onde, pelo menos, são conhecidos todos os elementos de sua estrutura lógica. Observe que não é obrigatoriamente necessário conhecer a estrutura hierárquica entre os elementos, se não for declarada restrição sobre a mesma. Este mecanismo se adapta perfeitamente ao tratamento de documentos onde sua estrutura lógica é fixa e determinada (como, por exemplo, os formulários).

3.2 Mecanismo de tipo fraco

Este mecanismo é o mais próximo da realidade pois ele aceita documentos de uma maneira flexível segundo a vontade do usuário. Este mecanismo permite a aplicação de uma estratégia parcialmente fechada acarretando um tratamento flexível da estrutura lógica dos documentos manipulados pelo sistema. Aqui um documento é aceito mesmo se todos os elementos de sua estrutura lógica não são conhecidos. O importante aqui é o reconhecimento dos elementos declarados como obrigatórios.

3.3 Ausência de mecanismo de tipo

Não se pode utilizar um mecanismo de tipo quando nenhuma restrição sobre a estrutura lógica é declarada. Desta forma a única estratégia possível de se aplicar é a estratégia de sistema aberto. Esta estratégia aceita todos os tipos de documentos, sem se preocupar com a estrutura lógica dos mesmos. A principal desvantagem desta estratégia é a falta de padronização dos documentos aceitos pelo sistema. Como consequência, não é possível responder a consultas do tipo: <<Quais são os

documentos cujas estruturas lógicas contenham "introdução" e "conclusão"?>>.

A utilização destes mecanismos acarreta vantagens e inconvenientes, dependendo das restrições declaradas sobre a estrutura lógica dos documentos. As próximas seções deste trabalho analisarão a aplicação destes mecanismos quando da utilização de restrições sobre os elementos da estrutura lógica considerando ou não a hieraquia entre eles.

4. tipo forte e lista de elementos

O mecanismo de tipo forte tem como principal inconveniente a falta de flexibilidade para aceitar documentos onde não são conhecidos todos os elementos de sua estrutura lógica. Este mecanismo funciona muito bem para os documentos que possuem estruturas lógicas conhecidas e regulares (por exemplo, formulários). No entanto, para os documentos onde as estruturas lógicas não são regulares e definidas, este mecanismo pode causar problemas. Segundo Rabitti [Rab84] o conceito de tipo poderia ser utilizado no modelo documento, se ele permite um alto grau de flexibilidade com relação à estrutura lógica dos documentos.

A declaração da estrutura lógica como uma lista de elementos é simples, no sentido de que ela não leva em conta a estrutura hierárquica entre estes elementos. Todas as restrições citadas anteriormente podem ser utilizadas sobre os elementos da lista.

Com este mecanismo de tipo, um documento pode ser recusado pelo sistema se contém pelo menos um elemento não especificado na lista. Suponhamos que uma carta tenha sido declarada composta obligatoriamente dos elementos "destinatário", "remetente" e "assinatura" e, opcionalmente, dos elementos "introdução" e "conclusão". Neste caso, a carta especificada abaixo é recusada pelo sistema pelo simples fato da presença do elemento "corpo", que não tinha sido declarado.

Carta(destinatário,remetente,assinaturas,corpo)

Outro motivo que pode causar a recusa de um documento é a falta de

um elemento declarado como obrigatório. Por exemplo, as cartas abaixo são recusadas:

Carta(destinatário,remetente,conclusão)

Carta(destinatário,assinatura)

VANTAGENS:

a. O conhecimento da hierarquia dos elementos da estrutura lógica é dispensado.

b. As necessidades de aplicações que manipulam documentos padrões tais como certos tipos de cartas, formulários, etc são satisfeitas plenamente.

c. A análise semântica dos comandos de manipulação é feita, de uma maneira mais completa, pois todos os elementos possiveis de serem referenciados são declarados e, portanto, conhecidos.

INCONVENIENTES:

a. O conhecimento de todos os elementos que compõem a estrutura lógica é exigido. Isto não significa conhecer a posição dos elementos dentro da estrutura hierárquica ou melhor a hierarquia entre estes elementos.

b. A estrutura hierárquica entre os elementos não é conhecida. Como conseqüência, não podem ser respondidas consultas do tipo:<<Quais são os títulos dos capítulos do livro x?>>. Além disso, é impossível saber se um elemento título declarado é de um livro ou de um capítulo.

5. tipo fraco e lista de elementos

O mecanismo de tipo fraco é mais flexível que o mecanismo de tipo forte, e parece mais próximo das situações reais. Na verdade, os documentos encontrados no mundo real têm estruturas freqüentemente complexas e difíceis de serem completamente identificadas. Por exemplo, é fácil deduzir que uma "Carta" tem, normalmente, "destinatário" e "remetente". No entanto, outros elementos tais como "logotipo", "introdução" ou "conclusão" não são evidentes. Existem elementos que dependem da aplicação ou do do tipo de documento. Por exemplo, uma carta informando o preço de um produto pode conter uma tabela com os prêços dos elementos que compõem o produto, bem como gráfico contendo a evolução dos preços dos produtos concorrentes nos últimos seis meses.

A principal diferença entre os mecanismos de tipo forte e tipo fraco diz respeito à declaração dos elementos: o primeiro exige a declaração de todos os elementos da estrutura lógica, enquanto, o segundo, aceita documentos contendo elementos não declarados. De forma semelhante ao mecanismo de tipo forte são também recusados os documentos nos quais faltam elementos declarados como obrigatórios. Por exemplo, se uma carta é declarada como composta, obrigatoriamente, dos elementos "remetente" e "destinatário", pelos mesmos motivos já citados as seguintes cartas são recusadas:

Carta(destinatário,assinaturas)

Carta(destinatário,corpo)

VANTAGENS:

a. O conhecimento da estrutura hierárquica dos elementos é dispensado.

b. Não é necessário conhecer todos os elementos componentes da estrutura lógica do documento manipulado.

c. Funciona bem se o usuário conhecé os elementos sujeitos a recuperação futuras.

INCONVENIENTES:

a. A estrutura hierárquica entre os elementos não é conhecida. Como consequência, não podem ser feitas consultas do tipo: <<Quais os titulos dos capítulos e das seções do livro x?>>.

b. A análise semântica dos elementos não declarados é impossível de ser feita.

6. Tipo forte e restrições sobre a hierarquia entre elementos

As restrições sobre a hierarquia entre elementos da estrutura lógica de documentos são mais ricas, no sentido de que dão um conhecimento hierárquico da referida estrutura. A utilização destas restrições com o mecanismo de tipo forte exige o conhecimento antecipado de todos os elementos da estrutura lógica dos documentos que serão manipulados pelo sistema, bem como, suas posições dentro da referida estrutura.

VANTAGENS:

a. A estrutura arborescente dos elementos é conhecida completamente pelo sistema.

b. A análise semântica dos comandos de acesso pode ser feita de uma maneira mais clara e completa.

c. Funciona bem para os tipod e documentos onde as estruturas lógicas são totalmente conhecidas.

INCONVENIENTES:

a. E obrigatório declarar todos os elementos e a hierarquia entre eles. Este nível de detalhe nem sempre énecessário.

7. Tipo fraco e restrições sobre a estrutura hierárquica

A principal vantagem desta combinação é a flexibilidade oferecida para especificação da estrutura lógica dos tipos de documentos que serão manipulados. Desta maneira, não é necessário o conhecimento completo dos elementos, nem da hierarquia entre seus componentes.

VANTAGENS:

a. Permite a especificação parcial da estrutura lógica dos documentos que serão armazenados e manipulados posteriormente. Como conseqüência, o usuário pode especificar somente os elementos que lhe interesam, tornando o sistema flexível e sob medida.

b. Permite a especifição parcial da arborescência da estrutura lógica.

INCONVENIENTES:

a. A análise semântica dos elementos não especificados é impossível de ser feita.

8. Ausência de mecanismo de tipo e de restrição

Na verdade, o mecanismo de tipo e as restrições sobre os elementos da estrutura lógica estão intimanente ligados. Isto quer dizer que é impossível a aplicação de um mecanismo de tipo se nenhuma restrição é declarada. Este caso caracteriza a estratégia de um sistema aberto onde todos os documentos são aceitos independentemente de suas estruturas lógicas.

Esta estratégia abre várias possibilidades de tratamento da estrutura lógica, as quais exigiriam uma análise mais detalhada.

9. Conclusão

Neste artigo foi abordado a modelagem flexível da estrutura lógica dos documentos. Como visto, esta modelagem poderia levar em conta os elementos independentemente de sua posição dentro da hierarquia. Neste caso, uma lista simples dos elementos seria suficiente. Mas, também se poderia declará-los conforme a estrutura hierárquica. Neste caso, as restrições sobre a hierarquia dos elementos seriam necessárias. Além disso, foram descritas várias outras restrições que poderiam ser aplicadas a cada elemento declarado da estrutura lógica.

Foram também apresentados os mecanismos de conformidade a estas restrições, que ditam, na realidade, a estratégia de aceitação dos documentos. A estratégia de sistema fechado é resultado da aplicação de um mecanismo de tipo forte, enquanto que a estratégia de sistema parcialmente fechado é resultado da aplicação de um mecanismo de tipo fraco. A impossibilidade da aplicação de um dos dois mecanismos, devido, por exemplo, à falta de declaração de restrições, acarreta uma estratégia de sistema aberto.

Como conclusão, pode-se considerar que o mecanismo de tipo fraco é o mais conveniente e próximo da realidade, pois permite uma aceitação flexível dos documentos pelo sistema, sem o conhecimento prévio de suas estruturas lógicas. Além disso, esse mecanismo permite a especificação de alguns elementos, contribuindo desta maneira para uma certa padronização dos documentos armazenados dentro do servidor de informações.

Agradecimentos

O autor agradece a todas as pessoas do projeto DOEOIS que contribuiram para realização deste trabalho em especial aos Srs. Mauricio Lopez e Jean-Pierre Martin.

10. Bibliografia

[AG87] ABITEBOUL, Serge et GRUMBACH, Stéphane, Bases de données et objets structurés. Technique et Science Informatiques v.6 n.5, v.6 1987.

[CYT86] CHRISTODOULAKIS S. et al., Office Document Retrieval in MULTOS, ETW 1986.

[ECM85] ECMA/TC29/85 Standard ECMA-101 Office Document Architecture. September 1985.

[GZC87] GUTING Ralf Hartmut et al., An Algebre for Structered Office Documents. Research Report of Almaden Research Center, San Jose, California, March 1987, 77 pag.

[Hor85] HORAK W. Office Document Architecture and Office Document Interchange Formats: Current Status of International Standardization. IEEE Computer, october 1985, pag. 50 - 60.

[OIS87a] DOEOIS Data Definition and Data Manipulation Services. OIS-DH-01.01, april 1987.

[Rab85] RABITTI F., A Model for Multimedia Documents. In: D. Tsichristzis (ed.), Office Automation, Springer 1985,226-249.

[OIS87b] DOEOIS Future Office Information Server Functionality. 3A3/3A5 Deliverable, March 1987.

[OIS86a] DOEOIS Bibliographical Annex Bibliographical Informations and Standards in OIS: Overview. 1B1 Deliverable, January 1986.

[OIS86b] DOEOIS Review of Data Models and Data Management Systems. 1B2 Deliverable, January 1986.

[OIS86c] DOEOIS External Functional Interface for an Office Information Server. A Description. 2B1 Deliverable, September 1986.

[OIS87c] DOEOIS Document Representation in an Office Information Server. 2B3 Deliverable, 1987.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

HYPERMEDIA:
NUEVAS HERRAMIENTAS PARA EL CONOCIMIENTO

Francesca Greselin, Riccardo Ferrari*.*

Laboratorio Electrónica Digital
Facultad de Ciencias y Tecnólogia
Universidad Católica
"Nuestra Señora de la Asunción"
Asunción, Paraguay

En este artículo queremos comentar la esencia de los hypertextos, discutiendo sus ventajas y desventajas principales.
También queremos mostrar que esta nueva tecnología abre algunas perspectivas muy excitantes, particularmente en la utilización de la computadora como medio para comunicar y elaborar ideas.
El L.E.D. de la U.C. de Asunción está en la fase de preparación de un proyecto de investigación en cooperación con la Universidad de Milano (Italia) para la aplicación de los hypermedia a sistemas didácticos y organizativos. De los estudios preliminares brota la presente profundización de esta nueva e interesante manera de estructurar la información.

---

* De AVSI (Asociación de Voluntarios para el Servicio Internacional), entidad de cooperación internacional italiana, no gubernamental, reconocida por el gobierno italiano y el Parlamento Europeo, con la cual ha sido desarrollado el Laboratorio de Electrónica Digital.

Los Hypertextos, como nuevos sistemas de información electrónica, superan los límites humanos en este campo proveyendo mecanismos de almacenamiento compacto y búsqueda rápida de enormes volúmenes de datos textuales, numéricos, o visuales. La importancia de estos sistemas está en su capacidad potencial de aumentar y amplificar el intelecto humano. Necesitamos esta capacidad por el crecimiento exponencial, el aumento de la complejidad y la naturaleza multidisciplinaria del conocimiento científico (ya sea en sentido estricto, de la ciencia pura, como aplicado: económico, médico...).
Los primeros sistemas de almacenamiento/recuperación de informaciones buscaban en grandes bases de datos para encontrar ítems que satisfacían a queries booleanas o por llave. Un segundo paso fue proveer el acceso a los recursos de información mediante jerarquías de selección de menús. Una tercera estrategia, llamada data-base management system, permitió encontrar los datos necesarios a través de búsquedas lógicas de programas procedurales o de precisos lenguajes de query. Las aplicaciones de estos métodos se superponen en algunas necesidades, pero las diferencias revelan la variada filosofía de acercamiento y de utilización de la computadora.
Un nuevo enfoque - Hypertextos - se ha añadido recientemente al universo de los sistemas de información electrónica.
El término HYPERTEXTO denota escritura y lectura no secuencial, obtenida mediante conexiones (sea entre los documentos como adentro de un mismo documento) proveídas por computadora.
Un hypertexto es esencialmente un conjunto de documentos que contenien una colección de nudos conectados por *links* directos. Un nudo representa una parte de texto, y los *links* entre los

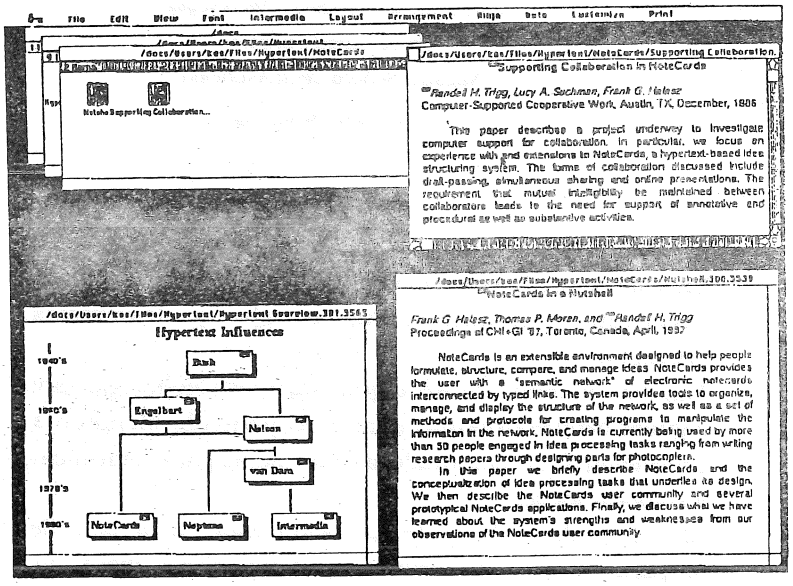


Fig.1: Presentación de dos documentos y de una visión de conjunto mediante el hypertexto INTERMEDIA. Los íconos presentes en diferentes puntos de la pantalla visualizan las conexiones existentes.

nudos son exactamente los que dan al hypertexto su aspecto no lineal.
Ventanas en la pantalla son asociadas a objetos en una base de datos y las conexiones entre aquellos objetos son proveídas sea gráficamente (en pantalla) como en la base de datos (pointers). A la vez una herramienta para el autor y un medio para el lector, un sistema hypertextual de documentos permite a los autores o a grupos de autores conectar informaciones, crear caminos a través del cuerpo del material relacionado, poner comentarios a textos existentes y crear notas para que los lectores apunten a datos bibliográficos o a los mismos textos referenciados.

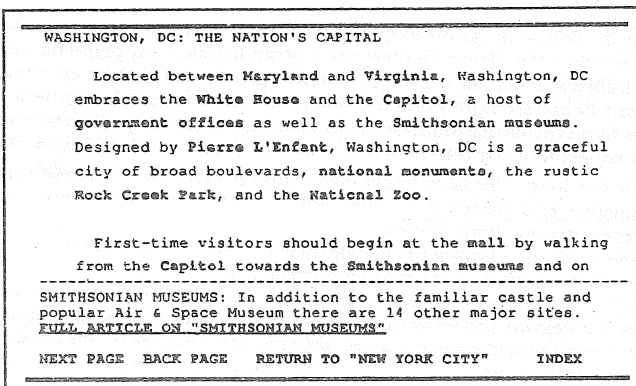


Fig.2: Esta pantalla Hyperties muestra en negrita los items de menus integrados con el texto. Estos items son selecionables por contacto con la pantalla o teclas de posicionamento. El usuario puede seguir un tópico de interés, girar páginas, volver al artículo precedente o ver el index.

Desde el punto de vista del autor, los sistema de hypertextos ofrecen la mejor generación de procesadores de palabras. Además de proveer características como el mover bloques, buscar y remplazar, y aparte del chequeo lexical y sintáctico, las herramientas de escritura de los hypertextos pueden soportar y extender el proceso de escritura con esquematizaciones (outlines) telescópicas, notas post-escritas que no afectan el texto principal, señaladores (book-marks) electrónicos y métodos de browsing.
Desde el punto de vista del lector, los sistemas de hypertextos son una nueva generación en la gestión de bases de datos. El texto entero es accesible desde múltiples perspectivas, para varios propósitos y a través de diferentes estrategias de búsqueda.
Los lectores pueden hojear a través de textos interconexos, con cross-reference (referencias cruzadas) y provistos de notas y comentarios, en una manera ordenada (guiada por asociaciones de ideas) pero no necesariamente sólo secuencial.

Por extensión de concepto, la palabra HYPERMEDIA denota la funcionalidad de un hypertexto pero con las características adicionales de almacenar gráficos de 2 o 3 dimensiones, habla digitalizada, dibujos, hojas electrónicas, imágenes de video, sonidos y animación.
A través de los hypermedia, un autor puede crear conexiones entre diagramas, textos, fotografías, video-discos, audio-grabaciones y otros.

Hypertexto e hypermedia son entonces nombres que implican capacidades básicas que incluyen:

- fácil trazado de referencias
- fácil creación de nuevas referencias
- la modularización de la información
- la consistencia de la información
- la estructuración de la información
- vistas globales y locales de la información
- documentos personalizados para cada usuario
- colaboración en la construcción de nuevos documentos
- apilación de caminos de búsqueda contemporáneamente activos.

BREVE RECORRIDO HISTORICO

La historia de los hypertextos es rica y variada porque los hypertextos no son una nueva idea exclusiva del campo de utilización de la computadora. Muchas personas han contribuido al desarrollo de este concepto y cada una de ellas parece haber tenido algo característico en su mente. Queremos rever estos investigadores y esquematizar el corazón de sus ideas en una perspectiva histórica, mientras remitimos al lector consultar la literatura específica para tener una mirada más detallada.

La utilización tradicional de fichas de apuntes (note-card) es el ejemplo más conocido de hypertexto manual. A menudo, ellas llevan referencias a otras fichas, y su pequeño tamaño permite modularizar las informaciones en trozos. El usuario puede reorganizar en cualquier momento el conjunto de sus fichas siempre y cuando una nueva información lo requiera.

Otros ejemplos de antiguos hypertextos manuales son el diccionario y la enciclopedia, vistos como grafos de nudos textuales conectados por referencias. El Talmud[1], como también los escritos de Aristóteles, son documentos en los cuales las referencias a otras partes de ellos mismos o a otras fuentes de información, constituyen la parte más considerable del texto: se revelan entonces como antiguos prototipos de representación hypertextual.

En 1945 aparece por primera vez tan claramente la exigencia de extender la tradicional noción de textos lineales para permitir organizaciones más complejas de la información.

Vannevar Bush, presidente de la Roosevelt´s Science Advisor, en un artículo titulado "As we may think" sugirió tomar como modelo los procesos mentales humanos asociativos. La elaboración del pensamiento no construye nuevas ideas una a la vez, sino más bien las desarrolla en paralelo, en diferentes niveles: cada idea contribuye a la creación de nuevas ideas o depende del acrecentamiento de las demás.

"La mente humana... opera mediante asociaciones. El hombre no puede esperar simular plenamente este proceso mental artificialmente, pero tiene que estar en condición de aprender de ello. No se puede esperar igualar la velocidad y flexibilidad con la cual la mente sigue un camino asociativo, pero tiene que ser posible acercarse a la claridad y durabilidad de los elementos recuperados de la memoria." (28)

Bush propuso el estudio de una máquina, que llamó *Memex*, que permita el browsing (hojear, recoger, recolectar datos) y el tomar notas en un sistema extensivo de gráficas y textos on-line para mejorar el sistema de la literatura científica. Su idea era construir -sobre una base hecha por una vasta colección de microfilms- un entorno en donde una librería bien grande, cualquier clase de notas personales, fotografías o esquemas gráficos pudieran aparecer en diferentes pantallas al mismo tiempo, permitiendo además de establecer conexiones útiles entre dos puntos cualesquiera de la librería entera. Aunque el *Memex* nunca fue desarrollado, la literatura acredita a Bush el mérito de haber sido el primero en pensar utilizar una máquina para almacenar conexiones entre pedazos de informaciones.

En 1945 todavía no se podía imaginar cómo implementar todo esto: los desarrollos tecnológicos alcanzados en los cuarenta años que nos separan de él, nos permiten ahora no solo valorar su idea, sino también experimentar sus potencialidades en los primeros ejemplos de hypertextos.

Pero si queremos cualificar el hypertexto como exclusivamente soportado por computadora, la historia se vuelve decididamente más breve.

Douglas Engelbart,en 1963, en el Stanford Research Institute, fue influenciado por el artículo de Bush y desarrolló un sistema sobre computadora llamado NLS (oNLine System) que daba cuerpo a muchos conceptos de los hypertextos (29). El esquematizaba un sistema que habría sido adecuado a sus necesidades de trabajo:

"...poniendo en la memoria de la computadora todas nuestras especificaciones, planes, diseños, programas, documentaciones, impresiones, memos, bibliografías, y notas de referencias, etc. y

---

[1] Talmud (hebr. talmúd,, enseñanza): Libro de los judíos, que contiene la tradición, doctrinas, ceremonias y policía que suelen observar tan rigurosamente como la misma ley de Moisés.

permitiendo desarrollar todo nuestro trabajo de bosquejo, planeamiento, dibujo, búsqueda de errores, etc. y una buena porción de nuestras intercomunicaciones vía consolas." (30)
NLS enfatizaba tres aspectos: una base de datos no lineal de textos, filtros de visión que seleccionan las informaciones desde esta base de datos y puntos de vista que estructuran el display de la misma información en la pantalla.
NLS evolucionó sucesivamente con los años, hasta ser vendido en el mercado bajo el nombre de Augment (22) y respondiendo sobre todo a las necesidades de un entorno consistente para ingenieros del conocimiento.
Durante los mismos años 1960, Ted Nelson también empezó a trabajar en Xanadu, un sistema diseñado para utilizar conexiones en organizar grandes volúmenes de literatura on-line. Fue Ted Nelson el que acuñó el término "hypertexto". En 1967 se puso a colaborar con Andy Van Dam para el Editing System en la Brown University (que se utilizó en el Houston Manned Spacecraft Center para producir la documentación del Apollo). Seguidamente, Van Dam y sus estudiantes diseñaron un otro sistema de hypertextos, el File Retrieval and Editing System (FRESS), que se utilizó para la preparación de documentos y para cursos a los estudiantes. FRESS era ya una versión potenciada para tiempo compartido por muchas terminales.
Algunos trabajos sobre hypertextos también se desarrollaron en los años '70, pero no fue nada comparado con lo que aconteció en la década siguiente. Recientemente el interés en los hypermedias se acrecentó, posiblemente por la facilidad de conseguir estaciones de trabajo personales poderosas y con pantallas de alta resolución. Tres de los mas amplios proyectos en curso son NoteCards[1] de la XeroxPARC, Neptune[2] de Textronix, e Intermedia[3] de la Universidad de Brown. Hay también productos hypermediales comerciales, como por ejemplo HyperCard [4] de Apple y Guide[5] de Owl International.

Para una más detallada exposición histórica del desarrollo de hypertextos e hypermedia, se sugiere el interesante artículo de Jeff Conklin (2).

---

[1] El objetivo de este sistema fue la experimentación misma sobre la tecnología de los hypertextos. NoteCard provee fácil creación de nudos, según 40 o 50 tipos de nudos especializados. Una interfase abierta al usuario permite también construir nuevas aplicaciones en Lisp apoyándose al último nivel de NoteCard.

[2] Fue particularmente creado con una arquitectura abierta y en niveles. Neptune separa fuertemente el front-end (la interface hacia el usuario basada en Smalltalk), desde el back-end (un server que organiza transacciones en la base de datos, llamado Hypertext Abstract Machine). Provee acceso distribuido sobre una red y diferentes clases de browser.

[3] Se desarrolló principalmente como herramienta didáctica para las clases de los profesores y para el estudio interactivo de los alumnos. El equipo de Brown se ocupó principalmente de proveer al usuario modos para gestionar la creciente complejidad del entorno de los hypertextos al formar grandes cantidades de conexiones. Propusieron también la construcción de diferentes tramas (web) de informaciones para dar visualizaciones diferentes según el contexto elegido por el usuario.

[4] Hypertexto diseñado para aparecer diferente a los usuarios a medida de sus capacidades, brinda seis niveles de posible intervención: browsing, typing, painting, authoring, scripting y 'metascripting'. Organizado en stacks (grupos de fichas), permite la visión de una sola ficha en pantalla. Cada ficha a su vez está estructurada en niveles, englobando texto, imágenes, dibujos, animación , sonido e iconos de conexión. Muy interesante es la posibilidad ofrecida a través de conexiones ejecutables.

[5] Guide permite tres tipos de conexiones: de remplazamiento (el texto de donde hubo el llamado va siendo reemplazado por el texto llamado), de notas (visualiza la nota en una apósita ventana pop-up), y de referencia (trae consigo una nueva ventana con el texto requerido). Disponible también para PC .

ASPECTOS CENTRALES DE UN SISTEMA DE HYPERTEXTOS

Los diseñadores de sistemas de hypertextos tienen que considerar el contexto de la búsqueda de informaciones. El diseño de todo el complejo tiene que atender al sistema físico, al modelo conceptual que el sistema presenta (la interface hacia el usuario), y al modelo mental que se supone el usuario vaya desarrollando con el sistema. Las decisiones de diseño que afectan la búsqueda de la información en los sistemas de hypertextos se expresan en el definir los puntos de acceso, en el crear la interface usuario, y en el proveer características (features) de estrategias de búsqueda. Estas decisiones interaccionan de manera a veces inesperada, pero un proyecto bien diseñado puede preservar potencia y flexibilidad mientras reduce complejidad de uso.
Queremos dar un vistazo general de lo que se podría llamar un "hypertexto ideal". La mayoría de los sistemas existentes tienen muchas propiedades de las que iremos delineando.

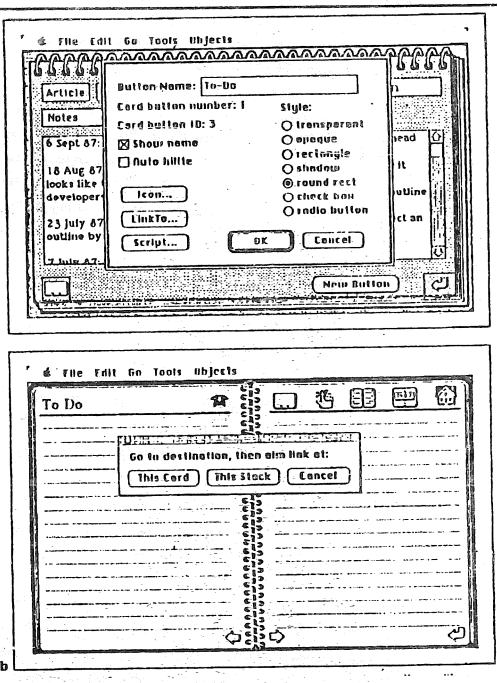


Fig.3: Conexión entre un nuevo botón y una destinación en Hypercard. Cada botón puede llamar un cuadro explicativo (a) que permite eligir la apariencia y el comportamiento del botón mismo. Se eligie la opción "LinkTo", luego, se va a destinación y se elige "This Card"(b). De esta manera el botón (renombrado "To Do") provoca Hypercard a saltar a la destinación querida.

Típicamente, diversos bloques de información se crean usando diferentes aplicaciones o editores. Un sistema de documentos de un hypermedia provee la capacidad de conectar heterogéneos bloques creados por diferentes aplicaciones como un programa de dibujo, un paquete para mapas geográficos o un editor de música.

Los sistemas primitivos permitían solamente conexiones esencialmente programadas, mientras que los sistemas más nuevos proveen la creación interactiva de conexiones como un componente esencial. Los sistemas de hypertextos difieren de los sistemas de búsqueda on-line porque alientan estrategias de búsqueda de informaciones que sean informales, personalizadas, orientadas al contexto. Entonces nuevos nudos y nuevas conexiones tienen que ser fácilmente creados por los usuarios.

Otra capacidad fundamental que debe estar presente para promover la conectividad (connectivity) es el acceso multiusuario a la información. Como mínimo, los usuarios pueden tener "read access" al documento de un colega; además se le puede permitir crear nuevas conexiones dentro del mismo documento y editarlo de una manera reglamentada. Nótese que el problema de consentir un ambiente multiusuario lleva consigo consecuencias bastante complejas relativas a los derechos de acceso pero sobretodo a la consistencia de las posibles modificaciones (p.ej.:¿qué hacer de las conexiones que apuntan a un nudo que se quiere borrar?).

Mucho valor se da a la interface para el usuario. Aunque los lenguajes de comandos ofrecen poderosos medios a los expertos, una variedad de selecciones de un menú permite a los novatos y a los usuarios ocasionales una manipulación directa y orientada a la visualización, una representación expresiva del mundo de las posibles acciones. En vez de tener que acordar comandos, el usuario puede apuntar a objetos y a acciones, viendo inmediatamente el éxito de la elección hecha, entonces reduciendo errores y aumentando la velocidad de acceso a las informaciones. Las acciones tienen que ser rápidas, incrementales y reversibles para brindar un sentido de maestría, control y confianza.

El manejo estandar de ventanas (abrir, cambiar medida, reposicionar, cerrar, traer hacia adelante o poner hacia atrás, poner al lado visualizándolas como pequeños iconos...) normalmente es soportado. La posición y la medida, como también su color y su forma en algunos casos, sirven para recordar el contexto de la ventana. Cada ventana puede contener cualquier número de íconos de conexión, que representan punteros a otros nudos en la base de datos. El ícono de conexión contiene una información breve (normalmente una palabra o un dibujo) que sugiere el contenido del nudo al cual apunta. Hacer click con el ratón en un ícono de conexión implica que el sistema busque el nudo referenciado y abra inmediatamente una nueva ventana para él en la pantalla.

Los mecanismos de búsqueda y de feedback son críticos porque estructuran el conocimiento y definen como ello sea accesible. La manera en que el conocimiento es organizado y hecho alcanzable afecta las estrategias para accederlo y entonces las prestaciones en la búsqueda de informaciones.

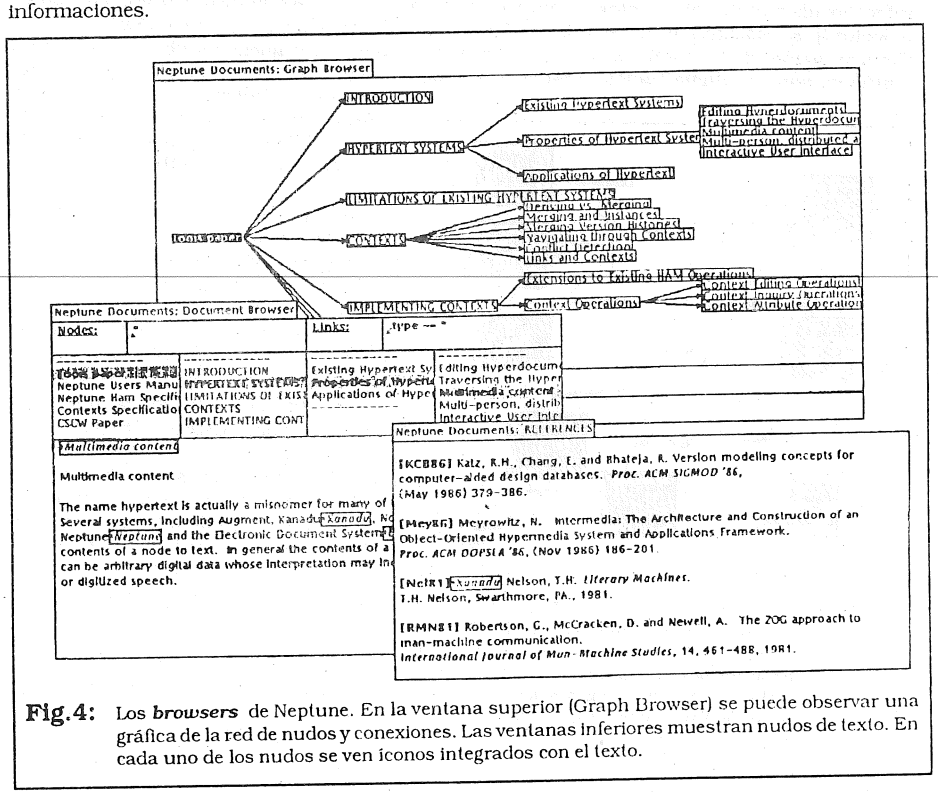


Fig.4: Los ***browsers*** de Neptune. En la ventana superior (Graph Browser) se puede observar una gráfica de la red de nudos y conexiones. Las ventanas inferiores muestran nudos de texto. En cada uno de los nudos se ven íconos integrados con el texto.

La base de datos puede ser recorrida de tres maneras:

- siguiendo las conexiones y abriendo las ventanas sucesivamente para examinar su contenido,
- buscando en la red, o en parte de ella, a través de una cadena de caracteres, una llave o un valor de atribución,
- navegando en el hypertexto usando un *browser* que dibuje la red gráficamente.

El browser es una característica importante de los sistemas de hypertextos. Mientras que la base de datos crece en tamaño y en complejidad, se vuelve penosamente fácil para el usuario perderse o desorientarse. Un browser visualiza parte o todo el hypertexto como un grafo, proveyéndole al usuario los medios esenciales para entender el entorno en que está actuando y las relaciones con los nudos más cercanos.

PROPIEDADES DE LOS HYPERTEXTOS

LOS NUDOS

Aunque la esencia de un hypertexto se expresa por el tipo de conexiones que la máquina está en grado de soportar, los nudos contribuyen de manera significativa a la definición de las operaciones que un sistema hypertexto puede proveer.

Los hypertextos invitan el autor a modularizar sus documentos en unidades (nudos) hasta expresar ideas o conceptos singulares de manera a permitir que una idea individual pueda ser referenciada en cualquier contexto y que una alternativa de sucesores de una unidad sea ofrecida al lector (p.ej. más detalle, un ejemplo, referencias bibliográficas, o el sucesor lógico).

Como ya hemos dicho, los nudos pueden representar imágenes gráficas, combinaciones de texto y gráfica, sonidos digitalizados, animación... Algunos sistemas permiten tipos diferentes de nudos, ofreciendo colores y tamaño particulares, o bien íconos específicos para cada tipo de nudo (así es en Note Cards, Intermedia e Ibis).

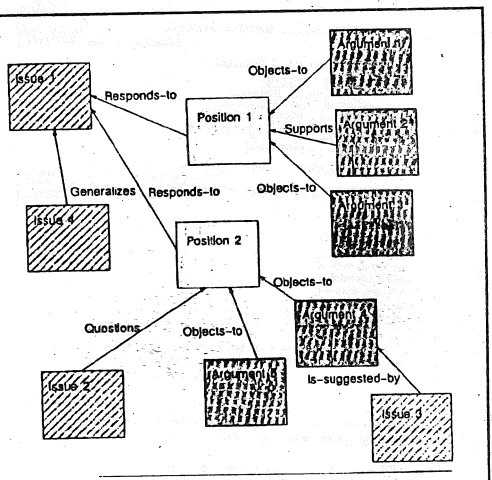


Fig. 5: La topología de la red IBIS. Cada nudo contiene informaciones sobre el tipo de nudo, la hora y fecha de creación, el autor, un breve frase que describe el contenido, un cuerpo de texto más largo con comentarios, una lista de palabras llaves, y una lista de conexiones en entrada y salida.

Para algunas aplicaciones se proyectaron nudos semiestructurados: tipos de nudos que contienen campos etiquetados y espacios para los respectivos valores. Otro mecanismo para agregar informaciones relacionadas es el nudo compuesto: muchos nudos se ponen juntos y la colección se trata como un nudo singular, con su nombre propio, tipo, versión,... Esto puede ser muy útil en situaciones en las cuales un conjunto de nudos forma una estructura a nivel más alto, como en el caso de las entradas de una tabla, o los detalles de una lista.

En lo que se refiere a la estructura de los nudos hay dos tendencias: los sistemas que prefieren tener muchos nudos pequeños (de manera de maximizar el grado de organización de la información) y los que eligen tener sólo pocos nudos de grandes dimensiones (para minimizar el atraso y la complejidad de atravesamiento de los nudos).

Hay dos maneras de implementar los

nudos en un sistema de archivos:

- cada nudo es un archivo
- todos los nudos son memorizados en un archivo grande.

El segundo método necesita un servicio cuidadoso de protección del sistema de archivos al tener muchos usuarios concurrentes.

LAS CONEXIONES

Los links pueden ser utilizados para muchas funciones, entre las cuales:
- conectar una referencia a un documento con el documento mismo
- conectar comentarios o anotaciones con el texto al cual se refieren
- proveer información de tipo organizativo (p.ej. establecer la relación entre dos trozos de texto o entre un índice y los párrafos referenciados)
- conectar dos partes sucesivas de texto, o una parte de texto con todos sus sucesores inmediatos
- conectar accesos a una tabla, o bien imágenes, a una descripción más detallada o a otras tablas o figuras.

Las conexiones pueden ser definidas por nombres, ser de diferentes tipos y poseer un rico conjunto de propiedades.
Al introducir conexiones en un sistema, se tiene que añadir un conjunto de mecanismos para su gestión, o sea para crearlas, cancelarlas, modificarlas en nombre o atributos... Hay dos métodos para conectar dos puntos en un hypertexto: conexiones de referencia (referencial links) y conexiones organizativas (organizacional links).

Conexiones de referencia.

Sirven para organizar el texto según una estructura de red (no jerárquica). Son las conexiones que más claramente caracterizan los hypertextos. Pueden unir dos puntos o regiones del texto, y tienen dos extremidades: la fuente (reference) y el destino (referent). Un punto es un icono que indica la presencia del link; una región es un conjunto de caracteres contiguos visualizado como una unidad singular (en reverse video o a través de colores diferentes del contexto).

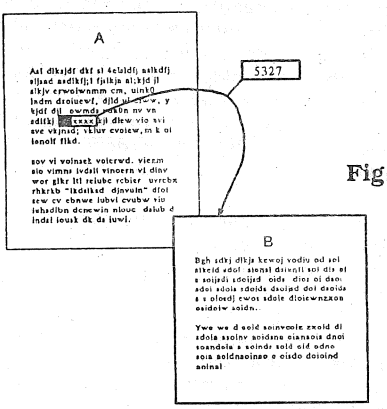


Fig.6: Un ejemplo de una conexión con un punto fuente y una región como destinación. La fuente de la conexión es un icono en el texto del documento A que contiene un identificador textual ("xxxx"). El identificador puede ser: (1) el nombre del nudo de destinación, (2) el nombre de la conexión, (3) una cadena de caracteres que no es ni el nombre del nudo ni lo de la conexión. La destinación de esta conexión es B, que es una región. La conexión tiene un nombre interno (5327) normalmente visible por el usuario.

Conexiones organizativas.
El método que utiliza estas conexiones organiza el documento en una estructura jerárquica, así que las conexiones unen un nudo padre con todos sus hijos, formando un árbol en el interior del hypergrafo (corresponden a los links 'is-a' de la teoria de las redes semánticas). Para atravesar estas conexiones normalmente hay mecanismos que se activan mediante comandos del tipo "gotoparent", " "...
Algunos sistemas de hypertextos ofrecen solamente una estructura jerárquica (p.ej. NLS, Augment, Zog, Ooutline Processors), otros solamente una estructura de red (p.ej. Xanadu, Neptune) y también hay otros que proveen los dos tipos (p.ej. Textnet, Notecards). Una ventaja de los sistemas orientados estrictamente a estructura jerárquica es la de tener un lenguaje de comandos muy sencillo para el atravesamiento: de un nudo el usuario puede moverse solamente hacia un nudo padre, hermano o hijo. Esto ayuda a resolver en parte el problema de la desorientación del usuario. Naturalmente una desventaja de tal estructura es que en fase de creación el usuario debe ya anticipar los principales criterios para los futuros accesos a la información. Las estructuras de multiples jerárquias permite una solución parcial a este problema, consintiendo subdividir el mundo de la información en diferentes subdivisiones ortogonales.

Conexiones por palabras claves (keyword)
Existe otro tipo de conexión, implicita, que utiliza palabras claves para la búsqueda de documentos o de partes de ello. De manera diferente del método explicito (que une los nudos uno a otro), la búsqueda por keywords conecta un conjunto de nudos entre sí.

LIMITACIONES EN LOS HYPERTEXTOS

Se pueden delinear dos clases de problemas: los que están relacionados con los sistemas ahora existentes y los que parecen ser derivados del concepto mismo de hypertexto.
En la primera clase podemos encontrar los atrasos en la visualización de la información conectada a un link, restricciones en los nombres o en las propiedades de las conexiones, falta o funcionamiento limitado de los browser.
A la segunda clase pertenecen dos problemas, que son fundamentales porque pueden de hecho limitar la utilidad de un hypertexto, y son la desorientación y el costo cognocitivo.

LA DESORIENTACION
Siendo el hypertexto un medio de información que permite organizar el material de una manera mucho mas compleja con respecto a los textos lineales a los cuales estamos acostumbrados, el usuario necesita saber, en cada momento, dónde está y cómo hallar otro punto que él sabe o piensa que exista en la red. Obviamente este problema también existe en los documentos tradicionales, pero en este caso, el lector tiene sólo dos posibilidades: buscar hacia adelante o hacia atrás con respecto al punto en que está. Los hypertextos ofrecen más grados de libertad, más dimensiones en las cuales moverse y entonces más posibilidad de perderse para el usuario.
Las dos principales soluciones tecnológicas para disminuir la desorientación son browser gráficos y mecanismos de interrogación/búsqueda.
Los browser permiten visualizar nudos y conexiones en un espacio 2-3 dimensional, ofreciendo propiedades útiles para una clara y visible diferenciación y manteniendo algunas similitudes con nuestro ambiente fisico (p.ej. dos objetos distintos no pueden ocupar el mismo espacio y no pueden cambiar posición si no son desplazados por el usuario).
Todavía no hay alguna precisa topología para la representación del espacio de información, así que el usuario, hasta cuando no adquiera un profundo conocimiento de las informaciones contenidas en el hyperdocumento, sigue quedándose desorientado.
También hay dificultades que surgen al querer representar un grafo cuando estén presentes un gran número de nudos o conexiones, o cuando hayan frecuentes cambios en la red. Una solución puede ser la de aplicar a la base de datos técnicas de búsqueda o de interrogación para localizar los nudos utilizando keywords, predicados lógicos y/u otra clase de filtros sobre atributos de nudos/conexiones.

COSTO COGNOCITIVO

Es difícil acostumbrarse al trabajo mental adicional necesario para crear, dar nombres apropiados y tener traza de las conexiones. Esta carga podemos llamarla "costo cognocitivo".
En el proceso de lectura, también, el usuario se encuentra con tener que decidir entre un gran número de elecciones con respecto a cuales conexiones seguir. La presencia de tales elecciones lleva consigo un cierto esfuerzo mental que no se requiere en los tradicionales textos lineales (en los cuales el autor ya definió el itinerario a seguir en la lectura del documento).

Nótese sin embargo que estos problemas (desorientación y costo cognocitivo) no aparecieron recién con los hypertextos, porque están relacionados directamente con el hecho que el cerebro humano puede crear ideas más rapidamente de cuanto las manos puedan escribirlas o la boca pueda expresarlas. Los hypertextos simplemente ofrecen un "anotador" bastante sofisticado para tratar de agarrar la riqueza, la variedad y la interrelación típicas del pensamiento creativo. Cuando esta riqueza sea necesitada los hypertextos proveen sus ventajas, mientras que en otros casos pueden mostrar sus límites.
Y aquí se abren las perspectivas de las investigaciones futuras. Apuntando a mejorar las prestaciones mediante perfeccionamientos de las técnicas de filtración de la información y de su representación, podrán ser al menos parcialmente resueltos estos problemas.

CONCLUSIONES

En todo caso ya se entrevee y se puede valorar la oportunidad entusiasmante que los hypermedia puedan ser las herramientas diarias de los "information workers", permitiéndoles integrar un bien fundamentado acceso a bases de datos on-line con una red de trabajos personales interconectados.
La idea básica de los hypertextos, o sea que las ideas corresponden a objetos perceptibles, y que entonces se pueden manipular las ideas y sus relaciones así como se manejan directamente las ventanas y los íconos, no deja de ser interesante. Es un primer paso hacia aquel entonces en el cual la computadora será una ágil y brillante extensión de la mente humana, así como Bush imaginó en su *Memex* hace 40 años.

BIBLIOGRAFIA

1. T. H. Nelson, *"Literary Machines"*, Swarthmore, Penn., 1981,conseguible del autor.

2. J. Conklin, *"Hypertext: An Introduction and Survey"* Computer, Sept.1987, pp.17-41.

3. N. Yankelovich, N. Meyrovitz, A. van Dam, *"Reading and Writing the Electronic Book"* Computer, Oct.1985, pp.15-30.

4. D. Goodman, *"The two faces of HyperCard"*, Macworld, Oct.1987,pp.122-129.

5. B. Shneiderman, *"Designing the User Interface: Strategies for Effective Human Computer Interaction"* Addison-Wesley,Reading,Mass.,1987.

6. G. Marchionini, *"Information-Seeking Strategies of Novices Using a Full-Text Electronic Enciclopedia"* J.Am. Society for Information Science.

7. S. Weyer, A. Borning, *"A Prototipe Electronic Enciclopedia"*, ACM Trans. Office Information Systems, Vol.3, No.1, Jan. 1985, pp.63-88.

8. *Inside Macintosh*, Apple Computer, Inc., Cupertino, Calif., 1985.

9. S. Feiner, S. Nagy, A. van Dam, *"An Experimental System for Creating and Presenting Interactive Graphical Documents"* Trans. Graphics, Vol. 1, No.1, 1982, pp. 59-77.

10. A. Kay, A. Goldberg, *"Personal Dynamic Media"* Computer, Marzo 1977, pp.31-41.

11. T. W. Malone y otros, *"Intelligent Information-Sharing Systems"*, Communications of the ACM, Mayo 1987, pp. 390-402.

12. P. J. Brown, M. T. Russell, *"Converting Help Systems to Hypertext"* Short Communications, Software-Practice and Experience, Sept. 1987.

13. L. Koved, B. Shneidermann, *"Embedded menus: Selecting items in context"*, Communications of the ACM 29,4,(abril 1986), pp.312-318.

14. G. Marchionini, B. Shneidermann, *"Finding facts and browsing knowledge in hypertext systems"* IEEE Computer 21, (enero 1988), pp.70-80.

15. B. Shneidermann, *"User interface design for the Hyperties Electronic encyclopedia"*, Hypertext 1987 Workshop, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987, pp. 199-204.

16. R. Akscyn, D. MacCracken, E. Yoder, *"KMS: A Distributed System for Managing Knowledge in Organization'"* Hypertext 1987 Workshop, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

17. P. J. Brown, *"Turning ideas into products: The Guide System"* Hypertext 1987 Workshop, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

18. B. Campbell, J. M. Goodman, *"HAM: A General Purpose Hypertext Abstract Machine"* Hypertext 1987 Papers, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

19. J. Conklin, *"gIBIS: A Hypertxt Tool for Team Design"*
Hypertext 1987 Papers, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

20. G. H. Collier, *"Thoth-II: Hypertext with Explicit Semantics"*
Hypertext 1987 Papers, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

21 N. M. Delisle, M. D. Schwartz, *"Context-A Partioning Concept for Hypertext"*.
Hypertext 1987 Papers, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

22. D. Engelbart, *"Autorship Provision in Augment"*
Hypertext 1987 Papers, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

23. P. K. Garg, *"Abstraction Mechanisms in Hypertext"*
Hypertext 1987 Papers, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

24. C. Neuwirth, D. Kaufer, R. Chimera, T. Gillespie *"The Note Program: A Hypertext Application for Writing from Source Texts"*
Hypertext 1987 Papers, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

25. T. Oren, *"The Architecture of Static Hypertexts"*
Hypertext 1987 Papers, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

26. J. B. Smith, S.F. Weiss, G. J. Ferguson, *"A Hypertext Writing Environment and its Cognitive Basis"*
Hypertext 1987 Papers, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

27. J. Walker, *"Document Examiner: Delivery Interface for Hypertext Documents"*
Hypertext 1987 Papers, The University of North Carolina, Chapel Hill, NC, Nov. 13-15, 1987.

28. V.Bush, *"As We May Think"*
Atlantic Monthly, Julio 1945, pp.101-108. Reprinted in CD-ROM, The New Papyrus,S.Lambert y S.Ropiequet, ed. Microsoft Press, Redmond, WA, 1986.

29. D.Engelbart, *"A Conceptual Framework for Augmentation of Man's Intellect"*
en Vistas in Information Handling, Vol.I, Spartan Books, Washington, D.C., 1963, pp. 1-29.

30. D.Engelbart, W.K.English, *"A Research Center for Augmenting Human Intellect"*
AFIPS Conf. Proc.,Vol.33, Part 1, The Thompson Book Company, Washington, D.C., 1968.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

FERRAMENTAS PARA EDIÇÃO DE TEXTO EM UM AMBIENTE OPERACIONAL AMIGÁVEL

- **Ana Carla Bon**

COBRA - Computadores e Sistemas Brasileiros S/A.
Av. Comandante Guarannys, 447
Jacarepaguá
Cep: 22785 - Rio de Janeiro - RJ
Brasil

- **Luiz Carlos Montez Monte**

Universidade Federal do Rio de Janeiro
Cidade Universitária - Centro de Tecnologia
COPPE - Engenharia de Sistemas e Computadores
Bloco H - Sala 319
Ilha do Fundão
Cep: 21945 - Rio de Janeiro - RJ
Brasil
Telefone: 280-8832 - Ramal: 419

RESUMO

Esse trabalho desenvolveu o EdiTex, que é um conjunto de ferramentas de edição de texto para um ambiente operacional amigável, contruídos a partir da identificação do texto como objeto fundamental de interface com o usuário.

Essas ferramentas são um subconjunto do projeto FIU (14) - Ferramentas de Interface com o Usuário - desenvolvido na Cobra Computadores S.A., resultado da proposta de se suprir a carência de um ambiente amigável no SOX (15), que é um sistema operacional compatível com o Unix.

Uma das principais características do FIU é, através do EdiTex, fornecer edição de texto sofisticada embutida no sistema, disponível a qualquer aplicação. O Editex provê amplos recursos para que sejam atendidas de forma abrangente, as diversas necessidades de edição e exibição de texto na interface homem-máquina, como: a criação de diálogos textuais, a exibição de legendas e o desenvolvimento de aplicações voltadas exclusivamente ao processamento de texto.

Inédito no Brasil, o projeto FIU baseou-se em extensa pesquisa sobre o estado da arte, tanto na área de fatores humanos, quanto em Engenharia de Software, utilizando no seu desenvolvimento metodologias de projeto de comunicação com o usuário e definindo sua arquitetura orientada a objetos.

I. INTRODUÇÃO

Texto é, na maioria das vezes, indispensável na interação com o usuário. Algumas aplicações, tais como processadores de palavras, são especializadas em texto; outras, como planilhas ou aplicativos de exibição de relatórios, não são especialistas em texto, mas utilizam-no de forma intensa. Até mesmo aplicações gráficas utilizam texto, embora de forma esporádica. De forma geral, no entanto, as aplicações necessitam fornecer e obter informações textuais dos usuários.

Para o usuário é imprescindível que o texto seja editado da mesma maneira, não importando em que aplicação ou onde ele apareça. Por outro lado, para o programador de aplicações é útil que existam ferramentas dedicadas à exibição e edição de texto.

A implementação do conjunto de ferramentas de edição de texto, EdiTex, teve como objetivo prover recursos para atender as mais diversas necessidades de edição e exibição de texto dentro de um ambiente operacional amigável.

Pretende-se que toda aplicação, desenvolvida para o SOX, que utilize, de alguma forma, texto (seja na mais básica exibição de informação, seja criando softwares sofisticados de processamento de texto) obtenha os principais recursos necessários para sua implementação no que se refere a texto, dentro do conjunto de ferramentas especificadas.

O objetivo do EdiTex, portanto, não é implementar aplicações como editores de texto, mas, entre outros, permitir a redução do custo do desenvolvimento dessas aplicações e garantir uma interface com o usuário consistente com o ambiente no qual estão inseridas.

II. DESCRIÇÃO GERAL

II.1 - Metodologias Empregadas

Grandes esforços têm sido realizados, tanto na área de Fatores

Humanos quanto em Engenharia de Software, no sentido de oferecer critérios e métodos a serem seguidos na especificação de projetos de interfaces interativas.

Uma das propostas desse trabalho foi elaborar um desenvolvimento orientado à qualidade do produto, utilizando-se conhecimentos adquiridos dessas áreas.

O desenvolvimento do EdiTex foi realizado em três fases: a fase da Aquisição e Definição de Requisitos, a Fase do Projeto e a Implementação (fig II.1).

Segundo o grupo de desenvolvimento do Star (44), o primeiro passo a ser dado antes do projeto bem sucedido de qualquer interface com o usuário deve ser a "Definição de Requisitos", também denominada de "Análise das Tarefas". Esse passo envolve a análise da área da aplicação e de seus usuários, estabelecendo "quem são os seus usuários, quais os seus objetivos em realizar determinadas tarefas, quais informações eles geram e que métodos empregam".

A fase de Definição dos Requisitos é, em resumo, um levantamento de dados, que é concluído com a elaboração de um documento denominado Especificação de Requisitos de Software (ERS), utilizado como base para as decisões nas fases seguintes de projeto e implementação do software.

Após a fase de Definição de Requisitos, iniciou-se a fase de Projeto que foi dividida em três subfases: o Projeto Externo, no qual foram definidas as Interfaces do EdiTex, isto é, interface com o usuário-operador e com o usuário-programador; o Projeto Arquitetural onde foi realizado a decomposição do EdiTex em módulos; e o Projeto Detalhado, no qual foram elaborados os algoritmos e definidas as estruturas de cada módulo.

Finalmente, a última fase do desenvolvimento do projeto foi a implementação na linguagem de programação "C" do que foi estabelecido nas fases anteriores. A figura II.1 mostra um diagrama do ciclo de desenvolvimento do projeto, empregado conforme des-

crito acima.

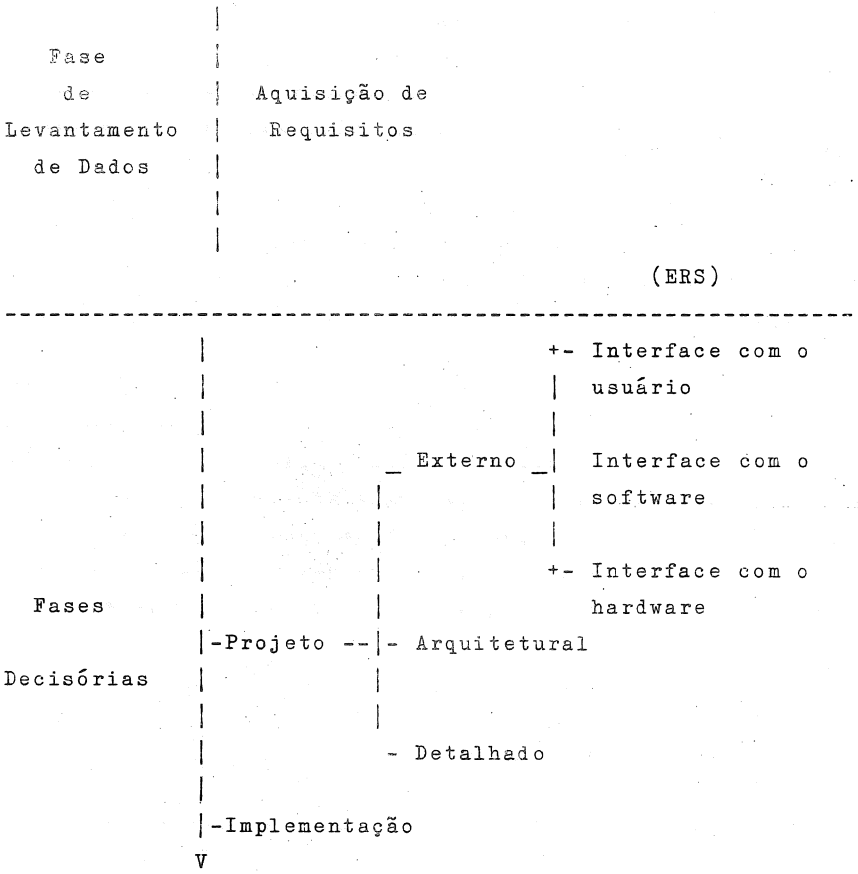


fig II.1 : Fases do Projeto EdiTex

Duas metodologias foram empregadas na realização dessas fase: a Metodologia de Projeto Orientada a Objetos e a Metodologia de Especificação de Interfaces Interativas.

A Metodologia Orientada a Objetos permite que, num dado nível da solução do projeto, seja identificado os objetos e definidas as operações apropriadas a esses objetos. Embora mais dirigida à fase de Projeto Arquitetural e Detalhado, pois como definiu BOOCH (10), ela é uma metodologia que "fornece critérios para a decomposição de um sistema em módulos", ela foi aplicada desde as primeiras fases do desenvolvimento do EdiTex: as entidades

do mundo real e os comportamentos dos usuários sobre essas entidades foram mapeadas em objetos e operações sobre esses objetos.

A abordagem orientada a objetos atrasa, ao máximo possível, o surgimento de processos e dados (só presente na fase de implementação), fazendo com que o mapeamento do mundo real para os objetos existentes nas especificações e nos projetos seja imediato. A aplicação do uso dessa metodologia torna mais fácil a manutenção e alteração do sistema.

Na elaboração do Projeto Externo foi utilizada a Metodologia de Especificação para Interfaces Interativas como descrita por FOLEY et alii (20), que propõem a descrição da linguagem de interação com o usuário-operador em quatro passos: conceitual, semântico detalhado, sintático e léxico.

A vantagem dessa abordagem "top-down", é que durante a especificação da linguagem de interação, é possível abstrair-se, em um dado passo, das decisões que só precisam ser tomadas nos passos seguintes. Por exemplo, durante a definição da semântica da linguagem de interface com o usuário, não é necessário considerar características relacionadas a dispositivos de hardware, que só afetam o projeto léxico.

II.2 - Perspectiva do Sistema

II.2.1 - Ambiente do Editex

O EdiTex é um subconjunto do FIU - Ferramentas de Interface com o Usuário. O FIU é "a parte do SOX responsável pela comunicação de dados entre os usuários e os processos (aplicações)"(COBRA(14)).

A interface FIU-aplicação é realizada por meio de chamadas e retornos de funções do FIU e somente utilizando essas funções é que a aplicação se comunica com o usuário.

Cada usuário envia dados e controles para o FIU pelo teclado ou

pelo mouse do seu terminal e recebe informações pela tela desse terminal .

Além de fornecer um conjunto de ferramentas para a programação da interface com o usuário dos aplicativos, o FIU "induz a uma padronização dessas interfaces" propiciando a construção de um ambiente amigável no SOX.

Entende-se por amigável, um software de fácil uso, cuja principal característica é comunicar-se bem (HECKEL (24)). Um conjunto de aplicações amigáveis, integradas entre si a partir de uma consistência nas suas interfaces com o usuário, formam um ambiente amigável.

Para que um ambiente seja amigável, é indispensável que exista um aplicativo responsável pela administração do sistema, cuja função seja permitir ao usuário a execução fácil de suas tarefas e a utilização de todos os recursos da máquina. Não é possível pensar em ambiente amigável nos sistemas compatíveis com o Unix, enquanto a comunicação usuário-sistema for realizada por meio da Shell.

Além disso, devem ser oferecidas ferramentas de propósito geral que visem atender às necessidades de usuários distintos. Um conjunto mínimo dessas ferramentas deve conter um processador de texto, um processador gráfico e alguns utilitários acessórios a outras tarefas como uma máquina de calcular e um bloco de rascunho.

II.2.2 - FIU

O FIU tem como principal característica "promover a utilização de técnicas avançadas de comunicação com o usuário, independentemente da existência de facilidades gráficas" (Monte(36)). Embora atualmente o FIU esteja sendo utilizado em um ambiente semi-gráfico, poucas alterações serão necessárias para evolução a um ambiente gráfico.

A utilização do mouse é opcional, podendo o operador utilizar

somente o teclado. As funções do FIU manipulam principalmente, texto, janelas, diálogos, cardápios e predefinidos. A seguir estarão descritos os significados destes componentes do FIU (COBRA (14)).

- Janelas

Visualmente uma janela é uma área retangular da tela na qual são exibidas informações e que pode sobrepor total ou parcialmente, ou ser sobreposta por outras janelas. Funcionalmente, uma janela é o meio pelo qual uma aplicação transmite e recebe dados do usuário.

A utilização de janelas sobrepostas no FIU baseou-se no conceito criado pelo Smalltalk-80 (22) e utilizados em diversos sistemas de interface surgidos posteriormente como o Star (44), Lisa (48), Macintosh (2) e o Amiga (46). Uma única aplicação, ou aplicações distintas, podem possuir simultaneamente um número qualquer de janelas na tela. Porém, em um dado instante, apenas uma janela pode estar selecionada, significando que somente essa janela está apta para receber dados do usuário. A facilidade de aplicações diferentes poderem possuir janelas na tela simultaneamente, está presente em vários sistemas de interface como o MS-Windows (34), X-Windows (35), Unix PC (4) e o Amiga. O Macintosh, não permite essa facilidade.

Existem vários tipos padronizados de janelas, cada um sendo adequado ao tipo de informação tratada, existindo desde janelas para exibição de mensagens de erro até janelas sofisticadas para edição de texto.

Cada janela tem, associada a ela, um ambiente de desenho, que é o conjunto completo das informações necessárias para as operações de desenho da interface com o usuário. Um ambiente de desenho contém, entre outras coisas: as dimensões de uma janela, sua posição na tela, o padrão de preenchimento da tela, os tipos e estilos de caracteres no texto. Utilizando conceitos gráficos como sistemas de coordenadas, um ambiente de desenho permite que todos os desenhos e cálculos de uma janela sejam fei-

tos baseados no próprio sistema de coordenadas locais dessa janela. Também são utilizadas técnicas gráficas de "clipping" (recorte) para determinar as regiões de uma janela que precisam ser redesenhadas num dado momento.

- Caixas de Diálogo

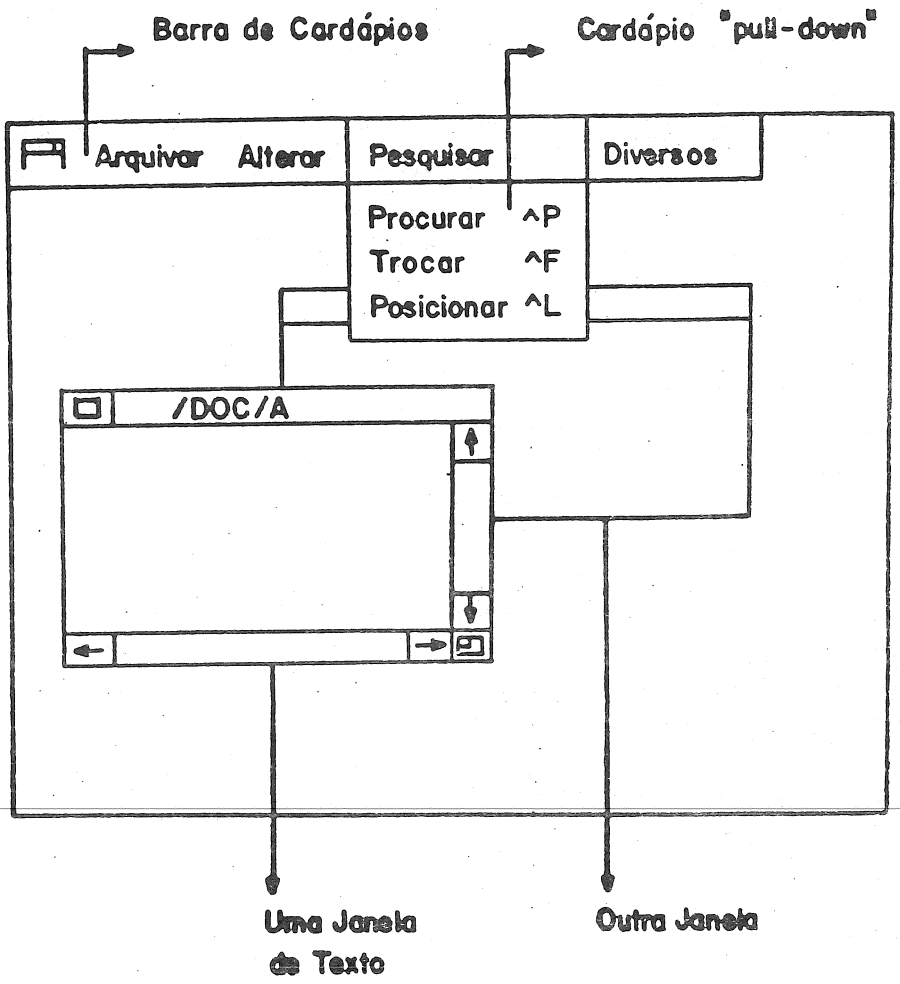
Uma Caixa de Diálogo é um tipo de janela utilizado para que uma aplicação obtenha informações adicionais do usuário a fim de poder executar um determinado comando. Caixas de Diálogo também são utilizadas para fornecer informações importantes, tais como mensagens de erro ou o acompanhamento de uma tarefa.

A interação com o usuário realizada por meio de uma Caixa de Diálogo denomina-se diálogo. Existem três tipos de diálogo, que diferem quanto ao nível de restrição imposta às ações do usuário: Diálogos Imodais - não restringem qualquer ação do usuário, permitindo-o interagir com outras janelas enquanto a Caixa de Diálogo permanece na tela; Diálogos Modais - limitam a ação do usuário a responder o diálogo ou descartá-lo, não permitindo qualquer interação com outras janelas; e Alerta - que só permite ser descartado, uma vez que não é pedida nenhuma informação ao usuário. Diálogos modais e imodais são utilizados para coletar informações do usuário, enquanto Alertas são utilizados para informar algo ao usuário como, por exemplo, mensagens de erro.

Dentro de uma Caixa de Diálogo, existem dispositivos padronizados por meio dos quais o usuário fornece dados à aplicação. São eles: Caixas de Texto, para o usuário preencher dados textuais; Caixas de Marcas para o usuário determinar opções não exclusivas; Botões de Rádio, para o usuário escolher uma opção dentre um conjunto de opções mutuamente exclusivas; e Teclas de Luz, para comandar ações.

- Cardápios

Um cardápio é uma lista de opções da qual o usuário pode reali-



FIU: A Tela de um Aplicativo

zar uma escolha. O FIU determina a existência de uma barra de cardápios sempre visível, posicionada na primeira linha da tela. Essa barra de cardápios, como no Macintosh, contém títulos de cardápios, que ao serem selecionados, desenrolam a lista de opções daquele cardápio ("pull-down menu"). A barra de cardápios possibilita ao usuário listar, a qualquer momento, todos os comandos de uma determinada aplicação e verificar quais estão disponíveis para serem executados.

- Predefinidos

Predefinido "é uma descrição de entidade previamente gerada e mantida em um arquivo sob controle do FIU" (COBRA(14)). O FIU utiliza predefinidos para armazenar formas de janelas, cardápios, caixas de diálogo, entre outras.

Predefinidos propiciam a separação entre o código da aplicação e o conjunto de dados que constituem a interface com o usuário. Isso permite que a aplicação experimente diferentes formatos de informação sem alterar o código do seu programa.

Arquivos de dados também podem estar associados a predefinidos e isso tem implicação importante no desenvolvimento de processadores de texto porque permite que o texto seja gravado, associado aos seus dados de formatação. O FIU fornece um compilador de predefinidos.

II.3 - **Usuários do Editex**

A palavra "usuário" tem sido empregada na redação desse trabalho com o único sentido de caracterizar as pessoas que operam o computador. No entanto, existe uma outra classe distinta de usuários do EdiTex.

Esta classe é formada pelos projetistas e programadores de interfaces com o usuário. De uma forma geral, o conjunto das ferramentas de edição deve ser apropriado ao desenvolvimento de qualquer aplicação que utilize texto.

A segunda classe é constituída pelos usuários operadores da máquina. Representantes dessa classe tanto podem ser profissionais leigos em computação, que utilizam um microcomputador pessoal como apoio às suas atividades, quanto profissionais da área de informática. Em ambos os casos, esses profissionais necessitam de formas eficientes de comunicação com o sistema, o que inclui ferramentas adequadas para a edição de texto.

II.4 - Visão Global

Essa seção fornece uma visão global da funcionalidade desejada para o EdiTex. O EdiTex - conjunto de ferramentas para edição e exibição de texto - deve fornecer ferramentas para a programação das partes textuais das interfaces com o usuário das aplicações e gerenciar as ações dos usuários sobre essas partes, durante a execução das aplicações. Estas ferramentas devem servir para a manipulação (tanto do programador quanto do operador) de três classes de objetos textuais: linhas de texto não editáveis, campos de texto e janelas especializadas em texto.

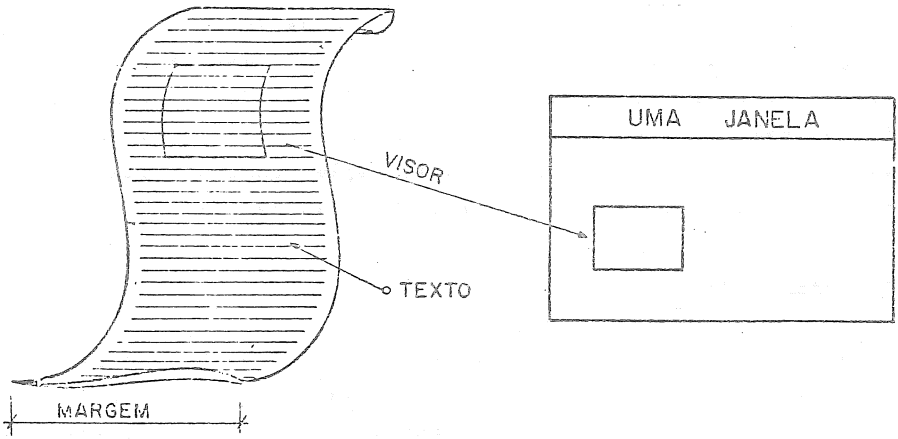
1 - Linha de texto não editável

Uma linha de texto não editável é constituída por uma seqüência de um ou mais caracteres. Linhas de texto podem ser exibidas na tela, ou seja, os caracteres que compõem a linha podem ser desenhados na tela de acordo com determinadas características de desenho.

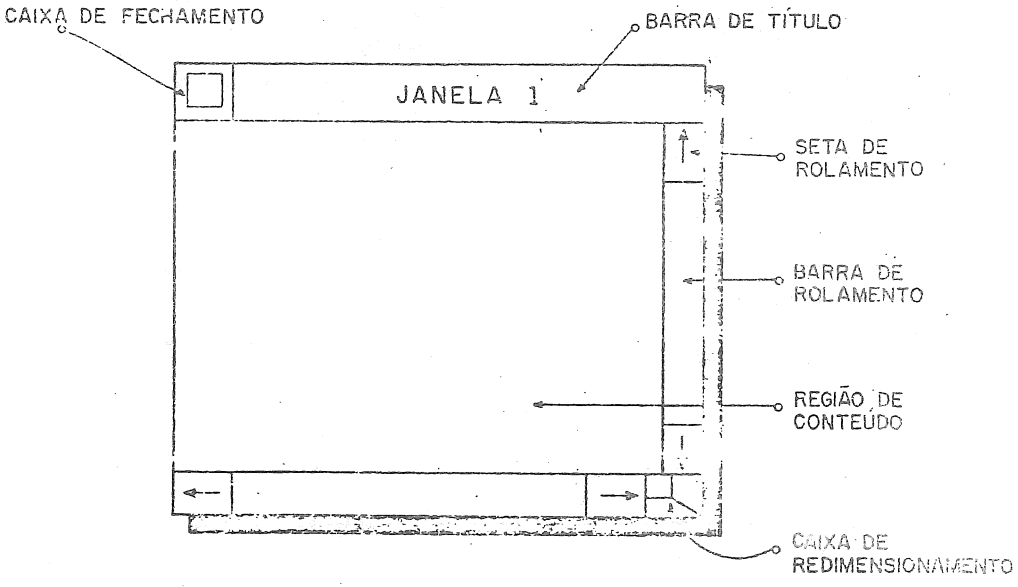
2 - Campos de Texto

Um Campo de Texto é um texto exibido formatado e que pode ser visualizado totalmente ou parcialmente em uma janela na tela através de um campo retangular denominado visor. Operações sobre o texto do Campos de Texto podem ser divididas em dois grandes grupos:

- operações de alteração de conteúdo do texto, tais como inserir novos caracteres ou copiar um trecho do texto para outra parte do mesmo texto.
- operações de definição da formatação do texto, tais como:



Objeto campos de texto



Objeto janela de texto

margens, alinhamento às margens, tabulação, estilo e tamanho dos caracteres.

3 - Janelas especializadas em texto

Uma Janela especializada em texto é um tipo de janela cuja região de conteúdo exibe um determinado texto. Operações possíveis de serem realizadas em uma janela especializada de texto incluem:
- todas as operações inerentes a janelas no FIU, tais como movimentação e redimensionamento da janela na tela;
- todas as operações de alteração e formatação do texto;
- operações específicas de janela de texto tais como: controle da visualização do texto dentro da janela e subdivisão da janela.

II.5. - Atributos Desejáveis

Com o objetivo de se obter uma boa interface com o usuário, foram definidos previamente os atributos que o sistema deveria atender. Os atributos selecionados são os propostos pela literatura pesquisada em Interfaces com o Usuário, Fatores Humanos e Processamento de Texto.

II.5.1 - Consistência:

Esse atributo determina, de uma forma geral, que qualquer software escrito para a mesma máquina deve possuir a mesma aparência, isto é, que o usuário não precise, para cada nova aplicação utilizada, reaprender uma nova interface (APPLE (?)). Além disso, o conhecimento que o operador adquire de uma parte do sistema o ajuda em outras partes (MARCUS (31), SIMPSON (42)).

Em particular, isso significa que o usuário deve ter a mesma maneira de editar um texto, seja qual for a aplicação que ele esteja utilizando. O EdiTex oferece facilidades das mais simples às mais sofisticadas para exibir e editar texto. Uma aplicação que não seja voltada para texto e que não necessariamente utilize todas essas facilidades, deve garantir que o conjunto

mínimo oferecido seja compatível com as do conjunto completo. Por exemplo, o usuário deve ter a mesma maneira de retirar do texto o último caractere digitado; porém, não necessariamente poderá em todas as aplicações selecionar palavras.

II.5.2 - Capacidade de fornecer respostas rápidas

A capacidade de fornecer respostas rápidas significa que resultados imediatos devem estar associados às ações do usuário para que ele possa avaliar a ação realizada e seguir em frente (APPLE (2)).

BERTINO (9) diz que é de "vital importância que uma interface seja projetada de forma que o usuário nunca fique pensando se o seu comando (ou dado) enviado, foi recebido". Ele classifica dois tipos de realimentação necessárias em um sistema de interface. O primeiro tipo fornece a realimentação das ações do usuário e incluem eco, destaque na tela de objetos e cardápios selecionados, movimentos do cursor e mensagens de erro. O segundo tipo fornece realimentação do progresso das ações do sistema, especialmente operações que consumam muito tempo, exibindo alguma indicação de que o que foi pedido está sendo realizado.

EMBLEY(16) et alii citam tempos de respostas aceitáveis para determinadas operações de edição: características de eco - menor do que 0.1 segundos ; pedidos conversacionais - menor do que 2 segundos ; procura página por página - menor do que 0.5 segundos para que apareçam as primeiras linhas da próxima página ; iniciar ou sair de edição - menor do que 15 segundos

II.5.3 - Permissividade

Significa que o sistema deve permitir ao usuário fazer qualquer coisa que seja razoável, isto é, que seja o usuário, e não o programa, quem determina o que deve ser feito. O sistema deve produzir ao usuário a sensação do domínio da situação e conseqüentemente de segurança em relação ao computador (APPLE (2)). Para HECKEL (24), uma interface deve ser permissiva; ele suge-

re: "dê o controle ao usuário".

Na edição de texto, o usuário deve ter flexibilidade para percorrer seu texto e selecionar parte dele sem se restringir a uma função particular ou às suas opções.

A melhor maneira de um sistema se tornar permissivo é evitando modos (TESLER(43), APPLE(2)). Um modo, ou um estado, pode ser definido como uma parte de uma aplicação onde o usuário deve, formalmente, entrar e sair e que, enquanto permanece ativo, restringe a realização de operações.

II.5.4 - Multimodalidade

A comunicação do sistema de interface com o usuário deve permitir interação em vários níveis de experiência do usuário de forma simultânea, aceitando comandos equivalentes com sintaxes de pouca complexidade e sintaxes mais complexas (APPLE(2), BERTINO(9), HECKEL(24)).

Isso permite que usuários pouco experientes consigam realizar as mesmas tarefas que um usuário experiente, tendo apenas que executar mais ações para obter o resultado esperado. Por outro lado, os usuários já familiarizados com o sistema podem executar suas ações mais rapidamente.

II.5.5 - Indulgência

Esse atributo implica em que um sistema deve, além de minimizar as chances do usuário cometer erros, limitar as conseqüências dos erros e também fornecer maneiras rápidas e simples do usuário se recuperar de situações causadas por erros.

Diagnósticos de erros devem ser claros para evitar que o usuário fique fazendo tentativas para corrigir os erros. (BERTINO(9)). Na edição de texto, conforme citado por EMBLEY et alii(18), "mesmo para usuários especialistas, de 5 a 30 por cento do tempo real de edição podem ser geralmente atribuidos à erros e correção de erros".

II.5.6 - Fácil de Aprender e Usar

"Um sistema que não tenta aliviar o medo, apreensão e incerteza que os usuários sentem quando estão usando o sistema, merecerá muito pouco interesse" (BERTINO(9)).

Alguns princípios a serem seguidos para se alcançar um sistema que tenha uma interface fácil de aprender e usar são:

- Mantenha o sistema simples ao invés de complexo (SIMPSON(42), HECKEL(24)).
- Comandos para o sistema devem ser intuitivos, isto é, eles devem descrever de forma sucinta e clara a ação que eles executam. HECKEL(24) sugere também a comunicação por metáforas.
- Deve ser minimizada a demanda da memória humana (SIMPSON (40)). Se para executar uma função, forem necessários diversas informações o sistema deve se encarregar de requisitá-las ao usuário, não obrigando-o a memorizá-las.
- Devem ser mantidos os comandos mais freqüentes na forma mais simples; comandos não freqüentes podem ser complexos (EMBLEY et alii(18)).
- O usuário deve, a qualquer momento, poder visualizar todo o conjunto de funções que o sistema oferece.
- O sistema deve fornecer um mecanismo eficiente de ajuda que auxilie, de forma clara e rápida, o usuário a entender algum aspecto do sistema (HECKEL (24), BERTINO (9), MEYROWITZ et alii (32)).
- O usuário deve ter formas de desfazer ações, seja para experimentar outras soluções, seja para se recuperar de erros (BERTINO(9), MEYROWITZ et alii (32)).

III. CONCLUSÃO

A evolução da Informática que levou o homem a conviver intensamente no seu cotidiano com o computador, vem direcionando o desenvolvimento de software no sentido da definição de interfaces mais amigáveis. Passou-se a considerar conceitos sobre o comportamento humano e aplicá-los nos sistemas de interfaces de

forma a produzir no homem a sensação de segurança e domínio em relação ao computador.

Reconhecendo a comunicação textual como um dos elementos básicos de um sistema de interface com o usuário, desenvolveu-se um conjunto de ferramentas para edição e exibição de texto - o EdiTex.

Atualmente o EdiTex é amplamente utilizado pelos demais módulos do FIU, além de já estar inserido em um sistema produtivo, tendo sido utilizado no projeto de um editor de texto experimental e no Projeto SIR - Sistema Integrado Relacional (COBRA (15)). Ambos os projetos utilizam intensamente Janelas de Texto editáveis ou não: o editor permite que sejam criadas até oito janelas de texto editáveis na tela, cada uma contendo um texto com formatos totalmente diversos; o SIR utiliza Janelas de Texto para que o usuário edite consultas a bancos de dados e para exibir relatórios. A maioria das Caixas de Diálogos criadas por esses projetos, contém diversos dispositivos de Caixas de Texto.

A aplicação da metodologia orientada a objetos, em particular na fase de Projeto Interno, tem proporcionado diversos benefícios pois permite que melhorias e alterações realizadas ao longo do desenvolvimento das ferramentas fiquem sempre restritas ao módulo afetado, sem impactar os demais módulos que o utilizam.

O objetivo do EdiTex, de atender as mais diversas necessidades de edição de texto em um ambiente operacional amigável, vem sendo alcançado. Em especial, a validade da proposta de se oferecer uma automatização da edição através de uma janela especializada, já foi comprovada.

Comparando-se as facilidades de edição de texto oferecidas pelos sistemas de interfaces pesquisados e o EdiTex, verifica-se que esse é um dos que mais recursos oferece para o desenvolvimento de programas aplicativos que utilizem de alguma forma texto. O sistema de interface, até então mais completo em rela-

ção a texto era o Macintosh, que embora ofereça através do seu "Toolbox" um imenso recurso gráfico na edição de texto, não oferece facilidades tais como tabulação, alinhamento às duas margens, janelas automáticas, e só permite um único estilo e conjunto de caracteres no texto. A aplicação que desejar esse recursos tem que elas mesmas programarem, sen utilizar o "Toolbox". Outro sistema que oferece grandes facilidades de edição é o X-Windows, que pelas propostas apresentadas no X-Toolkit, contém também janelas especializadas em texto, mas não oferece muitas facilidades de formatação de texto.

Seguindo o modelo de edição do Smalltalk-80, baseado na imodalidade, o EdiTex consegue oferecer ao usuário-operador uma grande facilidade nas suas tarefas de edição, ao mesmo tempo que integrado ao restante do FIU, pode proporcionar a construção de um ambiente amigável para o SOX.

REFERÊNCIAS BIBLIOGRÁFICAS

(1) APPLE COMPUTER INC., Mac Write, 1984.

(2) APPLE COMPUTER INC., Inside Macintosh, vol I,II e III, Addison-Wesley Publishing Company, Inc., 1985.

(3) ARCA, J.A, Pratical WordStar Uses, Sybex Inc, 1983.

(4) AT&T, AT&T UNIX PC Interface Specification, 1985.

(5) AT&T, The VI User's Handbook, 1984.

(6) AT&T, System V Interface Definitions, Issue 2, Vol. II, 1986.

(7) AT&T, UNIX User Reference Manual - System V - release 2.0, 1984.

(8) AVANT, H., "The AT&T UNIX PC: A UNIX Sistem for the Masses?", UNIX/WORLD, pp. 56-66, Setembro 1985.

(9) BERTINO, E., "Design Issues in Interactive User Interfaces", Interfaces in Computing, pp 37-53, Março 1985.

(10) BOOCH, G., Software Engineering with ADA, The Benjamin/Cummings Publishing Company, Inc, 1983.

(11) BYLES T., "One Approach to the human - computer interface", IEEE CG&A, pp. 245-251, Dezembro 1984.

(12) BYERS T.J., "TopView - IBMs long awaited multitasking

program makes its debut", BYTE, pp. 245-251, Outubro 1985.
(13) COBRA COMPUTADORES E SISTEMAS BRASILEIROS, "SIR/AC", Especificação Planejada de Software número 149, Outubro/1987.
(14) COBRA COMPUTADORES E SISTEMAS BRASILEIROS, "FIU - Ferramentas de Interface com o Usuário", Especificação Planejada de Software número 15, Setembro/1986.
(15) COBRA COMPUTADORES E SISTEMAS BRASILEIROS, "SOF", Especificaçaõ de Requisitos de Software número 5, junho/1987.
(16) DATA GENERAL CORPORATION, Using CEO Word Processing, Rev. 00, Maio 1982.
(17) EDWARDS J.R. et alii, "The Atari 520ST", BYTE, pp. 89-100, Janeiro 1986.
(18) EMBLEY, D.W. e NAGY, G., "Behavioral aspects of Text Editors", Computing Surveys Vol. 13, no. 1, pp. 33-70, Março 1981.
(19) FINSETH, C.A., "Managing Words - What Capabilities Should You Have With a Text Editor?", BYTE, pp. 302-310, Abril 1982.
(20) FOLEY J.D. et alii, "The Human Factors of Computer Graphics Interaction Techniques", IEEE CG&A, pp. 13-48, Novembro 1984.
(21) FOLEY J.D et alii, "The human Factors of Graphic Interaction Tasks and Techniques", School of Engineering and Applied Science, Janeiro 1981.
(22) GOLDBERG A. et alii, Smalltalk-80, The language and its implementation, Addison Wesley.
(23) GOOD M., "Etude and the Folklore of User Interface Design", ACM, pp.34-43, 1981.
(24) HECKEL P., The elements of friendly software design, Warner Books Inc, Maio 1984.
(25) HOFFMAN P., Microsoft Word, Made Easy, Osborne McGraw-Hill 1985.
(26) IBM, IBM Personal Computer Profissional Editor, Dezembro, 1972.
(27) IEEE, Guide To Software Requirements Specifications, The Institute of Eletrical and Eletronics Engineers Inc., 1984.
(28) JONG, S., "Designing a Text Editor? The User Comes First", BYTE, pp. 284-300, Abril 1982.
(29) LEMMONS, P., "Beyond the Word Processor", BYTE, pp. 53-56, Janeiro 1984.

(30) LEWIS, T.G., "Apple Macintosh Software", IEEE Software, pp.89-91, Março 1985.
(31) MARCUS A., "Corporate Identity for Iconic Interface Design: The Grafic Design Perspective", IEEE CG&A, pp. 24-32, Dezembro 1984.
(32) MEYROWITZ N. e DAM V.A., "Interactive Editing Systems", Computing Surveys, vol. 3, pp. 321-415.
(33) MICAL J.R., "Introduction to the Amiga ROM Kernel", BYTE, pp. 116-128, Fevereiro 1986.
(34) MICROSOFT CORPORATION, Microsoft Windows Users Guide, versão 1.0, 1986.
(35) MIT PROJECT ATHENA et alii, X Toolkit - A Proposed Architecture Preliminary Draft, Dezembro 1986.
(36) MONTE, L.C., "Sistemas de Interface com o Usuário", Monografia do Curso Tópicos Especiais em Banco de Dados não Convencionais, COPPE - Engenharia de Sistemas e Computação, Dezembro/1986
(37) NORMAN, D. A., "Design Rules Based on Analyses of Human Errors", Comunications of the ACM, vol. 26, n. 4, pp. 254-258, Abril 1983.
(38) ROBERT, T.L, "The Evaluation of Text Editors: Metodology and Empirical Results", Communications of the ACM, vol. 26, no.4, pp. 265-283, Abril 1983.
(39) SCHEIFLER, R. W. e GETTYS, J., "The X-Windows System", ACM Transaction on Graphics, vol.5, n.2, pp 79-109, Abril 1986.
(40) SIMPSON, H., Design of User Friendly Programs for Small Computers, McGraw Hill Book Company, 1984.
(41) SMITH D.C., "Macintosh System Architecture", BYTE, Fevereiro 1984.
(42) SMITH D.C. et alii, "Designing the Star User Interface", BYTE, pp. 242-282, Abril 1982.
(43) TESLER L., "The Smalltalk Environment", BYTE, pp. 90-147, Agosto 1981.
(44) WILLIAMS G et alii, "The AMIGA Personal Computer", BYTE, pp 83-100, Agosto 1985.
(47) WILLIAMS G., "The AT&T UNIX PC", BYTE, pp 98-105, Maio 1985.
(48) WILLIAMS G., "The Lisa Computer System", BYTE, pp. 33-53, Fevereiro 1983.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

ORGANIZAÇÃO DO SISTEMA OPERACIONAL SOMBRA

R. S. DE OLIVEIRA *, C. M. DA COSTA **

SUMARIO

Este trabalho apresenta o sistema operacional multiusuário SOMBRA. Este sistema incorpora uma série de características presentes em outros sistemas operacionais modernos, tais como: linguagem de comando com construções de alto nivel, sistema de arquivos hierárquico organizado em forma de árvore, possibilidade da execução de programas em background e spooling de saída para impressoras.

ABSTRACT

This paper describes a multi-user operating system called SOMBRA. This system exhibits many features which are found in other modern operating systems, such as: a command language with high level constructs, a hierarchical file system organized as a tree, background execution of programs and output spooling to the printers.

* Engenheiro Eletricista (PUCRS, 1983), Mestre em Ciência da Computação (UFRGS, 1987); áreas de interesse: programação concorrente e sistemas operacionais; Curso de Pós-Graduação em Ciência da Computação da UFRGS; Av. Osvaldo Aranha, 99 - Porto Alegre - RS - CEP 90000.

** Analista de Sistemas (PUCRS, 1978), Mestre em Ciência da Computação (UFRGS, 1985); áreas de interesse: programação concorrente e sistemas operacionais; Professor Adjunto do Departamento de Informática/Curso de Pós-Graduação em Ciência da Computação da UFRGS; Av. Osvaldo Aranha, 99 - Porto Alegre - RS - CEP 90000.

INTRODUÇÃO

O Sistema Pascal Concorrente /HAN 77/ cria um ambiente de programação concorrente de alto nivel, onde é possivel implementar sistemas operacionais que podem ser transportados para outras máquinas com relativa facilidade. O Curso de Pós-Graduação em Ciência da Computação (CPGCC) da UFRGS dispõe do Sistema Pascal Concorrente implementado em um minicomputador LABO 8034 /MED 81/.

O sistema operacional SOM /BRA 84/ foi especificado para criar um ambiente interativo, multiusuário, tendo como base o Sistema Pascal Concorrente. Entretanto, não foi implementado.

Entre os projetos em andamento no CPGCC está a construção de uma máquina multiprocessadora, baseada em dois microprocessadores MC68000 da Motorola. Atualmente, um protótipo desta máquina, conhecida como máquina SUMUS, está em fase de depuração.

Como parte deste mesmo projeto, foi prevista a implementação do Sistema Pascal Concorrente na máquina SUMUS e a implementação de um sistema operacional multiusuário para este ambiente, tendo como ponto de partida a especificação do sistema SOM. O projeto ainda prevê que este sistema operacional deve ser implementado no LABO 8034 e posteriormente transportado para a máquina SUMUS.

O presente trabalho descreve a especificação e implementação do SOMBRA, um sistema operacional multiusuário, escrito em Pascal Concorrente no LABO 8034. Este sistema apresenta diversas alterações em relação a especificação do SOM.

O SOMBRA foi criado para satisfazer um dos objetivos do projeto descrito. Para tanto, ele deverá ser transportado para a máquina SUMUS, tão logo o Sistema Pascal Concorrente esteja lá disponível.

Devido às restrições impostas pela pequena memória principal do LABO 8034 (se comparada com a da máquina multiprocessadora), não é possivel executar o SOMBRA completo e ainda dispor de um espaço razoável para a execução dos programas de usuário. Isto nos leva à execuções parciais do SOMBRA no LABO 8034, onde componentes específicos são testados separadamente. Testes de conjunto são efetuados na medida em que o tamanho da memória permite.

Abaixo estão enumeradas algumas características do sistema operacional:

- Multiusuário;
- Linguagem de Comando com construções de alto nível;
- Sistema de Arquivos organizado em árvore;
- Execução de programas em background;
- Spooling de saída para impressoras;

ORGANIZAÇÃO DO SOMBRA

Esta seção mostra uma visão geral do sistema, através da apresentação de suas características gerais, e de sua organização. Os diversos componentes do SOMBRA são descritos funcionalmente.

De uma forma geral, o SOMBRA pode ser dividido em três grandes partes:

- Os processos envelope;
- Os programas seqüênciais;
- Os subsistemas do programa concorrente.

PROCESSO ENVELOPE

Processo envelope é o elemento ativo do sistema que executa as tarefas requisitadas pelo usuário. Cada terminal é atendido por um processo envelope. O que este processo faz essencialmente é carregar e executar programas seqüenciais, e atender as chamadas de sistema destes programas.

A primeira tarefa do processo envelope é efetuar a conexão de um usuário. Isto é feito através da carga e execução de um programa seqüencial adequado. Uma vez tendo sido reconhecido pelo sistema, o usuário poderá dar início a sua sessão de trabalho. O processo envelope então carrega e executa o interpretador da Linguagem de Comando.

A Linguagem de Comando é a interface entre o usuário e o sistema. É através dela que o usuário determina as tarefas a serem realizadas pelo sistema. A linguagem pode ser usada, por exemplo, para listar o conteúdo de um diretório, para listar os usuários correntemente conectados ao sistema, para executar um programa seqüencial especifico. O interpretador da linguagem, ele próprio um programa seqüêncial, executa as tarefas determinadas pelo usuário através de chamadas de sistema ou através da execução de um outro programa seqüencial em overlay.

As chamadas de sistema oferecidas pelo SOMBRA estão listadas abaixo, juntamente com uma descrição sumária de cada uma delas.

OPEN - Abre arquivo para acesso, inclusive arquivos de impressão;

CLOSE - Fecha arquivo previamente aberto, inclusive arquivo de impressão;

WRITE - Escreve em um arquivo previamente aberto, inclusive arquivo de impressão. O acesso é feito a registro, cujo tamanho é definido na abertura e pode variar entre 2 e 512 bytes;

READ - Lê de arquivo previamente aberto. O acesso é feito a registro, cujo tamanho é definido na abertura e pode variar entre 2 e 512 bytes;

GETCHAR - Lê de arquivo previamente aberto. O acesso é feito a caractere;

PUTCHAR - Escreve em arquivo previamente aberto, inclusive arquivo de impressão. O acesso é feito a caractere;

RUN - Executa programa seqüencial através de overlay;

MARK - Determina a posição atual da área de heap do programa;

RELEASE - Libera área de heap do programa;

GETTIME - Obtém o dia e hora atuais;

ACCEPT - Lê caractere do teclado;

DISPLAY - Coloca caractere na tela;

IOTRANSFER - Reservada para uso futuro;

RUNJOB - Inicia execução em background de um programa seqüencial;

SENDMSG - Envia mensagem para outro terminal;

GETTERM - Obtém o estado de todos os terminais do sistema e dos processos envelope destinados à execução em background;

GETDISK - Obtém o estado de todas as unidades de disco do sistema;

GETUSER - Obtém o número do terminal e a identificação do usuário conectado;

MAKEDIR - Cria diretório;

DELETE - Remove arquivo, diretório ou ligação;

RENAME - Troca o nome de arquivo, diretório ou ligação;

LISDIR - Obtém o conteúdo de um diretório;

FILESTATUS - Obtém ou altera características de um arquivo;

CDIR - Posiciona o diretório corrente;

READCD - Obtém a identificação do diretório corrente;

LINK - Estabelece ligação para arquivo ou diretório;
MOUNT - Monta volume em unidade de disco;
UNMOUNT - Desmonta volume;
BLOCK - Bloqueia um outro terminal;
UNBLOCK - Libera terminal para uso;
WILLBLOCK - Determina se o terminal será ou não bloqueado;
BLOCKMYSELF - Bloqueia o próprio terminal;
SETREADY - Comunica conexão de usuário;
STOPPGM - Aborta execução de programa seqüencial em outro terminal;
SETDATE - Acerta a data no relógio do sistema;
SETCLOCK - Acerta a hora no relógio do sistema.

Uma vez terminada a conexão, através de comando adequado, dado pelo usuário ao interpretador, o processo envelope executa uma limpeza em suas estruturas de dados e passa a esperar uma nova conexão, sendo este o seu ciclo de trabalho ao longo da existência do sistema.

PROGRAMAS SEQÜENCIAIS

Um programa seqüencial é carregado e executado pelo processo envelope, dispondo de determinadas chamadas de sistema para realizar sua tarefa.

Além dos programas seqüenciais responsáveis pelo reconhecimento do usuário e pela interpretação da Linguagem de Comando, são necessários utilitários, tais como editor de texto, compiladores, programa para cópia de arquivos, programa para impressão de arquivos, e assim por diante.

SUBSISTEMAS DO PROGRAMA CONCORRENTE

O processo envelope, para executar as chamadas de sistema requeridas pelos programas seqüenciais, pode utilizar os serviços dos demais subsistemas do programa concorrente. Estes subsistemas geram o compartilhamento de recursos entre os diversos processos envelope.

Os sete subsistemas existentes são responsáveis pela gerência das seguintes áreas:
- Controle do tempo;
- Controle de volumes;
- Controle de terminais;

- Troca de mensagens;
- Sistema de Arquivos;
- Impressão;
- Execução subterrânea.

O sistema operacional mantém um relógio interno, atualizado a cada segundo. É possível, através das chamadas de sistema apropriadas, consultar o relógio, além de acertar dia e hora. O subsistema de controle do tempo é responsável por sua atualização, além de utilizá-lo no controle do tempo de execução de programas seqüenciais que podem, opcionalmente, ter um limite de tempo de execução, após o qual são abortados.

O subsistema de controle de volumes mantém tabelas sobre o estado de cada unidade de disco. Informações como quais unidades de disco estão ativas, qual a identificação do volume montado, entre outras, são mantidas nestas tabelas. Toda vez que uma unidade de disco é ativada ou desativada, estas tabelas são atualizadas.

O subsistema de controle de terminais mantém tabelas sobre o estado de cada terminal. Informações como que usuários estão conectados, em que terminais, entre outras, são mantidas nestas tabelas. Toda vez que um usuário inicia ou termina uma sessão de trabalho, estas tabelas são atualizadas.

Dentro do sistema operacional existe o subsistema que fornece o serviço de troca de mensagens entre usuários correntemente conectados. Este serviço permite que usuários enviem e recebam pequenas mensagens através do próprio sistema.

O Sistema de Arquivos implementa uma estrutura de diretórios em árvore, permitindo facilidades do tipo link e diretório corrente /PET 85/. Ele é responsável pela gerência de espaço em disco e pela criação, remoção e alteração de diretórios e arquivos de dados. O Sistema de Arquivos também é responsável pelo controle do acesso à arquivos, bem como pelo método de acesso aos dados de um arquivo, inclusive arquivos compartilhados entre usuários.

O subsistema de impressão realiza as funções de um output spooler /DEI 84/. O texto a ser impresso é colocado em disco pelo processo envelope e

posteriormente enviado para a impressora por este subsistema. Este mecanismo permite que diversos programas seqüenciais realizem suas impressões simultaneamente, mesmo que o sistema disponha de uma única impressora.

Existe, como recurso do sistema, um conjunto de processos envelopes que não estão vinculados a nenhum terminal. Um programa seqüencial pode, através da chamada de sistema adequada, alocar um destes processos e determinar que programa seqüencial ele deverá executar. Este tipo de execução é denominada execução em background. O sistema operacional não implementa nenhuma forma de comunicação ou sincronização entre processos envelopes.

O SISTEMA DE ARQUIVOS

O Sistema de Arquivos do SOMBRA está organizado na forma de uma árvore. Os nodos da árvore correspondem à arquivos e diretórios do sistema. Os diretórios contém nomes e endereços de outros arquivos ou diretórios. Arquivos são utilizados para armazenar dados do próprio sistema ou de usuários. As folhas da árvore do Sistema de Arquivos correspondem à arquivos ou diretórios vazios. Os nodos intermediários da árvore são necessariamente diretórios.

Cada arquivo ou diretório possui um nome e uma identificação. O nome pode ter de 1 a 8 caracteres. O nome serve para identificar o arquivo ou diretório dentro do seu diretório pai na árvore, ou seja, dentro do diretório onde ele aparece como entrada. Entretanto, é possível a existência de dois diretórios ou arquivos distintos, com o mesmo nome, mas localizados em diferentes diretórios. Para identificar completamente um diretório ou arquivo dentro da árvore, utiliza-se o caminho sobre a árvore que, a partir da sua raiz, leva até o diretório ou arquivo em questão. Este caminho é denominado de identificação do diretório ou arquivo. Uma notação própria é utilizada para escrever uma identificação. Os nomes dos diretórios que formam o caminho a partir da raiz da árvore, juntamente com o nome do diretório ou arquivo considerado, são dispostos em seqüência, separados por uma barra à esquerda (\). Uma barra à esquerda deve preceder o resultado da concatenação.

O Sistema de Arquivos do SOMBRA foi projetado para tirar proveito das características da situação mais genérica. Em outras palavras, sua

estrutura foi organizada para uma instalação alvo onde estão disponíveis diversas unidades de disco, e estas unidades podem ser acessadas simultaneamente. A impossibilidade de acesso simultâneo às unidades de disco, ou a existência de uma única unidade, constituem casos particulares da estrutura proposta. Cada unidade de disco permite o acesso a um volume, que o SOMBRA supõe removível. O fato do volume ser realmente removível ou não é indiferente para o sistema.

Cada volume tem o seu conteúdo organizado como uma subárvore completa do Sistema de Arquivos. Isto significa que um diretório não pode fazer referência a outro diretório ou arquivo colocado em outro volume. Todo volume é autocontido, e desconhece a existência de outros volumes.

Como os volumes são considerados removíveis, eles podem ser montados e desmontados. Montar um volume presente em uma unidade de disco significa trazer para memória os seus descritores, o que torna o seu conteúdo acessível. Por outro lado, ao desmontar um volume, os seus descritores que estavam na memória são salvos no volume, o que impede o acesso ao seu conteúdo, até que ele seja montado novamente.

O diretório raiz da árvore do Sistema de Arquivos contém uma entrada para cada volume montado. Na verdade, ele não pode ser acessado como um diretório comum e, ao contrário dos demais diretórios, não é salvo em disco. Ele somente indica quais unidades estão ativas no momento, e qual o nome do volume montado em cada unidade. Ele não é mantido pelo Sistema de Arquivos, mas pelo Controle de Volumes.

A identificação de um arquivo pode indicar o volume ao qual pertence o arquivo. Por exemplo, o arquivo cuja identificação é mostrada abaixo deverá ser procurado no volume SOMBRA1, se ele estiver montado:
\SOMBRA1\ROMULO\TESE.CP1

A identificação de um arquivo ou diretório é formada por um caminho sobre a árvore do Sistema de Arquivos. Até agora, este caminho sempre foi descrito como iniciando na raiz da árvore, com o primeiro nome identificando o volume em questão. O mecanismo de diretório corrente permite que este caminho não inicie na raiz da árvore, mas em um diretório qualquer. O usuário, através de comando apropriado, pode transformar um diretório qualquer da árvore, em seu diretório corrente. A partir deste momento, ele pode identificar arquivos ou diretórios a

partir de caminhos que iniciam na raiz da árvore, ou em seu diretório corrente. Se a barra inicial estiver presente na identificação de um arquivo ou diretório, então o caminho inicia na raiz da árvore. Se a barra estiver ausente, então o caminho inicia no diretório corrente do usuário.

O Sistema de Arquivos do SOMBRA permite que o usuário estabeleça ligações ("links") para arquivos ou diretórios. Estabelecer uma ligação significa criar em um diretório uma entrada que aponte para um arquivo ou diretório já existente. O nome associado com esta entrada pode ou não ser igual ao nome original do arquivo ou diretório apontado. As características dele continuam as mesmas, o seu dono continua o mesmo, apenas foi criado um novo caminho sobre a árvore para chegar-se até ele. Um arquivo ou diretório somente é removido da árvore quando não existir mais nenhuma ligação para ele. O Sistema de Arquivos exige que as ligações fiquem restritas a cada volume. Além disto, o diretório onde é introduzida a ligação como entrada deve estar em um nível mais baixo do que o diretório ou arquivo alvo da ligação. Isto impede a ocorrência de ciclos dentro do Sistema de Arquivos, o que dificultaria os algorítmos de procura e remoção de arquivos e diretórios.

A LINGUAGEM DE COMANDO

A Linguagem de Comando tem por objetivo ser a interface entre o usuário e o sistema operacional. Através dela o usuário requisita serviços do sistema operacional e recebe o resultado de tais serviços.

Abaixo estão algumas das construções de alto nível oferecidas pela linguagem:

- Expressões formadas através dos 16 operadores definidos;
- Construções do tipo seqüência de comandos;
- Construções do tipo se-então-senão;
- Construções do tipo laço com teste no início, teste no fim ou com número previamente determinado de iterações;
- Variáveis.

Além das construções citadas, a linguagem oferece comandos com as seguintes funções:

- Obtém a data atual;

- Obtém a hora atual;
- Obtém o estado das unidades de disco;
- Obtém o estado dos terminais e processos envelope destinados a execução em background;
- Envia mensagem;
- Obtém ou altera diretório corrente;
- Lista conteúdo de diretório;
- Cria diretório;
- Remove arquivo, diretório ou ligação;
- Dispara a execução de um lote de comandos;
- Dispara a execução de um lote de comandos em background;
- Executa um programa seqüencial;
- Realiza a desconexão do usuário;
- Aborta programa seqüencial em andamento;
- Monta volume;
- Desmonta volume;
- Bloqueia terminal;
- Libera terminal;
- Acerta o relógio do sistema.

A Linguagem de Comando é implementada por um programa seqüencial, seguindo a filosofia de máquina virtual de Brinch Hansen /HAN 77/. Os comandos do usuário são traduzidos em operadores de uma máquina virtual, que posteriormente serão interpretados. Esta filosofia permitiu que métodos utilizados na construção de compiladores /AHO 77/ /CAL 79/ fossem usados na tradução dos comandos em código da máquina virtual.

CONCLUSÕES

O sistema operacional SOMBRA, descrito neste trabalho, é composto basicamente por um programa concorrente, e pelo programa seqüencial que implementa a Linguagem de Comando.

O compilador Pascal Concorrente disponível no sistema operacional SOLO não permite a compilação de módulos em separado. A pequena memória do LABO 8034 impõe sérias restrições a capacidade do compilador. Devido a estas limitações, não foi possível realizar a integração do sistema.

O interpretador da Linguagem de Comando, programado em Pascal Seqüencial, foi depurado em um IBM-PC. Para isto, foi programado um conjunto de

rotinas capaz de simular, parcialmente, as chamadas de sistema do SOMBRA. O ILC não utiliza nenhuma facilidade própria do IBM-PC, podendo ser transportado para o SOMBRA sem nenhuma alteração. Graças a modularidade do ILC, a substituição do SOMBRA por um ambiente simulado foi necessária somente para a depuração da fase de interpretação do código virtual. O programa fonte do ILC ocupa aproximadamente 3000 linhas.

Apesar de codificado, o programa concorrente teve que ser dividido em versões parciais para que pudesse ser compilado e testado. Ele ocupa aproximadamente 5500 linhas. Diversas versões parciais foram criadas, ao mesmo tempo em que foram desenvolvidos programas e rotinas para auxílio à depuração.

Finalmente, é importante salientar que o transporte dos programas concorrente e seqüencial do LABO 8034 para a máquina SUMUS não deverá apresentar problemas. O uso de linguagens de alto nível confere elevada portabilidade ao sistema. Isto se aplica a qualquer máquina onde venha a ser implementado o Sistema Pascal Concorrente.

BIBLIOGRAFIA

/AHO 77/ AHO, Alfred & ULLMAN, Jeffrey. Principles of compiler design. New Jersey, Addison-Wesley, 1977.

/BRA 84/ BRANDÃO, M. F. R. Especificação de um sistema operacional multiusuário em Pascal Concorrente. Porto Alegre, PGCC-UFRGS, 1984. (Dissertação de mestrado).

/CAL 79/ CALINGAERT, Peter. Assemblers, compilers, and program translation. Potomac, Computer Science Press, 1979.

/DEI 84/ DEITEL, H. M. An introduction to operating systems. New Jersey, Addison-Wesley, 1984.

/HAN 77/ HANSEN, P. B. The arquitecture of concurrent programs. Englewood Cliffs, Prentice-Hall, 1977.

/MED 81/ MEDEIROS, G. C. R. Implementação do sistema Pascal Concorrente no LABO 8034. Porto Alegre, PGCC-UFRGS, 1981. (Dissertação de mestrado).

/PET 85/ PETERSON, J. & SILBERSCHATZ, A. Operating system concepts. New Jersey, Addison-Wesley, 1985.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Modelando propriedades dinâmicas de sistemas: comparação entre duas abordagens

Carlos A. Heuser

Universidade Federal do Rio Grande do Sul*
Departamento de Informática – Curso de Pós-Graduação em Ciência da Computação
Caixa Postal 1501
90001 Porto Alegre RS
Tel: (0512)218499
Telex: (051)2680 CC UF
Cirandão: 6007/UFRGS-HEUSER

Resumo

Duas abordagens de modelagem de propriedades dinâmicas de sistemas, a Formação de Procedimentos da Engenharia de Informações e as Redes de Petri, são analisadas comparativamente. A comparação considera a aplicação destas abordagens na modelagem conceitual de sistemas de informação. Na conclusão do trabalho, favorável às Redes de Petri, é discutida a utilização de Redes dentro da Engenharia de Informação.

1 Introdução

As propriedades de sistemas modelados na área de sistemas de informação, e, em geral, na área de sistemas discretos, podem ser divididas em duas grandes categorias. As propriedades **estáticas** de um sistema se referem aos **estados** que o mesmo pode assumir, enquanto que as propriedades **dinâmicas** se referem às **transições** entre os estados do sistema.

As metodologias de análise e projeto de sistemas que hoje são empregadas, desenvolvidas, em sua grande maioria, na década de 70, dão normalmente ênfase a apenas um destes tipos de propriedades, relegando o outro a um segundo plano. Assim, as metodologias provenientes da área da modelagem de banco de dados, como aquelas construídas ao redor do modelo Entidade/Relacionamento (E/R) de Chen[3], enfatizam a descrição dos estados do sistema, através de modelos de dados. Já as metodologias provenientes da área da análise e projeto de sistemas de informação, como p.ex. a análise estruturada de Gane[5], concentram-se na modelagem das propriedades dinâmicas, não ficando explicitadas as propriedades dos estados do sistema. Particularmente afetados por este estado de coisas são aqueles que desenvolvem aplicações em ambientes de integração de dados, pois eles necessitam tanto a explicitação das propriedades estáticas (o modelo de dados), que ser-

* Este trabalho possui financiamento parcial do CNPq, da FINEP e da SID Informática (Projeto Estra)

vem de base ao projeto da base de dados, quanto as propriedades dinâmicas, sobre as quais se baseia o projeto das aplicações que são executadas sobre a base de dados.

Tendo sido reconhecida esta situação, surgiram, nos últimos anos, propostas de metodologias integrando a modelagem de propriedades estáticas com a modelagem de propriedades dinâmicas. Uma destas propostas, que encontra considerável repercussão na prática, é a **Engenharia de Informação** (EI) de Finkelstein e Martin[4,1]. A metodologia é claramente voltada a dados, a informações, baseando-se na suposição, fundamental do enfoque de banco de dados, de que, em uma organização, as propriedades estáticas (o modelo de dados), tendem a ser mais estáveis contra mudanças que as propriedades dinâmicas (as aplicações executadas sobre os dados). Entretanto, a metodologia não despreza a modelagem de propriedades dinâmicas.

A modelagem de propriedades dinâmicas aparece dentro da EI em uma etapa denominada **Formação de Procedimentos** (FP), que segue às etapas de modelagem de propriedades estáticas, usualmente feitas com auxílio da abordagem Entidade/ Relacionamento. No presente trabalho, a técnica de Formação de Procedimentos é analisada criticamente, sendo detectados alguns problemas da mesma, como ferramenta de modelagem conceitual de propriedades dinâmicas. Como alternativa à Formação de Procedimentos é sugerida a utilização de **Redes de Petri**, uma técnica de modelagem de propriedades dinâmicas, largamente difundida na engenharia de hardware e que agora começa a encontrar aplicação também na engenharia de software.

2 Formação de Procedimentos dentro da Engenharia de Informação

A Engenharia de Informações, como metodologia moderna de análise e projeto de sistemas, parte do reconhecimento de que grande parte dos erros em sistemas de processamento de dados são causados por especificações incorretas, e de que o custo de correção destes erros é muito maior que o de correção de erros por outras causas (projeto incorreto, codificação incorreta, etc.). Baseada neste reconhecimento e na evidência de que o grande conhecedor da aplicação é o próprio usuário, a EI enfatiza as etapas de planejamento, análise e especificação procurando envolver ao máximo o usuário dentro destas etapas.

Fase	Etapa
Planejamento	Planejamento estratégico
Especificação	Análise de informações
	Análise de dados
	Formação de Procedimentos
	Análise do uso de dados
Projeto	Análise de distribuição de dados
	Projeto de banco de dados
	Especificação de programas

Figura 1: Visão geral da Engenharia de Informação

Nas suas fases iniciais (veja visão geral na figura 1), a EI se concentra nas propriedades estáticas, até chegar a um modelo de dados detalhado, ao final da etapa de análise de dados. Em cima deste modelo, que pode ser construído segundo a abordagem E/R, é feita, em conjunto com o usuário, a **Formação de Procedimentos**, objetivando coletar as propriedades dinâmicas do sistema.

Para dar um exemplo considera-se um sistema simplificado de registro do que acontece em um mercado de trabalho. O objetivo do sistema é registrar a oferta de vagas de trabalho e de mão-de-obra, bem como registrar os contratos de trabalho que são fechados. As etapas iniciais da Engenharia de Informações, que antecedem à Formação de Procedimentos, poderiam conduzir a um diagrama Entidade/Relacionamento como aquele apresentado na figura 2. Este diagrama informa os estados que o sistema pode assumir (as propriedades estáticas do sistema). Ele informa, por exemplo, que uma vaga pode estar fora ou dentro do mercado de trabalho, mas não em ambas situações ao mesmo tempo. Note-se que no modelo faltam as propriedades dinâmicas. Por exemplo, não está dito como uma vaga passa de fora para dentro do mercado de trabalho. Assim, tem-se uma definição das informações utilizadas no sistema modelado, sem entrar em detalhes

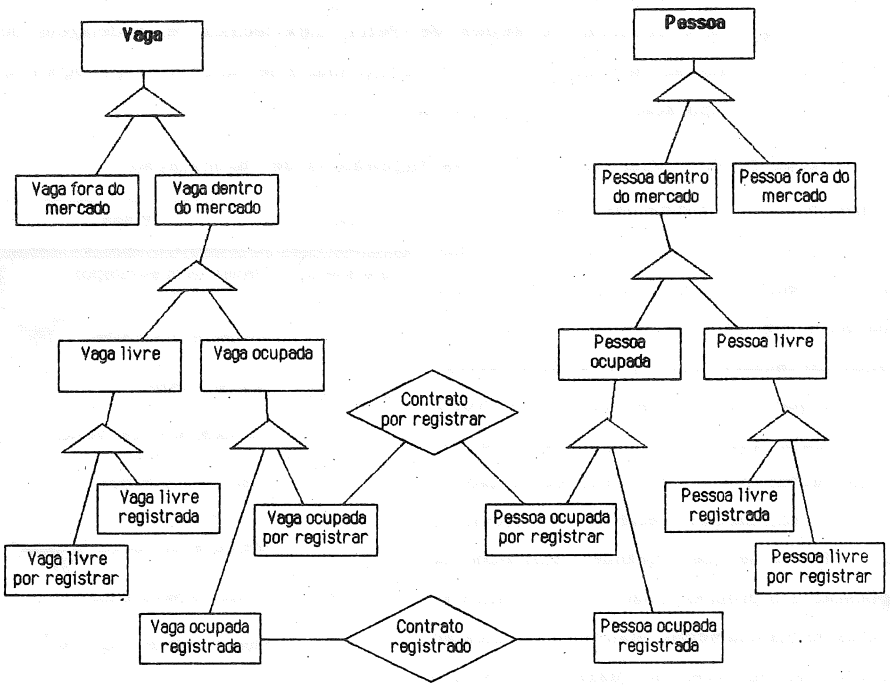


Figura 2: Diagrama E/R do mercado de trabalho

de como estas informações estão representadas em uma implementação especifica.

O objetivo da FP é o de, sobre o diagrama E/R e com ajuda do usário, identificar os **procedimentos** executados dentro do sistema. Um procedimento é composto por **eventos** (quando necessário, estes eventos serão chamados eventos-FP, para distinguí-los dos eventos definidos pelas Redes de Petri). Cada evento é uma ação isolada sobre uma entidade/relacionamento do modelo E/R ou então sobre as entradas/saídas do sistema. Um exemplo de procedimento, para o sistema de mercado de trabalho, é dado pela figura 3, o **diagrama de eventos** do procedimento que registra o fechamento de um contrato de trabalho. Neste diagrama as elipses representam os eventos e as setas indicam a sequência de acontecimentos destes eventos. Os eventos indicados por siglas na forma "EEn" (onde "n" é um número) são eventos **externos**, ou seja eventos de aquisição/exibição de informações na interface do sistema automatizado com seu meio ambiente. Os eventos indicados por siglas na forma "EXn" são eventos de **exceção**, que correspondem à detecção de um erro no procedimento. Finalmente, os eventos indicados por siglas na forma "En" são eventos **internos**, que tratam conjuntos de entidades/relacionamentos do diagrama E/R. Cada evento interno pode acessar somente um conjunto de entidades/relacionamentos.

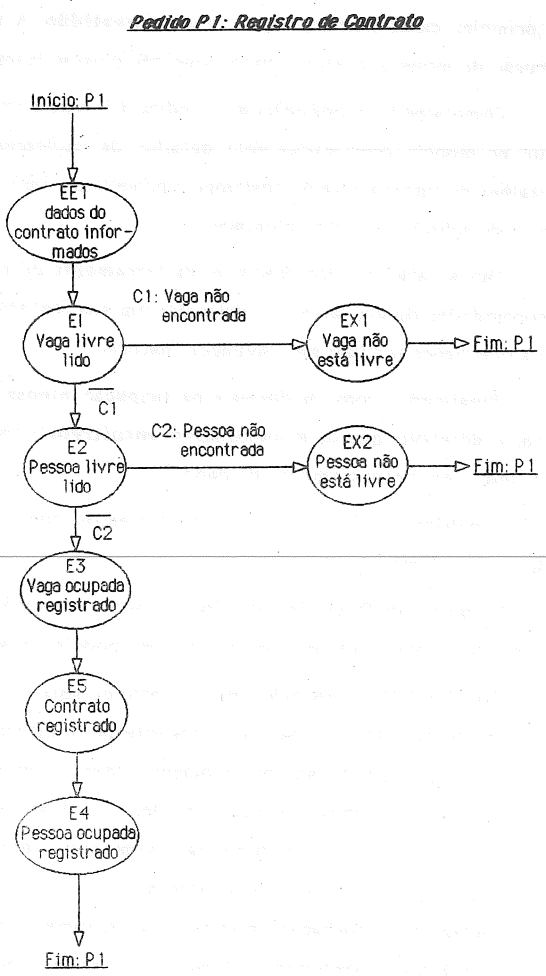


Figura 3: Formação de Procedimentos — Diagrama de eventos

Quando, partindo de um evento, existirem diversos eventos sucessores, como é o caso do evento E1 do exemplo, são indicadas condições para a escolha de um dos possíveis eventos sucessores. Uma introdução de-

talhada à FP, incluindo instruções de como executá-la, é dada por Cevallos[2].

3 Análise da Formação de Procedimentos

Para avaliar a FP é necessário que sejam relembradas as características que, ao menos idealmente[11], uma ferramenta de modelagem conceitual de propriedades dinâmicas deve possuir.

A primeira característica, básica, é a da **exatidão**. A ferramenta deve permitir a construção de modelos exatos e claros, que não deixem margem a interpretações ambíguas.

Como a palavra **conceitual** já indica, a ferramenta não deve obrigar o analista a incluir no modelo, prematuramente, detalhes de implementação. Um modelo que já contém detalhes de implementação restringe o projetista e implementador na sua liberdade de escolha de soluções de implementação.

Outra característica desejável da ferramenta de modelagem é a de permitir que as propriedades dinâmicas sejam definidas em sua **totalidade**, evitando-se, assim, o recurso a descrições textuais em linguagem natural.

Finalmente, como se deseja a participação intensa do usuário no processo de modelagem, é desejável que os modelos sejam **intuitivos** e facilmente compreensíveis, também por leigos em processamento de dados.

Analisando a Formação de Procedimentos frente a estas características, encontra-se alguns problemas:

- A Formação de Procedimentos força o analista a construir modelos que estão excessivamente próximos da implementação. Isto pode ser visto em dois pontos:
  - Na FP é feita a distinção entre a parte do sistema a implementar em computador e a parte não automatizada (eventos internos e externos, etc.). Com isso o usuário é obrigado, já prematuramente, a decidir até que ponto a sua aplicação será automatizada e fixar esta decisão dentro do modelo. Isto traz consigo diversos inconvenientes. Uma futura mudança do sistema para aumentar o escopo da automatização exigirá nova execução da Formação de Procedimentos, pois a interface do sistema mudará. A modelagem de operações manuais, não automatizadas, não é realizada dentro da Formação de Procedimentos, ficando de fora da modelagem. Da mesma forma, as ações do usuário do sistema automatizado não são modeladas. Esta falta é grave, pois se sabe o quão importante é o conhecimento das ações do usuário para a escolha de uma implementação eficiente (seqüências de ações do usuário, paralelismo ou não destas ações, calendário da aplicação, etc.).
  - A Formação de Procedimentos exige que o analista escolha uma determinada se-

quência de execução de eventos, mesmo que estes, do ponto de vista do sistema, possam acontecer independentemente. Este é o caso, no exemplo da figura 3, dos eventos E1 e E2 que podem acontecer em qualquer sequência (o mesmo vale para os eventos E3, E4 e E5). Obviamente esta é uma típica decisão de implementação. Sendo tomada prematuramente, ela pode impedir a escolha de uma sequência de eventos que leve a uma implementação mais eficiente, ou até que seja exigida por algum SGBD específico (alguns sistemas exigem a inclusão de registros antes da inclusão dos relacionamentos entre estes registros).

- Propriedades dinâmicas importantes não são levantadas pela Formação de Procedimentos. Além da falta, já citada, das operações não automatizadas, falta uma visão geral do sistema, mostrando o relacionamento entre os diversos procedimentos que compõe o sistema, mostrando sua seqüência de acontecimentos, a interdependência entre os mesmos, etc.

Resumindo, como ferramenta de modelagem de propriedades dinâmicas, a Formação de Procedimentos não preenche duas características desejáveis neste tipo de ferramentas: as características de **conceitualidade** e de **totalidade**.

4 As Redes de Petri dentro da Engenharia de Informação

Como alternativa à Formação de Procedimentos, é sugerida, neste trabalho, a utilização de **Redes de Petri** (RP). Redes de Petri, como ferramenta de modelagem de propriedades dinâmicas, já são largamente difundidas na área de hardware. Na área do software, principalmente na área de sistemas de informações, elas vêm encontrando aplicação somente nos últimos anos, principalmente depois da introdução das chamadas **Redes de Petri de alto-nível**[6,10,9] Como já indicado em 7, as Redes de Petri podem ser usadas convenientemente também para a modelagem de propriedes estáticas de sistemas, e, até mesmo, como indicado em 8, em combinação com ferramentas de modelagem de propriedades estáticas já existentes, no caso citado, em combinação com a abordagem E/R.

A figura 4 mostra uma Rede de Petri, usada em combinação com um diagrama E/R. O diagrama E/R, indicado no desenho pelas linhas com padrão pontilhado, é o da figura 2, tendo sido suprimidos alguns detalhes (as generalizações) apenas para não sobrecarregar a figura. A Rede de Petri tem como objetivo definir **eventos** (ou eventos-RP para não confundir com os eventos-FP da Formação de Procedimentos), cuja ocorrência altera o conteúdo de conjuntos de entidades e/ou relacionamentos. Estes eventos são definidos pelas **conexões** da Rede de Petri, representadas na figura por retângulos. Exemplificando, a conexão cr_j da figura 4 define, para cada vaga, um evento, cuja ocorrência resulta no desaparecimento da vaga do conjunto de entidades "Vaga fora do mercado" e no coincidente

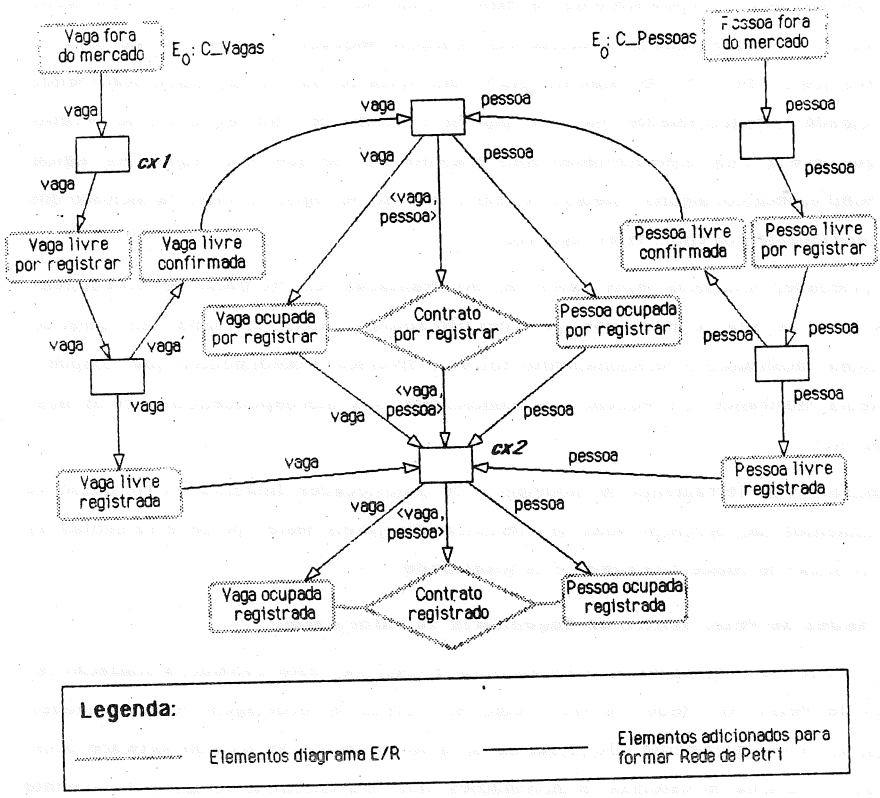


Figura 4: Rede de Petri combinada com diagrama E/R para o mercado de trabalho

aparecimento da mesma vaga no conjunto de entidades "Vaga livre por registrar". Esta conexão define, portanto, os eventos de entrada de vagas no mercado de trabalho.

A regra de funcionamento de uma Rede de Petri é simples: Um evento pode acontecer (se diz, está **habilitado**) quando as suas entidades/relacionamentos de entrada (aquelas, as quais com o acontecer do evento desaparecem) estiverem presentes e quando as suas entidades/relacionamentos de saída (as que desaparecem) estiverem ausentes. Não está dito, no modelo, quando um evento acontece, apenas que ele pode acontecer. Eventos que não possuem entidades/relacionamentos de entrada ou saída comuns são independentes entre si e podem acontecer concorrentemente, caso ambos estejam habilitados.

O modelo tem um estado inicial que é indicado através das anotações "E_0:". No caso do mercado de trabalho, no estado inicial, todas as vagas e todas as pessoas estão fora do mercado (os conjuntos de entidades "Vaga fora do mercado" e "Pessoa fora do mercado"

contém, respectivamente, os conjuntos de entidades "C_Vagas", conjunto de todas as vagas, e "C_Pessoas", conjunto de todas as pessoas). Todos os demais conjuntos de entidades/ relacionamentos estão vazios no estado inicial.

Feita esta introdução sucinta às Redes de Petri (para maiores detalhes, ver a literatura) pode-se passar a uma discussão comparativa das Redes em relação a Formação de Procedimentos. Como ferramenta de modelagem conceitual de propriedades dinâmicas de sistemas, as Redes de Petri apresentam uma série de vantagens sobre a Formação de Procedimentos:

- Utilizando as Redes de Petri, o analista não é obrigado a fazer uma divisão prematura do sistema em parte a automatizar e meio ambiente. Desta forma, o sistema completo pode ser modelado, o que permite que o pessoal de projeto e implementação obtenha uma melhor visão de qual é o problema a ser resolvido com a automatização.
- Quando completada a análise do problema, isto é, quando a Rede de Petri estiver completa, aí sim, é possível passar a fase de especificação, cujo primeiro passo consta exatamente em dividir o sistema em parte a automatizar e meio ambiente, como feito na figura 5. A partir deste momento é que ficam determinados os conjuntos de entidades/

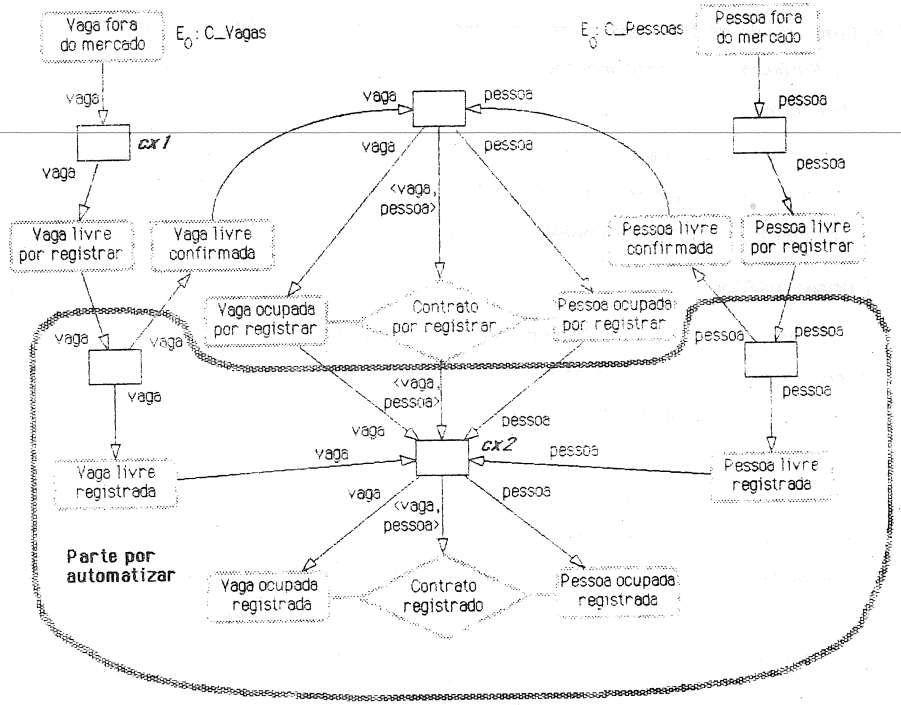


Figura 5: Rede de Petri como especificação

relacionamentos que serão implementados pela base de dados em computador (no caso do exemplo, os conjuntos cujo nome termina com a palavra "registrada": "Vaga livre registrada", ...), os conjuntos de entidades/relacionamentos que compõe a interface entre o usuário e a parte automatizada (no exemplo, os conjuntos de entidades/relacionamentos cujos nomes terminam com "por registrar" ou "confirmada"), e, finalmente os conjuntos de entidades/relacionamentos que ficam no meio ambiente do sistema automatizado. Note-se, que assim a Rede de Petri é estável quanto a mudanças no posicionamento da interface do usuário, isto é quanto a mudanças na abrangência da automação do sistema, o que não é o caso dos diagramas obtidos pela Formação de Procedimentos.

- Com as Redes de Petri trabalha-se em um nível mais conceitual, mais abstrato, do que com a Formação de Procedimentos. Não há, nas Redes de Petri, a restrição de que um evento tenha, como entrada/saída, uma entidade/relacionamento apenas, nem a obrigação de seqüencializar todos eventos. Assim, o registro do fechamento de um contrato, que na Formação de Procedimentos havia exigido todo procedimento da figura 3, é representado na Rede de Petri por um tipo de evento apenas, definido pela conexão cr_2.
- Com a Rede de Petri obtém-se uma descrição completa do sistema, o que não é o caso da Formação de Procedimentos, na qual falta o relacionamento entre os diversos procedimentos.
- A médio e longo prazo as chances de surgirem ferramentas automatizadas para a construção e principalmente para a análise de modelos é maior nas Redes de Petri, pois sobre elas estão sendo realizados os maiores esforços de pesquisa e desenvolvimento.

5 CONCLUSÕES

O artigo compara duas ferramentas de modelagem de propriedades dinâmicas de sistemas, a Formação de Procedimentos e as Redes de Petri, considerando sua utilização dentro de uma metodologia abrangente de análise e projeto de sistemas, a Engenharia de Informações, e levando em conta as carcterísticas desejáveis de ferramentas deste tipo[11].

Um ponto que propositalmente não foi discutido na comparação, por ficar em um plano muito subjetivo, é o da facilidade de aprendizado das ferramentas por parte de usuários leigos. Fica registrado, apenas, que ambas ferramentas se baseiam em alguns poucos conceitos simples e que ambas apresentam seus modelos em termos de diagramas, características que normalmente facilitam o aprendizado e compreensão.

Resumindo, a comparação entre as duas ferramentas mostrou que com as Redes de Petri é possível obter modelos em um nível de abstração mais adequado às necessidades

da Engenharia de Informações, bem como, que com as Redes podem ser obtidos modelos mais completos do problema a resolver.

Frente a estas conclusões, é sugerida a substituição, dentro da Engenharia de Informações, da Formação de Procedimentos por Redes de Petri combinadas com diagramas Entidade/Relacionamento. Entretanto, para permitir esta substituição, mais algumas investigações são necessárias:

- O trabalho de integração das Redes de Petri com a abordagem Entidade/Relacionamento iniciado em 8 deve ser completado. Falta investigar como modelar a alteração de atributos, bem como a forma de tratar algumas extensões à abordagem original de Chen que aparecem em muitos dialetos do E/R, como a abstração, a generalização, e outros.
- Deve ser desenvolvida uma técnica, envolvendo usuários leigos, para obtenção da Rede de Petri, a partir do diagrama E/R. Um caminho promissor, que pode ser seguido, é o de procurar adaptar ao caso da Redes de Petri a técnica descrita em 2 para a Formação de Procedimentos.

Finalmente, cabe mencionar, a possibilidade de se usar a Formação de Procedimentos em combinação com as Redes de Petri. Dentro desta abordagem, as Redes de Petri seriam utilizadas como ferramenta de modelagem conceitual, para análise e especificação. Confeccionada a Rede de Petri do sistema, usar-se-ia a Formação de Procedimentos para o **projeto** de procedimentos a serem implementados, provavelmente de forma bastante direta, em linguagem de 4ª geração. Neste caso, a Formação de Procedimentos seria uma ferramenta de projeto e não de modelagem conceitual, e seria usada estritamente pelo pessoal de processamento de dados, exatamente por envolver problemas de implementação. Dentro desta abordagem, poder-se-ia considerar o procedimento do diagrama de eventos da figura 3 como sendo uma implementação para os eventos-RP definidos pela conexão cr_2 da Rede de Petri da figura 5.

Bibliografia

1. Bastos Fº, E. A.: "Engenharia de Informações" - Uma proposta integrada para a informatização das organizações. In: *Anais do XX Congresso Nacional de Informática* São Paulo, SUCESU-SP (1986) 149-155.
2. Cevallos, J.T.: *Manual da Formação de Procedimentos* (Publicação interna) Porto Alegre, Cia. Proc. Dados PROCERGS (1986).
3. Chen, P.P.-S.: The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems* 1:1 (1976) 9-36.
4. Finkelstein, C. & Martin, J.: *Principles of Information Engineering* New Jersey, Prentice-Hall (1982) 2 vol.

5. Gane, C. & Sarson, T. *Análise Estruturada de Sistemas* Rio de Janeiro, Livros Tecnicos e Cientificos (1983).

6. Genrich, H.J. & Lautenbach, K.: System Modelling with High-level Petri Nets. *Theoretical Computer Science* vol. 13 (1981) 109-136.

7. Heuser, C.A. & Richter, G.: On the Relationsip between Conceptual Schemata and Integrity Constraints on Databases. In: Steel, T.B. & Meersman, R. (eds.): *Database Semantics (DS-1)* Noth-Holland, Amsterdam (1986) 27-39.

8. Heuser, C.A.: Integrando Propriedades Estáticas e Dinâmicas no Modelo Conceitual. In: *2º Simpósio Brasileiro de Banco de Dados* Porto Alegre, UFRGS/SBC (1986).

9. Heuser, C.A.: Análise Estruturada de Sistemas com Redes de Petri. In: *Anais do XX Congresso Nacional de Informática* São Paulo, SUCESU-SP (1986).

10. Richter, G. & Durchholz, R.: IML-inscribed High-level Petri Nets. In: Olle, T.W. et. al. (eds.): *Informations Systems Design Methodologies: a Comparative Review.* Amsterdam, North-Holland (1982) 363-375

11. Van Griethuysen, J.J. (ed.): *Concepts and Terminology for the Conceptual Schema and the Information Base* Publication Number ISO/TC97/SC5-N695. New-York, ANSI (1982).

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

"Un Sistema de Compilación, Ejecución y Análisis de Redes de Petri Extendidas para Computadoras IBM-PC Compatibles"

Autores: Ludueña Verónica
Oyarzabal Juan Carlos

Director: Gallard Raúl Héctor

UNIVERSIDAD NACIONAL DE SAN LUIS
Ejército de los Andes 950, Local 106 - 5700 San Luis ARGENTINA
Teléfonos (0652) 25882/25108/26744/26747/26888
Telex 58125 UNSL AR

RESUMEN

En esta presentación se describe la implementación de un Sistema para la compilación, ejecución y análisis de Redes de Petri Extendidas sobre IBM-PC compatibles.

El compilador admite como entrada la definición de una Red de Petri, en forma gráfica o explicitandolo como ente matemático.
El proceso de compilación produce una representación interna ejecutable de la red.

El ejecutor procede a ejecutar la representación interna de la Red de Petri en forma visible (gráfica) o invisible (sin salida gráfica) al usuario, según su elección.

Bajo ambas opciones el analizador responde a las preguntas del análisis, recurriendo a las conocidas técnicas del árbol de alcanzabilidad o de las ecuaciones matriciales.
En el caso de utilizar la definición extendida de ejecución de una red podrán obtenerse respuestas adicionales a las preguntas del análisis.

1. INTRODUCCION

Este trabajo nace como consecuencia de la tarea de los autores y del grupo de Sistemas Operativos de la U.N.S.L. en el estudio e implementación de Software para el control de sintaxis, ejecución y análisis de Redes de Petri.

El Sistema desarrollado sobre una IBM-PC Compatible, en Quick Basic, constituye una facilidad automática para la modelización de sistemas que exhiban actividades concurrentes y asincrónicas.

Además, la implementación de la definición extendida de la ejecución de una Red de Petri, en base a un trabajo preliminar publicado en 1987 [8] amplía su capacidad de modelización permitiendo una más fiel representación del comportamiento de sistemas reales y una mejor comprensión de

los mismos, al responder a un conjunto importante de preguntas para las cuales el enfoque tradicional no tiene respuesta.

2. SOPORTES DE LA IMPLEMENTACION

Se decidió implementar el sistema sobre una computadora IBM PC AT compatible a efectos de hacer altamente accesible, este software, a una variedad de usuarios.
El equipo cuenta con 1 Mb de memoria principal, dos discos winchester de 20 Mb cada uno, impresora y tarjeta EGA. No obstante menores requerimientos son necesarios para correrlo, existiendo versiones que pueden ejecutarse en configuraciones mínimas de XT compatibles.

Se decidió utilizar un Compilador Quick Basic por la bondad de este lenguaje en su interface gráfica, por su satisfactoria estructuración, por sus facilidades para el desarrollo de Software y por su velocidad de ejecución.

3. CONCEPTOS PREVIOS

Una Red de Petri es un multigrafo, bipartito, dirigido, que tiene dos tipos de nodos: "sitios" y "transiciones". Los primeros modelizan las condiciones (estado) del sistema y las últimas modelizan los eventos que pueden darse en el mismo.
Sitios y transiciones se vinculan a través de arcos, conformando asi una estructura.

En los sitios pueden residir "tokens", en este caso se dice que la red esta marcada y se llama "marca" a una particular distribución de tokens en la red.
La distribución de tokens gobierna la ejecución de una red, habilitando o deshabilitando transiciones y permitiendo o no, consecuentemente, su disparo.
La ejecución de la red consiste en una secuencia de transiciones de estado producida por el disparo de transiciones.
Más precisamente:

Dada una Red de Petri marcada $M = \langle S,u \rangle$

donde

u es una marca inicial y S la estructura definida por;

$S = \langle P,T,I,O \rangle$

con
- $P = \{p1,p2,\ldots,pn\}$ Conjunto de n sitios, $n \geq 0$
- $T = \{t1,t1,\ldots,tm\}$ Conjunto de m transiciones, $m \geq 0$
- $I : T \dashrightarrow P'$ Función de entrada
- $O : T \dashrightarrow P'$ Función de salida
- P' Conjunto de todas las bolsas posibles formadas con elementos de P.

el espacio de estados R(S,u) de la red, tiene por elementos a todas la marcas alcanzables desde dicha marca inicial u.

Este conjunto de alcanzabilidad, R(s,u), tiene una representación finita por medio del árbol de alcanzabilidad. Cada nodo de dicho árbol representa una marca alcanzable, o un conjunto de marcas alcanzables si la red es no acotada (presencia del símbolo w en el árbol).

Además para cada uno de estos nodos se determina unívocamente el conjunto Tau contenido en T, de transiciones habilitadas. Esto es cierto aun en los casos para los cuales en el nodo en cuestión se halle presente el símbolo w, pues la presencia de "tokens" extra en un sitio no afecta la habilitación ni el disparo de las transiciones.

En consecuencia para una marca particular (estado de la red) existe un único conjunto Tau de transiciones que pueden disparar.

Como ejemplo, la siguiente red modeliza la sincronización entre un proceso productor y otro consumidor a través de un buffer no acotado:

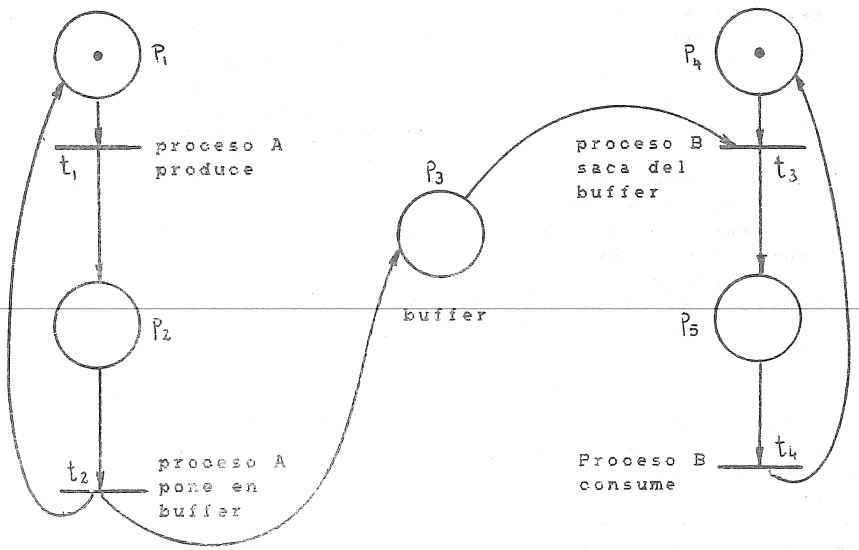


Figura 1

Habiendo remarcado estos conceptos se describen a continuación los componentes del software.

4. COMPONENTES DEL SOFTWARE

4.1. EL COMPILADOR

Este componente produce una representación interna ejecutable de la Red de Petri admitiendo la definición de la red en forma gráfica o como ente matemático.

4.1.2. ENTRADA DE LA DEFINICION DE LA RED EN FORMA GRAFICA

La entrada en forma gráfica se realiza por medio de comandos implementados a través de las teclas de función del teclado IBM compatible.

A continuación se describe la gramática (1) del lenguaje de definición de la red en forma gráfica:

```
                    <grafo> ::= <dibujo de nodos>
          <dibujo de nodos> ::= <trans. horiz.> | <sitio> |
                                <trans. vert.> |
                                <trans. sudoeste> |
                                <trans. sudeste> |
                                <cambio a dibujo de arcos>
              <trans. hor.> ::= F1
                    <sitio> ::= F2
             <trans. vert.> ::= F3
          <trans. sudoeste> ::= F4
           <trans. sudeste> ::= F5
<cambio a dibujo de arcos> ::= F6 <dibujo de arcos>
          <dibujo de arcos> ::= <punto inicial> |
                                <punto intermedio> |
                                <punto final común> |
                                <punto final inhibidor> |
                                <redibujar nodos> |
                                <cambio a marca inicial>
            <punto inicial> ::= F1
        <punto intermedio> ::= F2
      <punto final común> ::= F3
  <punto final inhibidor> ::= F4
        <redibujar nodos> ::= F5 <dibujo de nodos>
<cambio a marca inicial> ::= F6 <marca inicial>
           <marca inicial> ::= <entero sin signo>
                               <marca inicial> |
                               <entero sin signo>
```

El término "trans. sudoeste" (o sudeste) describe una transición con inclinación de 45 grados orientada al sud oeste (sud este).
El proceso de entrada, que incluye el análisis sintáctico, se realiza en forma interactiva y amigable al usuario, implementado por medio de ventanas que lo guian acorde con sus necesidades y siguiendo pasos lógicamente estructurados (dibujo de nodos, dibujo de arcos, redibujo de nodos y consecuentemente de arcos).
La salida del compilador en este caso es una representación gráfica de la red en la pantalla y una traducción, transparente al usuario, de las estructuras de información para el soporte de la red (multilistas encadenadas).

(1) Tal como ocurre con los lenguajes de programación, donde las gramáticas tipo 2 no permiten controlar todas las características del mismo, se hace necesaria aqui la existencia de tablas de símbolos para efectuar los controles no provistos por la gramática. De esta forma se controla la asociación de marcas a sitios definidos en la red y sólo a ellos.

4.1.3. ENTRADA DE LA DEFINICION DE LA RED COMO ENTE MATEMATICO

La entrada como ente matemático se hace simplemente definiendo una estructura de Red de Petri y una marca inicial ⟨S,u⟩.
A continuación se describe la gramática (2) del lenguaje de definición de la red como ente matemático:

```
        ⟨Red de Petri⟩ ::= ⟨estructura⟩ (⟨marca inicial⟩)
          ⟨estructura⟩ ::= ⟨sitios⟩ ⟨transiciones⟩
                           ⟨fción de entrada⟩
                           ⟨fción de salida⟩
                ⟨sitios⟩ ::= ⟨cantidad de sitios⟩ | ⟨lambda⟩/
          ⟨transiciones⟩ ::= ⟨cantidad de trans.⟩ | ⟨lambda⟩/
     ⟨fción de entrada⟩ ::= I⟨id. trans.⟩=⟨bolsa de sitios⟩ |
                           ⟨lambda⟩/
      ⟨fción de salida⟩ ::= O⟨id. trans.⟩=⟨bolsa de sitios⟩ |
                           ⟨lambda⟩/
   ⟨cantidad de sitios⟩ ::= ⟨entero sin signo⟩/
   ⟨cantidad de trans.⟩ ::= ⟨entero sin signo⟩/
                ⟨lambda⟩ ::=
             ⟨id. trans.⟩ ::= ⟨entero sin signo⟩
      ⟨bolsa de sitios⟩ ::= ⟨entero sin signo⟩,
                           ⟨bolsa de sitios⟩ |
                           ⟨entero sin signo⟩/
         ⟨marca inicial⟩ ::= ⟨entero sin signo⟩,
                           ⟨marca inicial⟩ |
                           ⟨entero sin signo⟩ | ⟨lambda⟩
```

Asi por ejemplo dada la Red de Petri M = ⟨S,u⟩ con

S = ⟨P,T,I,O⟩ donde

P = {p1,p2,p3,p4}
T = {t1,t2}

I : T ---⟩ P' tal que

I(t1) = {p1}
I(t2) = {p3,p3}

O : T ---⟩ P' tal que

O(t1) = {p1,p3}
O(t2) = {p2,p4,p4}

y u =(1,0,0,0)

sería especificada de la siguiente forma:

4/2/I1=1/I2=3,3/O1=1,3/O2=2,4,4/(1,0,0,0)

La salida del compilador en este caso consiste simplemente en la traducción, de la entrada a las estructuras de información para soportar la red.

(2) Al igual que en (1) las tablas de símbolos permitirán controlar la asociación de marcas a sitios definidos en la red y sólo a ellos, asi como también la correspondencia establecida por las funciones de entrada y salida entre transiciones y sitios definidos por la estructura y sólo entre ellos.

4.2. EL EJECUTOR

Este módulo es el encargado de implementar la ejecución de la red previamente compilada.

En ambos casos, entrada gráfica o como ente matemático, el ejecutor permite al usuario optar por la acotación de sitios a su elección y por efectuar o no tests de conservatividad y vitalidad para un número especificado de disparos de transiciones.
También se ha previsto registrar opcionalmente la secuencia de disparos y/o la de estados por los que transita la red, que pueden ser listadas luego de la ejecución.
Una vez finalizada la ejecución (por haber ejecutado el número de disparos especificados o haber llegado a un estado de muerte) se puede optar por una nueva ejecución para una nueva marca inicial y un distinto número de disparos.
También, opcionalmente, podrá ejecutarse la red acorde con la extensión en la definición de la ejecución (Ver punto 4.3.1.1).

En el caso de entrada gráfica, la red muestra en pantalla el cambio de las marcas, la habilitación, deshabilitación y el disparo de transiciones mediante un adecuado cambio de colores.
Los tokens se muestran como pequeños círculos, si la cantidad de ellos en un sitio es menor o igual a 9, o como un entero de hasta tres dígitos en otro caso.
La ejecución gráfica puede hacerse en modo "trace" (paso a paso) o automáticamente, con una demora elegida por el usuario entre un disparo y el siguiente.
En este caso una vez ejecutada la red, también puede re-entrarse en la etapa de dibujo de nodos (y consecuentemente de arcos) para introducir modificaciones de interés antes de una nueva ejecución.
También, opcionalmente, pueden salvarse en disco las estructuras soporte y las coordenadas de los nodos del grafo para un posterior re-dibujo automático y su correspondiente ejecución y análisis.
Por último, el sistema se provee con algunas redes previamente compiladas y almacenadas en disco que representan casos típicos de sincronización de procesos. Esta facilidad es de utilidad para el principiante en el tema.

4.3. EL ANALIZADOR

Este módulo responde a las preguntas del análisis, opcionalmente, mediante una de dos técnicas; el árbol de alcanzabilidad y las ecuaciones matriciales.

4.3.1. ARBOL DE ALCANZABILIDAD

El árbol de alcanzabilidad representa todos los estados posibles por los que transita la red y describe además las secuencias posibles de disparos que hacen alcanzables tales estados. Por ejemplo, el siguiente árbol corresponde a la red de la la Figura 1:

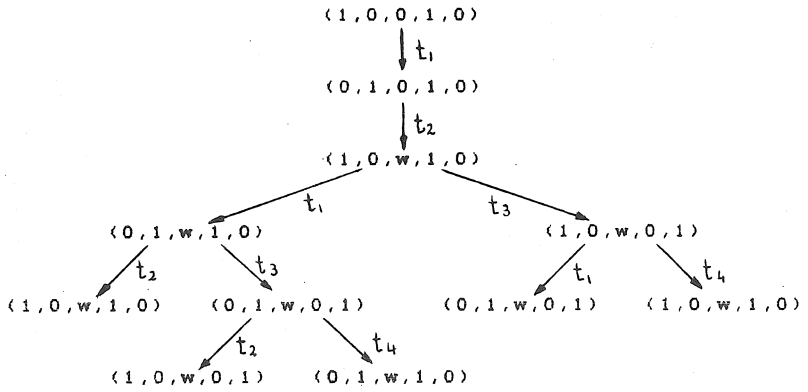


Figura 2

donde:

$R(S,u) = \{ (1,0,w,1,0), (0,1,w,1,0), (1,0,w,0,1), (0,1,w,0,1) \}$

con $w \geq 0$

En general el árbol de alcanzabilidad de una Red de Petri es un árbol donde el semigrado exterior de cada nodo es variable dependiendo del número de transiciones habilitadas en cada estado de la red.
A los efectos de economizar almacenamiento se ha recurrido al método de las transformadas de Knuth para encontrar una representación del árbol como árbol binario.

Por medio de la técnica del árbol de alcanzabilidad pueden responderse las preguntas del análisis en lo referente a: seguridad, acotabilidad y cobertura. En muchos casos también es posible responder a las preguntas de alcanzabilidad y vitalidad (Peterson [4]).

4.3.1.1. EXTENSION DE LA DEFINICION DE LA EJECUCION, SU IMPLEMENTACION VIA EL ARBOL DE ALCANZABILIDAD

Según el trabajo preliminar [8] se ha implementado esta extensión acorde a lo siguiente:

i)

En la rutina de generación del árbol de alcanzabilidad se extiende el concepto de nodo a fin de que cada uno de ellos esté constituido por el par:

(u,H) = (Marca corriente, Vector de Habilitación Corriente)

ii)

Se cuenta además con otra rutina que genera, recorriendo el árbol y en base al conocimiento del usuario, el correspondiente vector de ponderación W para ese estado de la red, a partir del cual se construirá el vector de probabilidades W' el que reemplazara en el nodo en cuestión al vector de habilitación, quedando ahora el nodo como sigue;

Nodo del árbol = (u,W')
= (Marca corriente, Vector de Prob. asociado con la marca)

iii)

Por último, existe una rutina de ejecución alternativa de la Red de Petri de modo que en cada estado de la misma, se recupere el vector W' asociado a la marca corriente para luego invocar a otra rutina que aplica el método de Montecarlo a fin de decidir que transición será disparada.

4.3.2. ECUACIONES MATRICIALES

Por medio de la búsqueda de soluciones a ecuaciones matriciales, que describen la estructura de una Red de Petri, pueden abordarse los problemas del análisis en lo concerniente a la alcanzabilidad y la conservatividad estricta y no estricta.
En este último caso, si existe solución, se provee un vector no nulo de ponderación asociado a los sitios para el cual la red se mantiene conservativa ([4] Peterson).

Esta técnica ha sido implementada utilizando una variante del método de Gauss.

5. CONCLUSIONES

El presente artículo resume las características esenciales del Trabajo Final, para optar por el Título de Licenciado en Cs de la Computación, de los autores.

Vale aqui remarcar otros puntos de interés:

1)

El sistema constituye una facilidad versatil para encarar la modelización de sistemas, computacionales o no, que exhiban actividades asincrónicas y concurrentes.
Esto se logra a través de salidas gráficas, si la cantidad de nodos de cada tipo no es mayor que 50.
Las salidas consisten en la visualización de la ejecución de la red y en el diagnóstico de la misma a través del análisis.

Otra opción permite la ejecución en memoria, principal o secundaria, no visualizada de la red. Esto se obtiene, cuando la misma se entra como ente matemático, permitiéndose entonces ejecutar redes de mayor envergadura, liberando al usuario de las restricciones impuestas por el tamaño de la pantalla.

En ambos casos podrán obtenerse listados sobre los resultados de la ejecución y del análisis.

2)

A través de la implementación de la extensión tal como lo plantea el autor del artículo original [8], se permite un comportamiento del modelo más granularmente afín con el estado del sistema modelizado en cada instancia de la red.

En consecuencia una acumulación de tokens en un sitio de la red, modelizaría ahora más fielmente la formación de una cola cuyos parámetros pueden registrarse para análisis posteriores.

Además, La asignación de una probabilidad de disparo igual a 1 ó 0 para una de las transiciones habilitadas, describiría naturalmente una política de prioridades sin necesidad de recurrir al uso de arcos inhibidores.

Con respecto al problema de la alcanzabilidad se está ahora en condiciones de responder no sólo que cierto estado es alcanzable, sino que además se precisaría más la respuesta diciendo con qué probabilidad se arribaría a ese estado luego de k disparos.
Esto es asi pues,

si u' = d(u,Sigma)

donde d es la función extendida de próximo estado y Sigma es una secuencia de disparos de longitud k, la probabilidad de arribar a u' estaría dada por el producto de las probabilidades asociadas con cada transición disparada, en las sucesivas marcas (estados) por los que pasa la red.

6. RECONOCIMIENTOS

El presente trabajo no hubiera sido posible sin la cuota de motivación que sobre el mismo aportara el Dr Alan J Harget de la University of Aston in Birmingham (U.K.) quién en 1982 introdujo al director en el tema.

Se agradece también en especial a las Licenciadas Susana Graciela Barbenza y Mirtha Susana Escudero quienes desarrollaron una versión inicial de Software de Redes Petri para la PDP-11/34 que los autores han extendido y convertido a IBM PC compatibles.

En general se agradece también las sugerencias aportadas por distintos integrantes del grupo de proyecto en Sistemas Operativos de la Universidad Nacional de San Luis.

7. BIBLIOGRAFIA

[1] C. Petri, "Kommunikation mit Automaten" , Ph. D. dissertation, University of Bonn, West Germany, 1962.

[2] J. Sifakis, "Use of Petri Nets for Performance Evaluation",Procedures of the Third International Workshop on Modelling and Performance Evaluation of Computer Systems, North Holland 1977, pages 75-93.

[3] J. Peterson, "Petri Nets", Computer Surveys, Volume 9, Number 3, September 1977.

[4] J. Peterson, "Petri Net Theory and the Modeling of Systems", Prentice Hall, 1981.

[5] H. J. Genrich and K. Lautenbach, "System Modelling with High-Level Petri Nets", Theoretical Computer Science 13, 1981, pages 109-136.

[6] R. Gallard, "A Set of Petri Net Routines for System Modeling", MSc dissertation, University of Aston in Birmingham, Birmingham, U.K., July 1982.

[7] M. Escudero, S. Barbenza, R. Gallard, "Un Sistema para la Definición, Ejecución y Análisis de Redes de Petri" 15 JAIIO, U.N.S, Nov 1985.

[8] R. Gallard, "An Extension in the Definition of a Petri Net Execution", Computer Journal, Vol. 30, Nro 1, Feb. 1987.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

Convergence Properties of Stepwise Refined Sequences of Petri Nets

Ricardo Ferré
Departament of Computer Sciences
Lund University
Sweden

Abstract

This paper consists of two parts.
In the first part, the idea of using top-down, stepwise refined sequences of Petri nets as a tool for program derivation[Fer] is motivated.
The particular need for program derivation in the special case of the construction of large programs is particularly considered in this paper, based on the work of Dana Scott and others.
The SPE classification of Lehman has been used to be able to define what a large program is, using precise and technical considerations.
The life-cycle model is applied to describe the need for change which is built in in P and E programs. The economical demand of avoiding totally new program constructions for each change, motivates an appropriate program derivation to be able to follow the logical steps that lead from a specification to its corresponding implementation in order to restrict the changes only to the necessary steps.
The inferential programming's point of view is given as the most important contribution to clarify the process of program construction.
In the second part, properties of Places/Transitions Petri nets which make them useful for program derivation are studied. The property of a place to be expanded as a whole subnet and conversely, is used here to propose the definition of a partial order for Petri nets. This partial order is proposed to be used to prove convergence properties of sequences of Petri nets, applying the partial order's topology. The limit operation can then be added to these sequences of Petri nets.The stepwise refined sequences that lead from a specification to its implementation are then increasing in this partial order.
These ideas are exemplified in an appendix by giving the Petri net derivation of a parallel regulator and a summary of the whole algorithm for the regulator described in the programming language ADA because of its suitability for the design and description of parallel processes, as an application of these ideas in real-time programming.

The SPE-classification

The definition of the concept 'large program' varies depending on the author. The intuitive definition, that can apply to many cases, takes the number of program lines as the principal factor and establishes several thousand lines as an approximate limit.
It is clear that this depends on many factors, for example the particular programming language in which these lines are written, the particular computer, the compiler, etc.
The SPE-classification[Leh] introduces more technical and precise considerations which are independent of the particular features of the implementation.

S-programs

Its function is formally defined and derivable from a specification. A problem of the real world or of the universe of the discourse is presented as a formal statement which in turn originate a program specification, this one in turn gives a program which brings the solution of the problem.

If we redefine the problem and change the specifications, then a new program must be written, even though the changes in the specifications are small, and therefore the complete correctness of the program must be proved all over again.

P-programs

The problem statement of these programs can not be precise. It is a model of an abstraction of a real-world situation.

The correctness of a solution is determined by the program environment, after comparison with it.

The concern is centered on the value and validity of the solution obtained in its real-world context and therefore can cause a change in the program itself. Typical examples are chess programs, weather prediction programs and programs for reservation system for an airline.

E-programs

The program communicates with the real world of which it has become a part, it is contained an embedded in it. The variables of its implicit function are about the same than those of P-programs, with the addition of predictive views.

In this type of programs the pressure for change is built in.

In this paper we study in particular the application of Petri net sequences to the last two types of programs (P and E).

The production of large programs must consider management aspects to organize and control the engaged personnel, software design methodologies, and the software environment which include software tools to assist the design and maintenance of the program. In the last years we have seen an increased importance of the maintenance aspects, specially in the large systems because that the statistical study has shown that 70% of the total costs are on program maintenance and only 30% in development.

The importance of these data increase in direct proportion with the role of computers in the modern society. A fail in the computer systems for defense of either of the superpowers could lead to the total extermination of the humanity.

In such large programs the following characteristics are specially important:

Reliability - A small error can be extremely costly both economically and in other aspects.

Modifiability - These programs can not be substituted often. The requirement of being easily released and modified implies that they must be well documented, explained and derived. It should be possible to change the internals of one module without requiring changesto the entire system.

Modularization - The job must be divided in modules that can be designed by a group of tens or perhaps hundreds of persons.

Verifiability - It must be possible to show the correctness of the system based on the correctness of the partial modules.

Research in software design methodologies stresses the importance of top-down, structured programming as an effective way facing the special difficulties embedded in large systems.

The requirements for appropriate theory, models, methodologies, techniques and tools are specially hard in these cases.

The life-cycle model

The P- and E-systems are long lived. The cost associated with producing such a system is so high that it is not practical to replace it with a totally new system.

During its lifetime, the system undergoes considerable modification. Both changes in the real-world environment and proved weaknesses of the system itself can determine the need of changes in the system. These considerations origin the concepts of a program's life-cycle and to techniques for life-cycle management.

The life-cycle involves three levels

- Successive generation of system sequences
- The sequence of releases of the system
- The sequence of activities that constitute the life-cycle phases of the development of an individual release.

The grossest level is constituted of system definition, implementation and maintenance.

These three phases require a careful planning in order to achieve long life software and lifetime cost effectiveness.

The laws which govern the dynamics of the evolution process indicate that project plans must be related to dynamic characteristics of the process and the system, and to the statistics of change.

Program derivation

We have already remarked the importance of proving the correctness of large programs and software systems because requirements are specially hard in these systems. It is in this aspect that the concept of program derivation arises as a logical way of showing the correctness of them.

We can define program derivation as the sequence of insights that is required to derive the implementation from straightforward specifications. By representing the program development as a sequence of programs such that the final implementation is reached from the initial specifications by a series of refinement steps we can structure the process so that we can improve both clarity and efficiency.

Conventional mathematical proofs of programs do not use to justify the structure of a program, and they do not help to develop new programs that may be similar to the program that was proved. This implies that they would be useless to the process of programming.

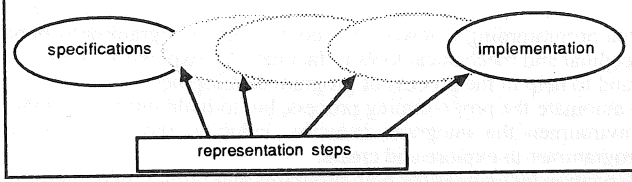
The program derivation as an idealized record of the sequence of design decisions can instead help in both these two aspects.

In the development of new similar programs the sequence of conceptual steps can be followed and changes can be made at the appropriate levels. Also the structure of the program is well clarified by the derivation.

These qualities make them particularly useful when programs must be adapted. This is specially important when we remind that the proportion of program modification, in particular in the case of large software systems is 70% with respect to design and code.

In the traditional framework modification is more difficult because it requires rediscovering concepts used under the development.

fig. 1



It is sufficient to change some steps to adapt a program to new similar specifications.

The derivation structure makes explicit the rational form of the programming structure.

Specifications are different from programs in that they do not constrain how the functionality of an algorithm is to be achieved. It has been proved that this difference is only one of degree,

and that there is a mutual dependency between specifications and programs.

The role of mathematical logic in the construction of large systems

With the aid of program derivation we can get an insight on the program's logical structure, but we can go even further : we can reason about program derivations as formal structures.

In mathematics, it can be reasoned about proofs as legitimate mathematical objects. In the same way, we could reason about program derivations as formal structures representing the evolution of programs.

It has been shown[Sch&S] that the programmers must produce provably correct software, at least in the sense that documentation is generated concerning the extent to which a program is reliable, but even more, we must tend to get a derivation with the right observations and commitments at the right points in the derivation.

Specially in large systems, documentation must be provided that shows the program derivation with all the inferential steps (commitments, deductions, analogies, simplifications) that have led to the particular implementation from the specifications so that necessary maintenance and the future releases would be easier. It is clear that there is a relation between mathematical logic and programming.

It is logical reasoning that leads the derivation steps, consciously or unconsciously.

On the other side, this leads to comparisons between mathematical theorems, proofs and processes and programs, proofs of programs and data processes.

Perhaps the best example of the need for program derivation is the proof of the four-colour theorem. The correctness of the computer program that is an essential part of this proof must also be proved.

The computer-aided check of steps in the proof of mathematical theorems is going to be a increasingly more common and useful resource, as we learn to use the power of the computer. The systems for symbolic computation configure a step in this direction.

The logicians have also shown that every closed axiom-system for arithmetic is insufficient.(Gödel). This destroyed the hope of some mathematicians of constructing indestructible, self-sufficient systems and must bring computer scientists a word of warning.

The inferential programming

We must distinguish the difference between program derivation and inferential programming.

"Inferential programming is the real, somewhat undisciplined and exploratory process of building, manipulating and reasoning about program derivations"[Sch&S].

On the other side "program derivation is the idealized and stylized record of the sequence of design decisions that leads from the specifications to the particular implementation"[Sch&S].

The programming activity involves nevertheless lots of decisions in which aesthetics, among other considerations can influence, and much backtracking, experimentation, revision and many corrections are included.

The purpose of the inferential programming's viewpoint is not to coerce programmers, but that they be provided with its conceptual and mechanical tools to facilitate the expression of the program and its justification and to help in the process of program development.

The meaning is not at all to automate the programming process, but to build interactive tools in an inferential programming environment that integrates deductive, inductive and even analogical facilities, which permit the programmer to explore and create.

The use of powerful modularization techniques for both programs and program derivations will permit them to scale up and be applied to very large software.

These techniques would allow modules of a large system to be derived independently and combined together in such a way that all possible interactions could be anticipated, and should make version control to become a more precise activity, by which changes in a module would

need to be propagated only to those modules that were truly affected.

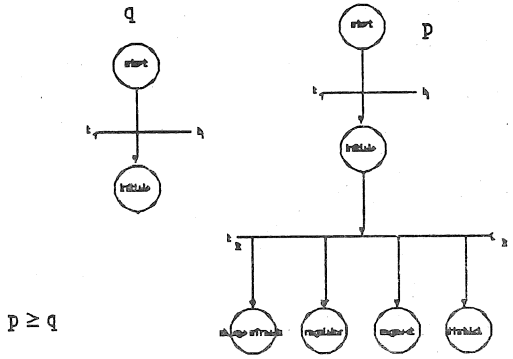
Convergence in a partial ordering

Definition A Petri net p is a refinement of a Petri net q **iff** p is the result of replacing the empty place of q by a proper Petri net.

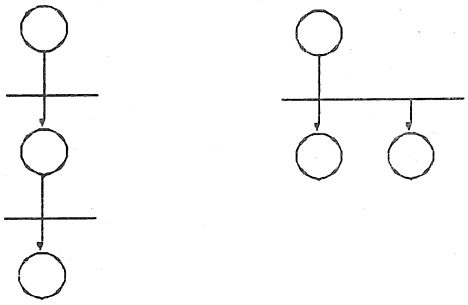
Definition $p \geq q$, where p and q are in a subset of directed Petri nets **iff** q is (equivalent to) a part of p.

We use the notation of Halmos **iff** to mean **if and only if**

Examples



incomparable Petri nets



The properties of the partial ordering$^{\text{Petri}}$

1. $\leq \supseteq$ id reflexive; i.e.: $x \leq x$ for each Petri net x
2. $\leq \supseteq \leq^2$ transitive; i.e.: $x \leq y$, $y \leq z$ imply $x \leq z$ for Petri nets x, y, z

The partial ordering restricted to stepwise refined sequences of Petri nets $x \geq y$ iff x is a refinement of y

Theorem A stepwise refined sequence of Petri nets is a chain, that is for every pair p,q in the

sequence, $p \geq q$ or $q \geq p$.

The partial ordering's topology.

There is a natural way of introducing an algebraic topologization through ordering relations[Petri]. In any partially ordered system least upper bounds(l.u.b.), greatest lower bounds(gr.l.b.), limits superior and inferior, and limits can be defined successively, without assuming the existence of none of them. This can be considered as a definition of limits, thus of closure. Besides this, a property closed related to completeness can be given: any partially ordered system can be completed so as to ensure the existence of at least: least upper bounds and greatest lower bounds in a certain determinate way. This is based on the specialization of Dedekind cuts for numbers.

To any partially ordered set corresponds a unique complete lattice formed by adjoining as ideal elements all Dedekind sections[Birkh].

Definition Directed sets are functions of a poset .

In this complete lattice, an algebraic topology can be introduced by defining for directed sets x_α, $\lim x_\alpha = x$ to mean:

$$\text{l.u.b}_\alpha \{ \text{ gr.l.b.}_{\beta>\alpha}\, x \cup x_\beta \ \} = x$$

$$\text{gr.l.b.}_\alpha \{ \text{ l.u.b.}_{\beta>\alpha}\, x \cap x_\beta \ \} = x$$

An algebraic topology can then be introduced by letting $\lim x_\alpha = x$ in the original partially ordered set mean that $\lim x_\alpha = x$ in the extended set

Then, the topology of the original poset is a relativization of the topology of the extended set.

The idea of using stewise refined sequences of Petri nets has been also presented by B. Krämer[Kräm] in a more formal way. The convergence properties of these sequences could then be used for these sequences to the derivation of computer programs.

Appendix

The derivation of a regulator modeled with a stepwise refined sequence of Petri nets

In this paper we use stepwise refined sequences of Petri nets as a valuable way of describing program derivation.

We show then the concrete example of the derivation of a parallel regulator, whose final algorithm is described in the programming language ADA, which offers facilities for the treatment of parallelism.

The extensive motivation for the need of program derivation has been sufficiently developed, but it hasn't't originated the appearance of tools which are enough formal and that simultaneously permit the programmers experiment the freedom and the joy of creating their works.

This idea is illustrated by presenting the derivation of a parallel regulator whose final version is expressed as an algorithm in the programming language ADA because its possibilities of describing parallel processes.

The simple regulator is a theoretical machine composed by a power source and controls which permit choosing two basic states: automatic and manual.

In the manual state the user can increase or decrease directly the power.

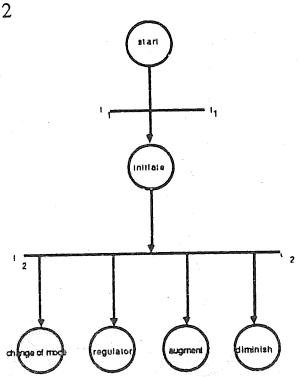
In the automatic state the regulator must obey the equation: $du/dt = k(r - y)$ (integrating regulator).

The variable r is called the reference and can be changed manually. The variable k can be read and changed, the same as the sample interval.

We represent the regulator by means of marked P/T Petri nets.

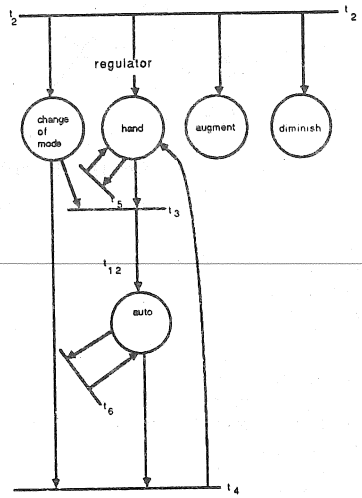
In fig. 1 the transition t_1 is enabled in the beginning of the execution of the regulator by placing a token in the place called *start*. This transitions fires by moving the token to the place called *initiate*.
In this marking, the transition t_2 enabled. We fire then the transition t_2 by removing the token from the place *initiate* and placing tokens in all its output places.

Fig. 2



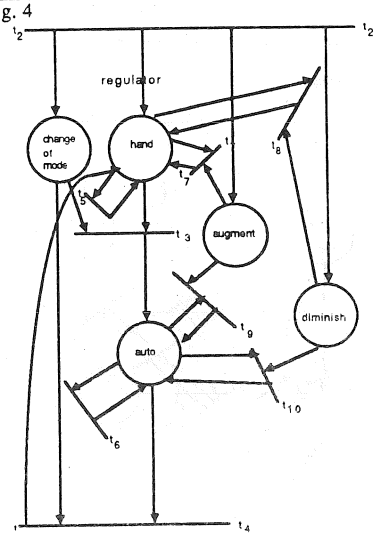
The tasks *change of mode*, *augment* and *diminish* are simple and they influence only the regulator task.

fig. 3



The regulator is composed by two parts with transitions t_3 and t_4 between HAND and AUTO. The transitions t_5 and t_6 are the final of the HAND-LOOP and the AUTO-LOOP in the program. The transition t_4 represents the REG-LOOP.

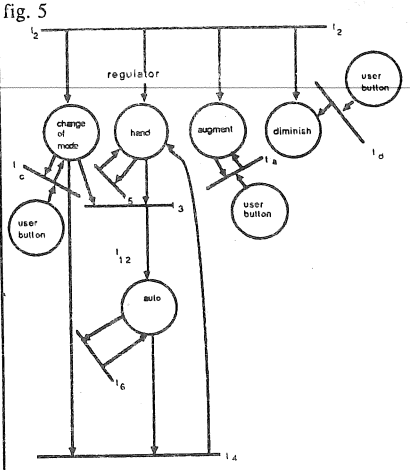
fig. 4



The transitions t_7,t_8,t_9 and t_{10} represent the rendezvous between the regulator and the tasks AUGMENT and DIMINISH.

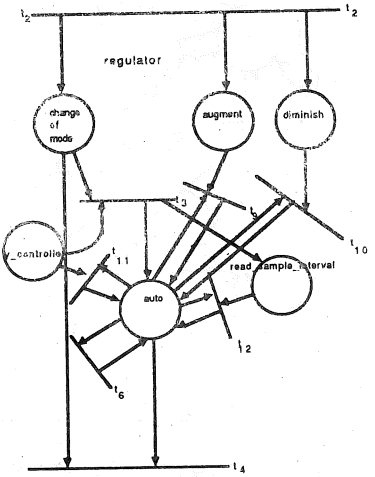
The task CHANGE OF MODE determines the transitions HAND-AUTO and AUTO-HAND(t_3 and t_4).

fig. 5

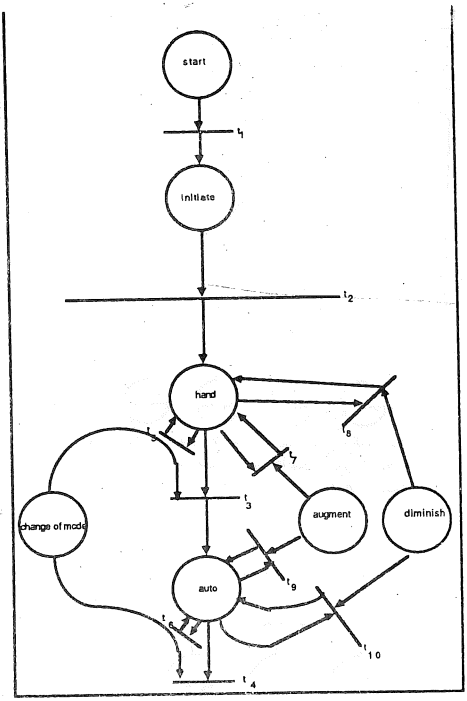


The tasks augment and diminish have actually an interface with the user that who should place a token in his place to allow the firing of the corresponding task.

fig 6 The task AUTO



The transitions t_{11} and t_{12} model the rendezvous between the AUTO-part of the REGULATOR-task and the tasks Y_CONTROLLER and READ_SAMPLE_INTERVAL.
fig. 7 The parallel algorithm of the regulator as a Petri net



The ADA Algorithm for the Regulator

```
   with text_io;use text_io;
   with calendar;use calendar;
procedure main is
-- declarations
package real_io is new float_io(float);
with real_io;use real_io;
type mode_type is (hand,auto);
--- declarations: du_dt ,x,y,l,r,k,change_unity ----
    hand_mode : boolean:=true;
    mode      : mode_type:= auto;
    signal    : character;
    sample_interval : duration:=0.01;
    next_time       : time:=clock;
--- declarations: hand_auto_button, ---
--- declarations: console, sensor, ---
--- declarations: increase_button,
--- declarations: decrease_button,out_device
 procedure initiate is
  -- opens the files, initiates some variables.
 begin
--  open files, etc.  ----
 end initiate;
 begin -- main
  initiate;
  main_block:
  declare
  task regulator is
```

```
 entry change_to_auto(mode :in mode_type);
 entry change_outsignal_hand(l : in float);
 entry change_to_hand(mode :in mode_type);
 entry change_reference(l  : in float);
 entry change_y(y : in float);
 entry change_sample_interval(
       sample_interval : in float);
end regulator;
task change_of_mode;
task augment;
task diminish;
task body regulator is
 begin
  reg_loop:
  loop
      hand_loop:
      while mode=hand loop
       du_dt:=du_dt+l*change_unity;
        select
         accept change_to_auto(mode :
                          in mode_type);
         or
         accept change_outsignal_hand(l :
                                 in float);
         else null;
        end select;
      end loop hand_loop;
      auto_loop:
      while mode=auto loop
        du_dt:=k*(r-y);
        select
         accept change_to_hand(mode :
                          in mode_type);
         or
         accept change_y(y : in float);
         or
         accept change_reference(l : in float) do
           r:=r+l*change_unity;
         end change_reference;
         or
         accept change_sample_interval
               (sample_interval : in duration);
         else null;
        end select;
       auto_block:
       declare -- auto_block
        task y_controller ;
       -- reads the actual state of the process
        -- from sensors and transmit it
        --- to the main task
        task read_sample_interval;
        -- reads the sample interval
        --- and transmit it to the
        -- regulator task
        task body y_controller is
        begin
         loop
          x:=y;
          get(sensor,y);
          if x/=y
           then regulator.change_y(y);
          end if;
          delay next_time-clock;
          next_time:=next_time+
                      sample_interval;
         end loop;
        end y_controller;
        task body read_sample_interval is
        begin
         loop
          select
           accept fin do exit;
           else   get(console,sample_interval);
           regulator.change_sample_interval
                          (sample_interval);
          end select;
         end loop;
        end read_sample_interval;
       begin          --------- block ----------
         null;
       end auto_block; --------- block --------
      end loop auto_loop;
  end loop reg_loop;
end regulator;
task body change_of_mode is
-- receives the signal hand/auto
-- and transmits it
-- to the regulator task
begin
 loop
  hand_mode:=not hand_mode;
  if hand_mode
    then
      mode:=hand;
     regulator.change_to_hand(mode);
     read_sample_interval.fin;
    else
      mode:=auto;
     regulator.change_to_auto(mode);
  end if;
 end loop;
end change_of_mode;
task body augment is
begin
 loop
    case mode is
     when hand=>
        regulator.change_outsignal_hand(1.0);
     when auto=>
        regulator.change_reference(1.0);
    end case;
 end loop;
end augment;
task body diminish is
begin
 loop
    case mode is
     when hand=>
        regulator.change_outsignal_hand(-1.0);
     when auto=>
        regulator.change_reference(-1.0);
    end case;
 end loop;
end diminish;
 begin -- main_block
  null;
```

```
end main_block;
null;
```

```
end main;
```

References

Fer R. Ferré 1987 *A Parallel Version of a Regulator with a Petri Net Derivation.* NordDATA 87, Proceedings.

Sch&S W. L. Scherlis, D. S. Scott 1983 *First steps towards inferential programming* I.P. 83 , IFIP 83

Leh M. Lehman 1980 *Programs, Life Cycles and laws of software evolution* Proc. IEEE 68

Wir N. Wirth 1971 *Program development by stepwise refinement..* CACM 14 4

Petri C. A. Petri, E. Smith 1986*Concurrency and Continuity.* Advances in Petri Nets 1987, Springer Verlag, Heidelberg, 1987.

Birkh G. Birkhoff 1937 *Moore-Smith Convergence in General Topology.* Annals of Mathematics, Vol. 36, No. 1, Jan. 1937.

Kräm B. Krämer 1984 *Stepwise Construction of Non-Sequential Software Systems Using a Net-Based Specification Language.* Advances in Petri Nets 1984. Springer Verlag, Heidelberg, 1984.

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

TABA: UMA ESTAÇÃO DE TRABALHO PARA O ENGENHEIRO DE SOFTWARE

Ana Regina Cavalcanti da Rocha
Jano Moreira de Souza

COPPE/UFRJ
Caixa Postal 68511
CEP 21945 - Rio de Janeiro - Brasil

RESUMO

Características particulares de áreas de aplicação determinam as características desejáveis para seu ambiente de desenvolvimento. A definição do ambiente, por sua vez, vai depender do estado da arte em métodos e ferramentas e do conhecimento de especialistas no uso dos mesmos.

Este artigo descreve o estado atual de um projeto de pesquisa (Projeto TABA) cujo objetivo é a construção de uma estação de trabalho para o desenvolvimento de software, tendo em conta as características de um produto a ser desenvolvido. Este projeto está relacionado com o Projeto ETHOS.

1 INTRODUÇÃO

A qualidade de produtos de software, bem como a produtividade do seu processo de desenvolvimento, depende da qualidade e adequação do ambiente onde é desenvolvido.

Poderia se pensar que uma solução na busca de produtividade-qualidade estaria no desenvolvimento de um único ambiente completo e tentar utilizá-lo para o desenvolvimento de quaisquer projetos. Entretanto, aplicações diferentes tem perfis diferentes e estas diferenças têm influência sobre seus processos de desenvolvimento.

Podemos afirmar que as características de uma determinada aplicação devem determinar as características de seu ambiente de desenvolvimento. A definição deste ambiente, por sua vez, vai depender do estado da arte em métodos e ferramentas e do conhecimento de especialistas no uso dos mesmos.

Diante disto, o ideal será definir um ambiente de desenvolvimento de software adequado para cada área de aplicação, ou mesmo para cada projeto, de acordo com suas particularidades e com experiências anteriores [De Marco 84].

Esta, entretanto, não é uma tarefa trivial pois consome muito esforço e exige alto grau de conhecimento tanto no que se refere a engenharia de software quanto à área de aplicação.

Este artigo descreve o estado atual de um projeto de pesquisa cujo objetivo é a construção de uma estação de trabalho para desenvolvimento de software em diferentes áreas de aplicação. Este projeto está relacionado com o Projeto ETHOS (Estação de Trabalho Heuristica Orientada a Software).

O projeto ETHOS faz parte do acordo de cooperação internacional Brasil-Argentina. O projeto TABA, descrito neste artigo, é uma instância do projeto ETHOS em desenvolvimento na COPPE/UFRJ em colaboração com grupos de pesquisa de outras Universidades.

O nome TABA - em língua tupi, aldeia indígena - foi escolhido por analogia. A taba indígena agrega várias ocas e, nelas se desenrola toda a vida da comunidade. Analogamente a estação TABA agrega várias ferramentas que suportam todo o ciclo de vida do software.

Ao longo do tempo uma mesma estação TABA pode abrigar

diferentes ambientes (aldeias) construídos para projetos diversos.

2 TERMINOLOGIA E CONCEITOS BÁSICOS

Antes de descrevermos o projeto TABA é necessário definirmos alguns termos e conceitos básicos.

O conceito fundamental, neste projeto, é o de **Ambiente de Desenvolvimento de Software**. Consideramos que um Ambiente de Desenvolvimento de Software é composto de:

- um **ciclo de vida**, que define as fases do processo de desenvolvimento e as atividades a serem realizadas em cada fase;
- **métodos**, usados para organizar o pensamento e o trabalho do usuário ao longo do processo de desenvolvimento;
- **ferramentas**, automatizadas ou não, que tornem possível a utilização dos métodos.

Uma definição completa destes termos pode ser encontrada em [Rocha 87].

3 DESCRIÇÃO DA ARQUITETURA GERAL DA ESTAÇÃO

O projeto completo tem como objetivo construir uma estação de trabalho que poderá ser usada com quatro objetivos [COPPE 87]:

a) **auxiliar** o engenheiro de software **na configuração** (especificação) **do ambiente** mais adequado ao desenvolvimento de um produto específico;

b) **auxiliar** o engenheiro de software **a implementar** as **ferramentas** necessárias ao ambiente definido em a);

c) **permitir** aos desenvolvedores do produto (software) **usar a estação através do ambiente especificado em a) e produzido em b)**;

d) **executar o software**, dado que, eventualmente, o software-produto poderá ser executado na própria estação para ele configurada (isso é sempre verdade, pelo menos, na fase de testes).

A seguir detalhamos cada um dos ambientes descritos acima (vide figura 1):

a) **AMBIENTE CONFIGURADOR DE AMBIENTES (META-AMBIENTE)**

A primeira função da Estação de Trabalho é especificar o ambiente de desenvolvimento adequado para uma determinada aplicação. Esta função é realizada pelo ambiente configurador de ambientes (meta-ambiente).

O meta-ambiente auxilia o engenheiro de software na tomada de decisões sobre as características desejáveis para um ambiente de desenvolvimento tendo em conta as características de um produto a ser desenvolvido.

O objetivo do meta-ambiente é fornecer um acesso conveniente aos dados e conhecimentos e, ajudar os engenheiros de software na definição de ambientes apropriados a aplicações específicas. Como tal o meta-ambiente tem aspectos que o assemelham a um sistema especialista de apoio à decisão [Blanning 85].

A base de conhecimentos do meta-ambiente é alimentada tanto pelo ambiente gerador de ferramentas, quanto por informações sobre ferramentas e ambientes adquiridos externamente. Ao longo do desenvolvimento de um produto na estação TABA, a base de conhecimentos do meta-ambiente sofrerá realimentação que irá influenciar na configuração de novos ambientes.

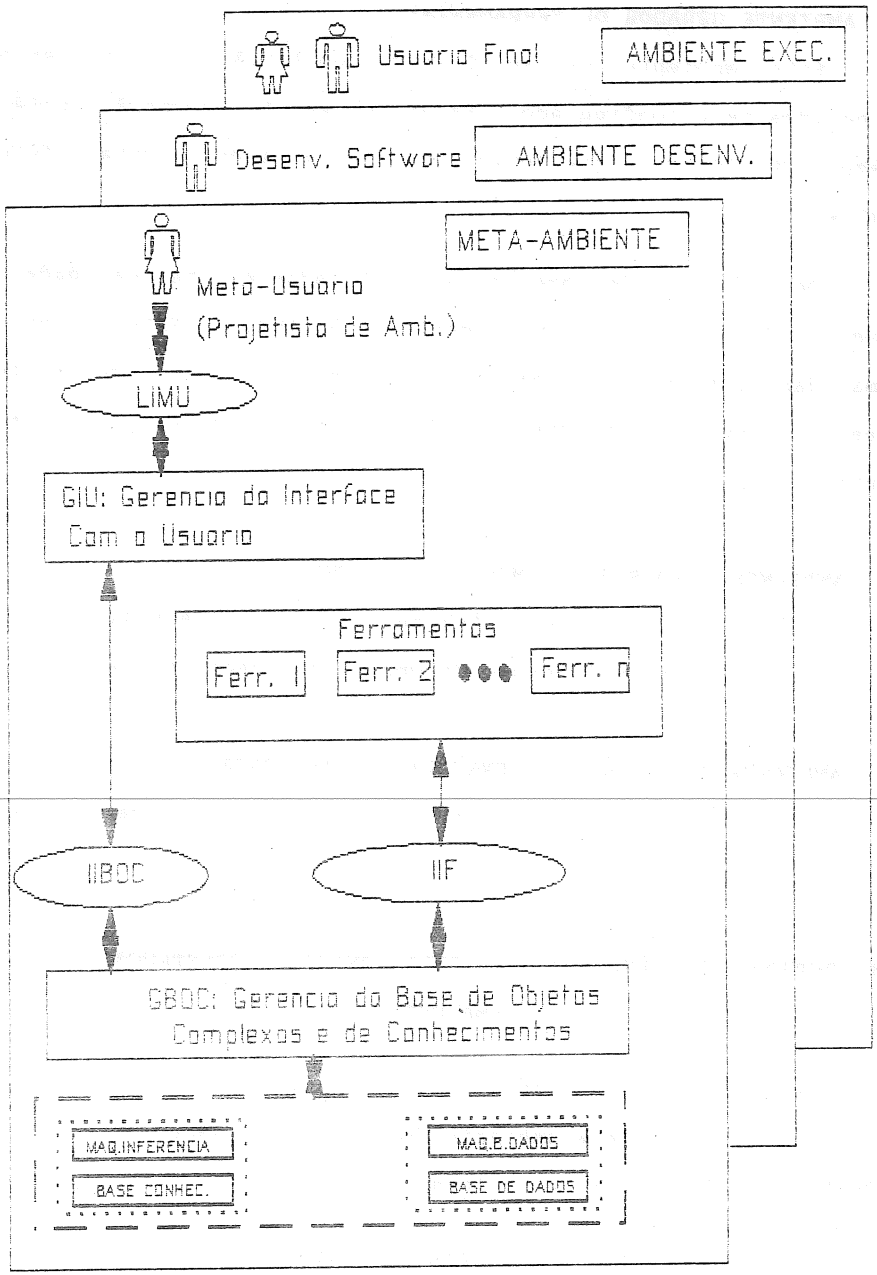


Figura 1 - Ambiente TABA

b) **AMBIENTE GERADOR DE FERRAMENTAS**

Este ambiente vai auxiliar na construção das ferramentas sugeridas pelo configurador. Estas ferramentas passarão, então, a fazer parte do repertório do configurador, para futuros ambientes.

Desenvolver ferramentas automatizadas é, na realidade, um caso particular de desenvolvimento de software. Este ambiente é uma das instâncias do ambiente de desenvolvimento de software descrito em c) com espeficidades por sua interação com o meta-ambiente.

c) **AMBIENTE DE DESENVOLVIMENTO DE SOFTWARE**

A partir de a) e b) tem-se o ambiente especifico a ser utilizado no desenvolvimento de uma determinada aplicação.

d) **AMBIENTE DE TESTE E/OU EXECUÇÃO DA APLICAÇÃO**

Este ambiente permite executar o software desenvolvido em c), na própria estação.

4 ESTUDOS E EXPERIMENTOS RELACIONADOS A CONSTRUÇÃO DO META-AMBIENTE TABA

Nesta seção detalhamos objetivos e sub-projetos relacionados à construção do **meta-ambiente TABA**. Outros aspectos do ambiente TABA serão descritos nas próximas seções.

Meta-ambiente é um ambiente que abriga um conjunto de programas que interage com um usuário especifico (meta-usuário) para definir interfaces, selecionar ferramentas e definir tipos de objetos que comporão um ambiente de desenvolvimento especifico. O meta-ambiente tem ferramentas de especificação,

como por exemplo interfaces gráficas, textuais e menus, ferramentas de definição de objetos da base de software, linguagens de definição de ativação de ferramentas especificas e bancos de conhecimento para apoio na seleção de métodos e ferramentas.

Como já observamos, diferentes aplicações tem caracteristicas diferentes e estas incidem sobre as características desejáveis para seus ambientes de desenvolvimento. Desta forma é desejável que, a partir das características de um determinado produto que se pretende desenvolver, seja possível determinar o ambiente adequado para seu desenvolvimento.

O problema torna-se, então, especificar e construir ambientes adequados ao desenvolvimento de produtos em diferentes áreas de aplicação. Esta, entretanto, não é uma tarefa trivial. A escolha de um ambiente é um processo heurístico e não algorítmico, ou seja, os aspectos envolvidos na escolha não são definidos explicita e claramente para todos os casos, necessitando do conhecimento de especialistas [Blum 86]. Estes especialistas devem ter **conhecimento da área de aplicação** e **experiência no uso de métodos e ferramentas**, o que muitas vezes não pode ser encontrado em uma única pessoa.

Estas questões nos levaram a pensar em um sistema de apoio à decisão suportado por um (meta-)ambiente que forneça apoio ao engenheiro de software encarregado de especificar e construir ambientes.

O primeiro passo na definição do meta-ambiente [Gottgtroy 87] foi identificar as informações que podem auxiliar o engenheiro de software na especificação de um ambiente de desenvolvimento. Foram identificados dois tipos de informação:

1. dados sobre ciclos de vida, ambientes, métodos e ferramentas atualmente disponíveis;
2. conhecimento e experiência no que se refere a:
   - área de aplicação;
   - desenvolvimento de software na área de aplicação;
   - definição de ambientes de desenvolvimento.

O meta-ambiente utiliza o conhecimento de especialistas na identificação e solução de problemas e o torna disponível a outros engenheiros de software auxiliando-os a tomar decisões. Para atingir este objetivo o meta-ambiente não fará apenas recomendações com relação a características do ambiente, mas deverá, também, discutir com seus usuários o porquê das mesmas.

O desenvolvimento do meta-ambiente proposto implica na resolução de algumas questões que, no momento, são objeto de pesquisa da equipe envolvida no projeto:

- aquisição do conhecimento de especialistas;
- representação do conhecimento e mecanismos de inferência;
- processo de desenvolvimento de sistemas especialistas;
- ciclos de vida de software;
- taxonomia de áreas de aplicação, e,
- taxonomia de métodos e ferramentas atualmente disponíveis.

Este trabalho está se realizando através dos seguintes sub-projetos:

- Taxonomia de Métodos e Ferramentas;
- Taxonomia de Domínios de Aplicação;
- Estudo e Classificação de Modelos de ciclo de Vida;
- Aquisição e Representação do Conhecimento;
- Raciocínio Impreciso para escolha de Ambientes;
- Construção do protótipo do meta-ambiente TABA.

4.1 Taxonomia de métodos e ferramentas

O objetivo deste sub-projeto é realizar um amplo estudo da literatura sobre métodos e ferramentas atualmente disponíveis. Este estudo visa realizar uma classificação dos mesmos. Para isto deverá:

i) realizar um amplo levantamento bibliográfico;

ii) elaborar um catálogo de métodos e ferramentas;

iii) definir características que permitam classificar os métodos e ferramentas;

iv) classificar métodos e ferramentas.

Neste momento tem-se concluído um amplo levantamento bibliográfico onde foram identificados, aproximadamente 100 métodos e/ou ferramentas de apoio ao desenvolvimento de software. O trabalho atual está consistindo na elaboração do catálogo de métodos e ferramentas.

4.2 Taxonomia de domínios de aplicação

O objetivo deste sub-projeto é realizar um estudo da literatura sobre características de diferentes áreas de aplicação. A partir deste estudo será realizada uma pesquisa de campo. O estudo e a pesquisa de campo visam possibilitar uma melhor caracterização dos domínios de aplicação. Para isto será realizado:

i) estudo da literatura;

ii) pesquisa de campo;

iii) levantamento de dados;

iv) caracterização dos domínios de aplicação.

Este sub-projeto teve concluída a fase inicial de estudo da literatura e, atualmente, está sendo elaborado o questionário para pesquisa de campo.

4.3 Estudo e classificação de Modelos de Ciclo de Vida

O objetivo deste sub-projeto é realizar um estudo da literatura sobre os diferentes modelos de ciclo de vida propostos. A partir deste estudo será feita uma classificação dos mesmos tendo em conta sua adequação ao desenvolvimento de software nas diferentes áreas de aplicação. Para isto será realizado:

i) estudo da literatura;

ii) classificação dos modelos de ciclo de vida tendo em conta a caracterização de áreas de aplicação feita no sub-projeto 4.2.

Até este momento foi concluída a fase de estudo da literatura e tem-se uma primeira classificação dos modelos.

4.4 Aquisição e Representação do Conhecimento

Este sub-projeto visa o estudo da aquisição do conhecimento de especialistas em Engenharia de Software sobre a utilização de métodos, ferramentas e sobre a área de aplicação; bem como sua representação e utilização por parte das ferramentas dos quatro tipos de ambiente.

Como parte do estudo a ser realizado neste sub-projeto estão formas de integração de bases de dados com bases de conhecimento.

4.5 Raciocínio impreciso para escolha de ambientes

Neste sub-projeto procura-se escolher, entre os métodos disponíveis para lidar com raciocínio impreciso, um método que se

adapte ao problema de escolha de ambiente de desenvolvimento. Este trabalho está baseado na tese de mestrado de Monat [Monat 88].

4.6 Construção do protótipo do meta-ambiente TABA

Os estudos descritos nos sub-projetos 4.1 a 4.5 tem como objetivo culminar na construção de um protótipo do meta-ambiente TABA. Este protótipo utilizará os conhecimentos adquiridos nos sub-projetos anteriores de forma a se ter um sistema de apoio à decisão para configuração de ambientes de desenvolvimento de software.

O estudo realizado em 4.2 (Taxonomia de áreas de aplicação) permitirá selecionar as áreas para consideração inicial, no meta-ambiente TABA. Destas farão parte, em princípio, software comercial, por sua grande difusão e conhecimento já existente, e software científico, pela experiência do grupo envolvido no projeto. Deverá, também, ser considerado software com modo de processamento em tempo-real por suas características peculiares e pelas exigências que apresenta em termos de métodos e ferramentas para desenvolvimento.

A construção do protótipo será realizada através de uma tese de doutorado, atualmente em andamento.

Uma descrição mais completa destes sub-projetos pode ser encontrada em [Rocha 88].

5 ESTUDOS E EXPERIMENTOS RELACIONADOS A SISTEMAS DE GERENCIA DE OBJETOS/DADOS/CONHECIMENTO

Subjacente a todo o software da estação TABA, deverá existir

uma camada responsável pelo armazenamento de todos os dados persistentes da estação. É também uma decisão de projeto que toda a comunicação entre ferramentas e entre estas e o usuário fique também sob a resposabilidade desta mesma camada gerenciadora dos dados/conhecimento. As funções específicas desta camada dependerão do modelo de dados adotado. Nas sub-seções seguintes descreveremos três sub-projetos envolvendo experimentação, que servirão para subsidiar decisões no tocante à arquitetura desta camada no protótipo definitivo. Os sub-projetos são:

i) Construção de um ambiente integrado específico para análise e projeto estruturado;

ii) Desenvolvimento de um protótipo de um sistema para gerência de objetos;

iii) Experimentação em integração de base de dados com base de conhecimentos.

5.1 Construção de um Ambiente Integrado específico para Análise e Projeto Estruturado

Este sub-projeto (já em andamento) visa a implementação de um ambiente para análise e projeto estruturado com as seguintes ferramentas:

a) editor de diagrama de fluxo de dados;

b) editor de diagrama de entidades e relacionamentos;

c) dicionário de dados;

d) editor de gráfico de estrutura.

A ênfase não é a implementação pura e simples das ferramentas (todas elas já implementadas de forma isolada na

COPPE, mas sim em sua integração. Esta integração se dará tanto "acima" na interface com o usuário, onde se está utilizando o MS-WINDOWS em ambiente MS-DOS e se experimentará com o EX-WINDOWS em ambiente UNIX, como "abaixo", via integração no armazenamento de todos os dados do projeto manuseados pelas ferramentas através de um SGBD relacional (o COPPEREL) extendido.

5.2 Desenvolvimento de um protótipo de um Sistema para Gerência de Objetos

O paradigma de orientação para objetos tem parecido promissor à muitos pesquisadores envolvidos em atividades do projeto ETHOS. Entre outras características atraentes do modelo está a aparente possibilidade de tratamento uniforme não só de programas e dados, como destes últimos e do conhecimento.

Há no momento pouca ou talvez nenhuma experiência em implementação de S.O.O. no Brasil, razão pela qual se pretende experimentar em dois níveis:

i) Dado que se possui disponível um S.O.O. tal como SMALLTALK, mesmo que não seja o ideal, como seria a implementação de um ambiente de desenvolvimento (de sistemas não orientados para objetos) sobre o SMALLTALK? Quais seriam as vantagens e desvantagens para a implementação de ferramentas, interface com o usuário, representação de dados e conhecimento? Que método deve ser empregado na implementação de um ambiente como esse? Pretende-se a implementação de um pequeno ambiente com algumas ferramentas sobre o SMALLTALK V para tentar responder a estas perguntas.

ii) Como implementar um S.O.O.? Estudo de diversas alternativas. Estudo de custo/desempenho, possibilidade

de implementações em máquinas paralelas, etc... Alguns experimentos nessa direção já estão em andamento. Duas pequenas implementações de sistemas "relacionais" orientados a objetos estão sendo especificadas. Uma delas utilizando o SGBD SIBANDABARMUS [Souza 81] e a outra o SGBD COPPEREL [Souza 87]. Um estudo sobre primitivas para suporte a bancos de dados não convencionais está sendo realizado e um possível resultado seria que essas primitivas sejam suficientes para a implementação de um SBDOO. Outro estudo já em andamento refere-se á especificação e garantia de restrições de integridade em modelos orientados a objetos.

5.3 Experimentação em integração de Base de Dados com Base de Conhecimentos

Neste sub-projeto, levado a cabo em paralelo com os sub-projetos 4.4 e 4.5 pretende-se estudar maneiras de integração de Bases de Dados com diversas formas de representação do conhecimento (ex. regras, frames, etc...). Um primeiro passo, já em andamento, é a integração de um SGBD relacional com um interpretador Prolog. Os quatro enfoques já clássicos para esta integração estão sendo estudados, incluindo a extensão de uma linguagem para manipulação de B.D., para suportar unificação e "backtracking".

Uma questão que deverá ser respondida por este sub-projeto, juntamente com o descrito em 5.2 é sobre o modelo de dados (e conhecimento) básico da estação.

6. ESTUDOS E EXPERIMENTOS RELACIONADOS A INTERFACE COM O USUARIO

Nesta seção detalhamos um sub-projeto relacionado à interface com o usuário na estação TABA.

Este sub-projeto será levado a cabo num trabalho conjunto de duas linhas de pesquisa Informática e Sociedade e Computação Gráfica.

6.1 Fatores Humanos na Interface Homem-Máquina

Este sub-projeto tem como objetivo estudar a questão dos fatores humanos na interface homem-máquina no caso da estação de trabalho TABA

O trabalho será realizado de acordo com as seguintes fases:

i) estudo da literatura;

ii) primeira identificação de atributos de qualidade desejáveis para a interface homem-máquina numa estação de trabalho;

iii) pesquisa de campo com entrevistas a gerentes, analistas e programadores (usuários potenciais da estação TABA) para identificação definitiva dos atributos da interface homem-máquina.

7 CONCLUSÃO

Este artigo descreveu o projeto TABA, em andamento no Programa de Sistemas da COPPE/UFRJ.

O projeto TABA conta com a participação de vários docentes e pesquisadores da COPPE, bem como com um grupo de aproximadamente 20 alunos de doutorado, mestrado e iniciação científica. Conta, ainda, com a colaboração de docentes e alunos de outras instituições do país.

AGRADECIMENTOS

Os autores agradecem a todos os participantes do projeto TABA por suas inúmeras e valiosas sugestões, bem como pelo entusiasmo com que cada um tem conduzido seu trabalho.

REFERENCIAS

[Blanning 85] Blanning,R.W. "Expert Systems for Management: research and applications"; **Journal of Information Science**; 9(1985).

[Blum 86] Blum,B.I. e Sigillito,V.G. "An Expert System for designing information systems", **John Hopkins APL Technical Digest**, vol.7, no.1 (1986)

[COPPE 87] COPPE/UFRJ; UFRGS e outros; **Uma Estação de Trabalho Heurística para o Engenheiro de Software**; Proposta de Participação no Projeto ETHOS - COPPE/UFRJ; CPGCC/UFRGS; USP/SC; UFPE; UFF; IME; Florianópolis, SC; setembro 1987

[De Marco 84] DeMarco,T. "A Meta-Methodology for Systems Development", **International Workshop on Models and Languages for Software Specification and Design**, Orlando, Estados Unidos, março/1984

[Gottgtroy 87] Gottgtroy,M.; Andrade,C.; Monat,A.; Rocha,A.R.C.da; Souza,J.M.; Aguiar,T.C. "Especificação de Ambientes de Desenvolvimento de Software usando técnicas de Inteligência Artificial"; **7o. Congreso de Metodologias en Ingenieria de Sistemas**; Santiago, Chile; agosto 1987

[Monat 88] Monat,A.S. **Métodos de Raciocínio Impreciso para Sistemas Especialistas baseados em Regras**; Tese de Mestrado, COPPE/UFRJ; junho 1988

[Rocha 87] Rocha,A.R.C.da; Aguiar,T.C.; Blaschek,J.R. **Ambientes de Desenvolvimento de Software: definição de termos**; Relatório Técnico; COPPE/SISTEMAS; agosto 1987

[Rocha 88] Rocha,A.R.C.da; Souza,J.M.; **TABA: uma estação de trabalho para o engenheiro de software**; Relatório Técnico, COPPE/UFRJ; maio 1988

[Souza 81] Souza,J.M.de "Uma experiência com implementações de Sistemas de Banco de Dados Didáticos"; **Revista Brasileira de Computação**; vol 1, no.2; 1981

[Souza 87] Souza,J.M., Mattoso,M.L.Q.; Mattoso,R.Q. "O Projeto Engenharia de Dados"; **II Simpósio Brasileiro de Banco de Dados**; Porto Alegre, RGS; 1987

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

EDITORES DIAGRAMATICOS BASEADOS EM FORMALISMOS GRAMATICAIS

Roberto Tom Price **,+ e Eloi Luiz Favero *,+

SUMARIO

Este trabalho apresenta a idéia de construir geradores de editores para linguagens diagramáticas (editores diagramáticos) partindo do aproveitamento das técnicas utilizadas na construção de geradores de editores textuais com conhecimento da sintaxe da linguagem.

É apresentada uma comparação à nível operacional entre os dois modelos de editores (diagramático e textual) destacando-se as diferenças.

Como resultado da comparação é sugerida uma extensão, no mecanismo de especificação da sintaxe da linguagem, a fim de incorporar a idéia de nodos compartilhados na árvore sintática abstrata.

Este trabalho foi desenvolvido com apoio financeiro da CNPq, SID-Informática e FINEP.

---

* Bacharel em Ciências de Computação (UFRGS/RS-1987), Mestrando em Ciências da Computação (UFRGS/RS);
** Professor e Pesquisador (UFRGS/RS), Eng, MSc, DPhil.
+ Curso de Pós-Graduação em Ciências da Computação, UFRGS, Caixa Postal 1501, Porto Alegre, RS.

1. Motivação

O objetivo de um ambiente de desenvolvimento de desenvolvimento de software (ADS) é de proporcionar um ambiente de construção de software uniforme e integrado. Este ambiente deve prover ferramentas para a execução das tarefas associadas a cada etapa do ciclo de desenvolvimento de software.

ADSs são baseados em um conjunto de ferramentas diagramáticas e/ou textuais.

Uma ideia atraente para a criação de ferramentas, que manipulam linguagens, em ADSs é a geração das ferramentas a partir de descrições (formais ou semi-formais) das linguagens.

Para ferramentas textuais, já, existem diversos sistemas que geram editores interativos a partir de descrições das linguagens[Rep 84],[Don 84],[Sne 85], [Pri 84].

Estes editores chamados de editores com conhecimento de sintaxe (ECS), podem incorporar várias facilidades como edição por templates, formatação, verficação semantica por gramáticas de atributos e outros.

Para ferramentas diagramáticas não existe o mesmo avanço técnico, i.e. mecanismos geradores de editores diagramáticos a partir das especificações das linguagens.

Em princípio é possível encontrar uma representação textual para qualquer linguagem diagramática. Para a representação textual é possível gerar um ECS, com o uso dos sistemas mencionados acima.

Quais são as modificações/extensões destes sistemas (geradores de EDSs) para gerar um editor diagramático ?

Este trabalho é um passo inicial na ideia de criar geradores de editores diagramáticos, partindo do aproveitamento das tecnicas utilizadas na geracao de editores textuais.

2. Estrutura do trabalho

No item 3 são sumarizados os principais conceitos relacionados com geradores de editores com conhecimento da sintaxe da linguagem (ECS). Os conceitos servem de base para a comparação entre os modelos de editores: textual e diagramático.

No item 4 a forma operacional do editor ECS e do editor diagramático (ED) é comparada; a discussão e centrada em exemplos de editores (textual e diagramático) para linguagem de diagramas de fluxo de dados (DFD). No item 5 são sumarizadas as diferenças.

No item 6 é apresentada uma sugestão de extensão na meta linguagem de especificação, a fim de permitir nodos compartilhados.

Por fim, no item 7 é redefinida a gramática livre de contexto (GLC), que especifica o DFD textual, a fim de tornar a representação interna do DFD mais apropriada para a linguagem diagramática do DFD.

Todas as figuras referenciadas no texto, exceto a figura 7, estão no final do artigo.

3. Geradores de EDSs uma visão geral

Neste item são apresentados rapidamentes os principais conceitos relacionados com geradores de editores de linguagens, servindo com base para a discusão que se segue.

Inicialmente os editores podem ser classificados em editores de propósito geral (sem conhecimento de qualquer aspecto da linguagem) e **editores com conhecimento** especifico de alguns aspectos de uma ou mais linguagens.

Sob o ponto de vista da estrutura de representação interna do texto, em um editor com conhecimento da sintaxe da linguagem (ECS), diferencia-se de um editor de propósitos gerais por possuir a representação interna em forma de árvore sintática abstrata (ASA), guardando as informações da estrutura sintática.

Todas as operações de edição num ECS são executadas em função do **foco de atenção** definido pela posição do cursor; antes de executar uma operação de edição o usuário se posiciona em um ponto do texto e o sistema gera um menu de operações de edição, em função da posição do cursor.

Com os ECS surgem os conceitos de **sintaxe concreta** (representação do documento na forma textual) e **sintaxe abstrata** (representação interna do documento na ASA)[Don 84].

A representação do texto na árvore sintática abstrata é definida a partir da gramática livre de contexto (GLC). Todo nodo da ASA está associado a um não terminal ou terminal da GLC.

O processo de submissão de texto ("parse") consiste numa transformação da sintaxe concreta para a abstrata e o processo de visualização ("unparse") na transformação da abstrata para concreta.

O termo ECS ou EDS é utilizado usualmente para editores que incorporam também conhecimento da semântica estática e/ou de outras facilidades como por exemplo interpretação.

Algumas implementações de ECS são especificas para uma linguagem, como por exemplo "The Cornell Program Syntethizer" [Tei 81]. Muitas implementações são baseadas em tabelas, as quais definem os aspectos do conhecimento da linguagem. Com a substituição das tabelas para diferentes linguagens são gerados diferentes editores especificos. Surge assim o conceito de geradores de ECS.

Os geradores de ECS utilizam metas linguagens para especificação dos diversos aspectos que definem a linguagem [Ali 83][Don 84][Rep 84]. Diversos níveis de conhecimento são especificados na meta-linguagem. Para cada nível exitem notações e mecanismos associados, suscintamente mencionados abaixo:

a) nível sintático

concreta (representação textual para submissão e visualização):

- variante de BNF (Backus Naur Form/Backus Normal Form)

abstrata (representação interna):
- phila/operator[Don 84][Rep 84]
- CLASS, LIST, NODE [Sne 85][Bah 86]
- anotações na BNF [Fav 88]

formatação (especificação da apresentação do teste):
- associada a concreta ou abstrata

auxílios a submissão: menus e templates
- gerados automaticamente da abstrata ou
- associados à abstrata

b) semântica estática (verificação de tipos, definições de variáveis, etc)
- ações semânticas anotadas na abstrata
- gramática de atributos associada a abstrata

c) interpretação (semantica da execução)
- associda a sintaxe abstrata (caminhamento na ASA)
- compilação e execução de fragmentos (gramática de atributos + ações semânticas)

d) depuração (interpretação controlada)
- associada a interpretação

3.1 Forma geral de operação

O algoritmo de um EDS consiste num ciclo de transformações de sintaxe concreta para abstrata ("parse") e de abstrata para concreta ("unparse").

3.2 O processo de submissão do texto ("parse")

É responsável pela edição do documento (criação/alteração dos nodos da ASA). Toda operação se faz em função da posição do cursor.

Na implementação de ECS, quanto a forma de submissão de texto ("parse") existem duas abordagens: a edição direta do texto (**reconhecimento incremental** [Sch 84] [Ghe 79]) e a **edição auxiliada por menus e "templates"** [Tei 81][Med 82].

O reconhecedor incremental é gerado automaticamente a partir a associação entre a sintaxe concreta e abstrata.

Os menus em um EDS são gerados em função da posição do cursor sobre a ASA e pela classificação dos elementos sintáticos como CLASS (menu), NODE("template"), LIST(repetição).

Os comandos de edição baseados em menus, gerados a partir da sintaxe da linguagem, garantem a consistencia a nível sintático.

O "Syntethizer Generator" utiliza o formalismo de gramática de atributos para gerar um mecanismo que executa a análise semântica incremental. Este mecanismo opera sobre árvore sintática, enriquecida com atributos que são associados a cada nodo da ASA. O valor de um atributo representa uma propriedade

especifica de um nodo da ASA. Toda vez que uma operação de edição altera nodos da ASA as equações semânticas associadas aos nodos com valores de atributos afetados são reavaliadas [Rep 83][Rep 84].

3.3 O processo de visualização ("unparse")

Este processo é responsável pela visualização do texto na tela (em torno da posição corrente do cursor). Para tal é necessário manter uma relação entre os tokens terminais na tela com a sua posição na ASA. Cada elemento terminal da tela (sintaxe concreta) está internamente representado na ASA(sintaxe abstrata); um posicionamento (movimento do cursor) sobre o elemento na tela, implica num posicionamento sobre o elemento na ASA. Este posicionamento pode ser por teclas de função ou com a utilização de "mouse".

O processo de visualização pode executar os comandos de navegação (mudança do foco de atenção), auxilios a visualização (ex. alterar o número de caracteres de endentação, "holophrasting").

4. Editor de DFD textual VS diagramático

A figura 1 mostra um DFD representado diagramaticamente. Na figura 2 está representado na forma textual o mesmo DFD.

Para esta representação textual do DFD é possivel gerar um EDS, utilizando um gerador de EDSs, como o "sinthesizer generator" [Rep 84] ou EDS-UFRGS [Pri 87].

A representação textual é definida pela GLC da figura 3. Esta GLC especifica precisamente quais são as sentenças sintaticamente válidas para o DFD. A verificação da semantica estática pode ser especificada pelo formalismo de gramática de atributos (GA).

As mensagens de erros gerados pelo mecanismo de GA são apresentadas na tela como mostra a figura 4 e, a cada ciclo de execução de um comando as mensagens são atualizadas.

Admitindo que a representação interna do DFD textual, em forma de ASA,é adequada para um editor diagramático, para mostrar que é possivel a criação de editores diagramáticos partindo das técnicas de geração de editores textuais, é suficiente mostrar que é possivel executar as operações relacionadas com a submissão (reconhecimento - "parse") e a visualização ("unparse") dos objetos diagramáticos representados na ASA.

A análise semântica, executada pelo formalismo de GA, pode ser utilizada também para o DFD digramático, pois a representação interna do DFD diagramático é também uma ASA. A notação utilizada para a especificação da análise semântica estática para os ECSs aplica-se de modo direto para os EDs.

As quatro operações de edição, que seguem, exemplificam a forma de interação no ECS vs a forma de interação no ED:

i) criação do depósito "D1" (entidades e processos são criados da mesma forma).

ii) criação do fluxo "a" entre a entidade "E1" e o processo "P2".

iii) remoção do fluxo "a".

iv) remoção da entidade "P1"

As figuras 5a a 5d exemplificam uma possível forma de interação para o editor de texto do DFD. Este editor de texto para DFD foi criado apenas para comparação com o editor diagramático de DFD; a representação diagramática de um DFD é mais expressiva e natural.

i) Posiciona-se o "mouse" sobre o "placeholder" {lista_de_depósitos} e digita-se "D1" (figura 5a-5b).

ii) Posiciona-se o "mouse" sobre o {fluxo} e digita-se a frase "a: E1 --> P2" (figura 5b-5c).

iii) Posiciona-se o "mouse" sobre a frase "a: E1 --> P2" e aciona-se a função "REMOVE" (figura 5c-5b).

iv) Posiciona-se o "mouse" sobre a frase "P1" e aciona-se a função "REMOVE" (figura 5c-5d).

As figuras 6a a 6e exemplificam uma possível forma de interação para editores diagramáticos de DFD

i) Seleciona-se o "placeholder" {depósito} do menu; o elemento é levado para dentro da janela de edição na posição desejada; e, a seguir, digita-se o seu identificador "D1" (figura 6a-6b).

ii) Apontamento de "E1" (6b-6c), seguido do apontamento de "P2" (6c-6d). Para a criação do arco segue-se o processo anterior (i). Por fim o editor deve associar os elementos apontados e o arco com um novo fluxo (6d-6e). Entende-se por apontamento: posiciona-se o mouse sobre o elemento visado e ativa-se a função "APONTAR" (mais adiante no artigo é definida a semântica desta função).

iii) Posiciona-se o mouse sobre o arco, elemento gráfico que identifica o fluxo na tela, e tecla-se "REMOVER" (6e-6b).

iv) Posiciona-se sobre o processo e tecla-se "REMOVER" (figura 6e-6f). No DFD textual a existência de fluxos que referenciam processos não definidos não cria maiores problemas. Porem, para o DFD diagramático não tem sentido a existência de um fluxo (arco) que referencia um processo inexistente. Assim parece normal remover juntamente com o processo todos os fluxos que o referenciam.

5. Comparação entre edição textual e diagramática

As diferenças entre EGs e EDSs surgem em função dos elementos primitivos e, principalmente, pela inexistência, nos EGs, do conceito de ordem na escrita do texto.

5.1 Elementos primitivos

Os elementos primitivos (nível léxico) para o ED são objetos gráficos, enquanto que no EDS são tokens de caracteres.

Para a especificação dos tokens léxico em EDSs é utilizada uma notação do tipo gramática regular.

Para especificar os elementos primitivos gráficos é necessário um meta editor gráfico (MEG) [Mel 88]. O meta editor gráfico permite a especificação de elementos gráficos compostos de elementos geométricos primitivos (círculo, polígnos, etc.), com rótulos (textos) associados.

Para ligação entre o nível sintático e léxico é criado um cabeçalho para todo elemeno gráfico. No cabeçalho são especificados atributos herdados ou sintetizados; um elemento gráfico primitivo pode ser visto como um nodo da GA.

5.2 Ordem de escrita do texto Vs caminhamento em pré-ordem na ASA

Nos EGs a ordem dos tokens na tela não tem relação com a estrutura da ASA; ao contrário no EDSs a ordem de escrita do texto está relacionada com um caminhamento em pre-ordem na ASA. Este conceito de ordem tem duas implicações:

5.2.1 A geração de menus

No EDS do DFD (textual) os "placeholders" são inseridos no meio do texto, na posição definida pela formatação, na ordem de escrita do texto.

Para o ED de DFD, como o conceito de ordem não existe, usualmente os "placeholders" (como primitivas gráficas) são deixados no menu lateral da janela de edição (figura 6). A seleção do "placeholder" do menu no ED corresponde ao posicionamento sobre o "placeholder" no EDS, que por sua vez corresponde a um posicionamento sobre a ASA. Após a escolha o usuário traz o elemento para a janela de edição - isto é define a posição do elemento na página.

5.2.2 O processo de visualização

A formatação para EDSs é gerada automaticamente (caminhamento em pre-ordem) a partir da sua especificação associada a sintaxe abstrata. No ED o processo de visualização

guarda a posição (explicitada pelo usuário no momento da criação) de cada elemento gráfico primitivo.

O processo de visualização exibe os objetos que são referenciados na ASA, como nodos da janela de edição corrente. A janela corrente usualmente está associada a uma página; o conteúdo da página é definido explicitamente pela edição dos elementos gráficos. Nos EDSs o formatador divide automaticamente o texto em páginas, pelo caminhamento na árvore.

No ED o processo de visualização é programado (não é automático) para executar o comando de troca de página. Por exemplo para o ED do DFD a troca de página é necessária para a explosão (refinamento) de um processo; a página contém um nível do DFD, sem os refinamentos.

O comando de troca de página necessita de um posicionamento do mouse sobre um processo, seguido da ativação da função "NOVA_PAGINA" (equivale a digitação de SUB_DFD "P1" no editor textual).

Uma alteração da posição de um elemento gráfico primitivo ou uma troca de página não afeta a estrutura da ASA.

6. Nodos compartilhados Vs apontamento

Para um ED é natural a escolha de elementos (apontamento) que estão na tela para a composição de novos elementos (operação ii). Em linguagens textuais esta forma de referenciar elementos é feita pela escrita de um identificador associado a unidade que é referenciada (por ex. o a explosão do processo "P1" da figura 1 é indicada pela rescrita do identificador "P1" após a palavra reservada SUB_DFD).

Esta forma de composição por apontamento de elementos introduz uma idéia de compartilhamento de nodos na ASA. sugerindo uma extensão da GLC para permitir nodos compartilhados na ASA (fig 7).

```
Fluxo --> Elementol : TipoElemento
          Elemento2 : TipoElemento
          Arco_gr.

TipoElemento --> Atividade | Deposito | Processo.

...
```

figura 7: especificação do fluxo (nodos compartilhados)

A notação "Elementol : TipoElemento" denota que o Elementol é do tipo TipoElemento, porém não é um nodo e sim uma referência para um nodo do TipoElemento. Uma instâcia deste tipo de nodo é criada pela operação de apontamento. A operação de apontamento é executada em duas etapas: a) pela marcação dos nodos na tela e, b) ligação com um nodo sintático. Na figura 6 o menu de "apontamento" mostra quais são os elementos que podem ser apontados.

No caso de utilização do mecanismo de GA, os nodos compartilhados não podem possuir atributos herdados, somente

sintetizados. Uma alteração nos atributos do nodo compartilhado implica numa reavaliação de todos os nodos que o referenciam.

7. A redefinição da GLC para o ED de DFD

Na figura 8 é apresentada uma GLC mais apropriada para o editor diagramático. Esta GLC foi especificada considerando os aspectos particulares de edição nos EGs.

Nesta gramática o refinamento de um processo é associado ao própio processo na ASA. A hierarquia de decomposição de processo é representada de forma mais natural nesta GLC.

Na GLC da figura 3 a consistência de que o refinamento é válido somente para processo é feita no nivel semântico, nesta GLC a a mesma consistência passa para o nivel sintático.

8. Conclusão

O DFD é uma linguagem baseada no conceito de grafo. O conceito de grafo na notação proposta neste trabalho é representado com a utilização de nodos compartilhados.

Uma alternativa a esta notação é a geração de ED a partir de formalismos que possuem (embutido) o conceito de grafo. Não existe ainda um formalismo, baseado no conceito de grafo, que permite (na prática) a geração de ED. Para o DFD a notação de nodos compartilhados parece não apresentar maiores inconvenientes.

Um argumento favorável a esta notação é o conceito de decomposição (refinamento), associado a uma estrutura de árvore hierarquica, presente em quase todas as linguagens (diagramáticas ou textuais). Este conceito é expressado naturamente em árvores (como estruturas sintáticas).

Outro argumento favorável é a utilização do mecanismo de gramática de atributos para executar a análise semântica de forma incremental; possibilitando a especificação dos aspectos de semântida da linguagem diagramática em uma notação declarativa.

Uma vantagem na utilização do mesmo formalismo (ainda que estendido) para a geração de ediores textuais e diagramáticas esta na compatibilidade da representação interna de documentos diagramáticos e textuais, facilitando a construção de de ferramentas integradas para os ambientes de dessenvolvimento de projetos em geral.

Os autores estão desenvolvendo um protótipo de um gerador de editores diagramático (ED), baseado nas idéias expostas neste artigo. O protótipo está sendo construido pela adaptação de um gerador de editores dirigidos por sintaxe com um meta editor gráfico [Pri 84][Mel 88].

9. Referências

[All 83] Allison, Lloyd; "Syntax Directed Program Editing". Software Practice And Experience, vol. 13, 453-465 (1983).

[Bah 86] Bahlke, Rolf e Snelting, Gregor; "Context-sensitive editing with PSG environment"; Lecture Notes in Computer Science: "advanced programming environment" no 244, june, 1986.

[Don 84]
Donzeau-Gouge, U.; Kahn, G.; Lang, B. & Melese, B. "Document Structure and Modularity in Mentor". SIGPLAN NOTICES, Vol 19, no 5,pg. 141-8, May 1984.

[Fav 88]
Favero, E.L. e Price R.T. "A Implementação de editores dirigidos por sintaxe - Algumas considerações" (não publicado) UFRGS, março 1988.

[Ghe 79]
Ghezzi C. e Mandrioli D. "Incremental Parsing", ACM TOPLAS, vol 1, 58-70, 1979.

[Med 82]
Medina-Mora R., "Syntax_directed editing: towards integrated programing environments". Dep. Comp. Sc., Carnegie-Mellon Univ., Pittsburgh, 1982. (Ph.D. dissertation).

[Mel 88]
Melo W.L.M. e Price R.T. "O editor gráfico generalizado". (a ser publicado nos anais do SEMISH, VIII CONGRESSO NACIONAL DA SOCIEDADE BRASILEIRA DE COMPUTAÇAO, Julho de 1988).

[Pri 84]
Price, R.T. "A Prototype syntax driven language editor". Brighton, University of Sussex, 1984. (Tese de Doutorado).

[Pri 87] Price R.T. e Favero E.L. "Introdução a Linguagem de Especificação (LDE)". VII Congresso da Sociedade Brasileira de Computação, 1987.

[Rep 83]
Reps T.; Teitelbaum T. e Demers A. "Incremental Contex-dependent Analysis for Language-based Editors". ACM TOPLAS vol 5, no 3, July 1983.

[Rep 84]
Reps T. e Teitelbaum T., "The synthesizer generation. SIGPLAN NOTICES, vol 19, no 5, 42-8, may 1984.

[Sne 85]
Snelting, G. "Experience with the PSG - Programming System Generator. Lecture Notes In Comp. Sci. 148-162 vol 2: Formal Methods & Softw. Development. Springer-Verlag, 1985.

[Sch 84]
Schwartz M.D.; Delisle N.M. e Begwni V.S. "Incremental compilation in Magpie". SIGPLAN NOTICES vol 19, no 6, June 1984.

[Tei 81]

Teitelbaum T., Reps T. e Horwitz S., "The Why and Wherefore of the Cornell Program Synthesizer", SIGPLAN NOTICES, Vol 16, no 6,pg. 8-16, June 1981.

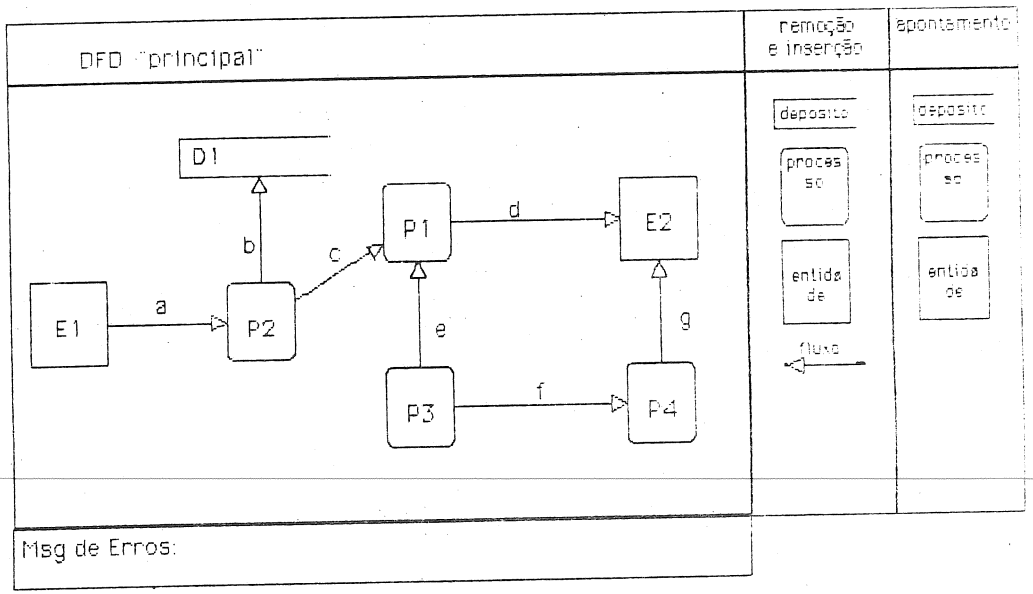


figura 1: representação de um DFD diagramático

```
DFD  principal
   e1, e2 : ENTIDADES;
   d1 : DEPOSITOS;
   p1, p2, p3, p4 : PROCESSOS;
   FLUXOS
         a : e1 ---> p2;
         b : p2 ---> d1;
         c : p2 ---> p1;
         d : p1 ---> e2;
          ...
    END_FLUXOS
    SUB_DFD  p3
          p3.1, p3.2, p3.3 : PROCESSOS;
          d3 : DEPOSITOS;
        FLUXOS
           d3_f1 : p3.2 ---> p3.1;
           d3_f2 : p3.1 ---> p3.3;
            ...
       END_FLUXOS
   END_SUB_DFD p3;
END_DFD principal;
```

figura 2: uma representação textual para o DFD

```
DFD  principal
   e1, e2 : ENTIDADES;
   d1, d1 : DEPOSITOS;  !ERRO1:  nomes de depósitos duplicados !
   p1, p2, p3, p4 : PROCESSOS;
   FLUXOS
         f1 : e1 ---> p2;
         f2 : p3 ---> e1;
         f3 : p3 <--> d2;  !ERRO2:  depósito (d2) nao especificado !
         f4 : d1 ---> e1;  !ERRO3:fluxo sem processo !
          ...
    END_FLUXOS
    SUB_DFD  p3
          p3.1, p3.2, p3.3 : PROCESSOS;
          d3 : DEPOSITOS;
        FLUXOS
           d3_f1 : p3.2 ---> p3.1;
           d3_f2 : p3.1 ---> p3.3;
            ...
                        !ERRO4:  fluxos nao utilizados!
                        !ERRO5:  depósitos sem entras ou  saidas!
       END_FLUXOS
   END_SUB_DFD p3;
END_DFD principal;
```

figura 4: exemplos de mensagens de erros geradas pela mecanismo de gramática de atributos

```
DFD  principal
   e1, e2, {entidade} : ENTIDADES;
   {lista_de_depósitos}
   p1, p2, p3, p4, {processo} : PROCESSOS;
   FLUXOS
         c : p2 ---> p1;
         d : p1 ---> e2;
         ...
         {fluxo}
    END_FLUXOS
    {lista_de_sub_DFDs}
END_DFD principal;
```

figura 5a: um DFD parcialmente editado

```
DFD  principal
   e1, e2, {entidade}: ENTIDADES;
   d1, {deposito} : DEPOSITOS;
   p1, p2, p3, p4, {processo} : PROCESSOS;
   FLUXOS
         c : p2 ---> p1;
         d : p1 ---> e2;
         ...
         {fluxo}
    END_FLUXOS
    {lista_de_sub_DFDs}
END_DFD principal;
```

figura 5b: Após a inclusão do depósito "D1"

```
DFD  principal
   e1, e2, {entidade}: ENTIDADES;
   d1, {deposito} : DEPOSITOS;
   p1, p2, p3, p4, {processo} : PROCESSOS;
   FLUXOS
         c : p2 ---> p1;
         d : p1 ---> e2;
         ...
         a : e1 ---> p2;
         {fluxo}
    END_FLUXOS
    {lista_de_sub_DFDs}

END_DFD principal;
```

figura 5c: Após a inclusão do fluxo "a: e1 ---> p2".

```
DFD  principal
   e1, e2, {entidade}: ENTIDADES;
   d1, {deposito} : DEPOSITOS;
   p2, p3, p4, {processo} : PROCESSOS;
   FLUXOS
         c : p2 ---> p1;
         d : p1 ---> e2;
         ...
         a : e1 ---> p2;
         {fluxo}
    END_FLUXOS
    {lista_de_sub_DFDs}

END_DFD principal;
```

figura 5d: após a remoção do processo "p1"

```
dfd ---> SubProcesso.

SubProcesso  ---> Id_Gr
                  ListaEntidades
                  ListaDepositos
                  ListaProcessos
                  ListaFluxos .

ListaEntidades ---> Entidade_Gr   ListaEntidades.

ListaEntidades ---> Empty.

ListaProcessos ---> Processo ListaProcessos.

ListaProcessos ---> Empty.

Processo ---> Processo_Gr
              [ SubProcesso ]

Fluxo ---> Elemento1:TipoElemento
           Elemento2:TipoElemento
           Arco_GR .

TipoElemento ---> Entidade | Deposito | Processo.
```

figura 8: uma especificação da sintaxe para o DFD diagramático

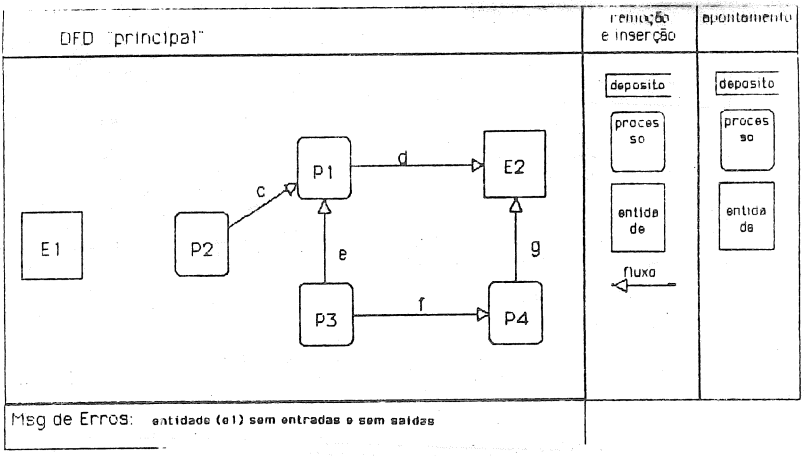


figura 6a: um DFD parcialmente editado

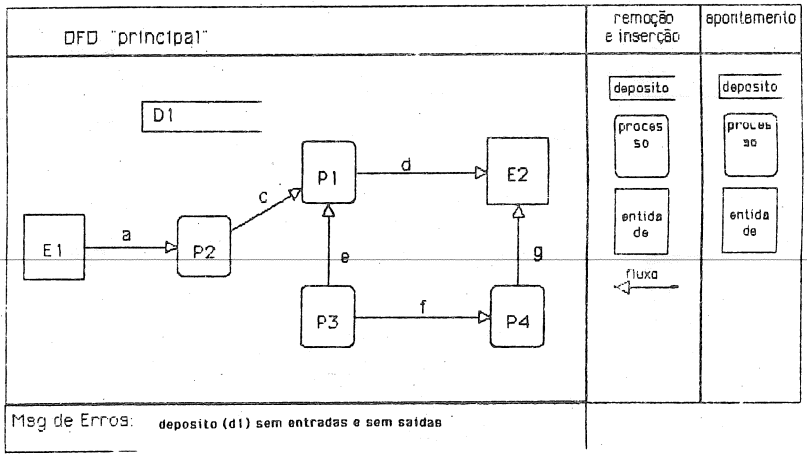


figura 6b: após a criação do depósito "d1"

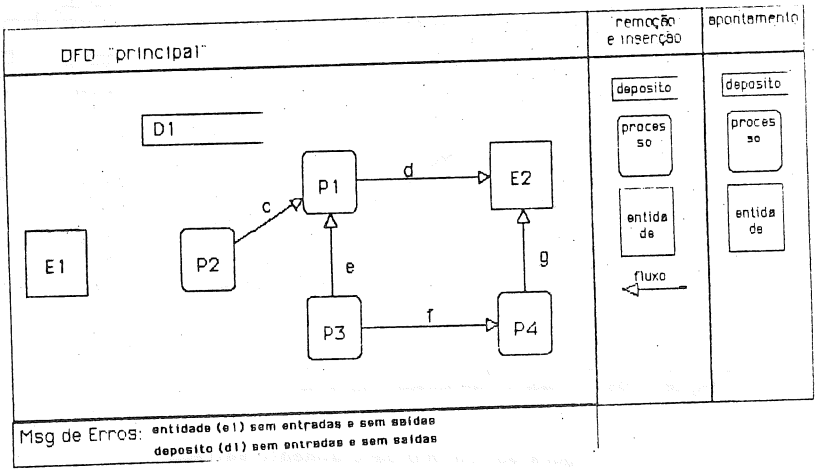


figura 6c: a entidade "E1" está apontada para a criação do fluxo

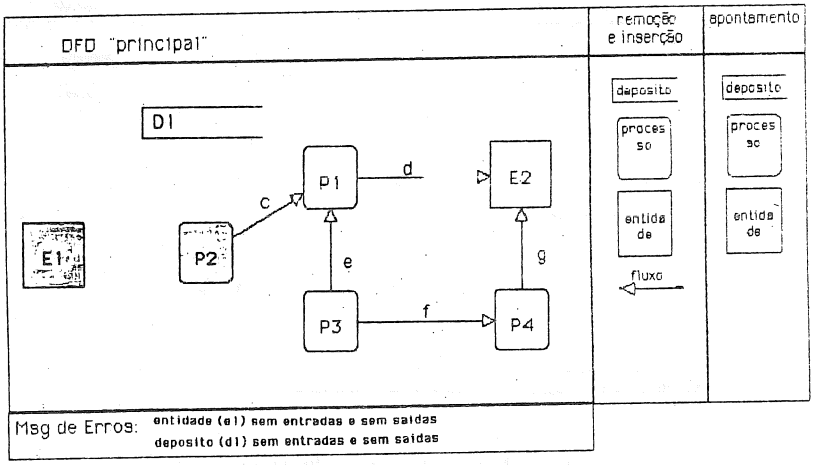


figura 6d: a entidadade "E1" e o processo "P2" estão apontados para a criação do fluxo

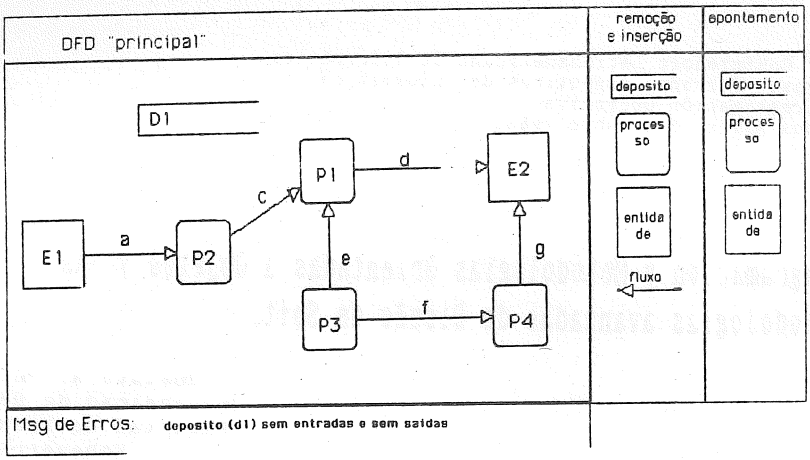


figura 6e: após a criação do fluxo "a"

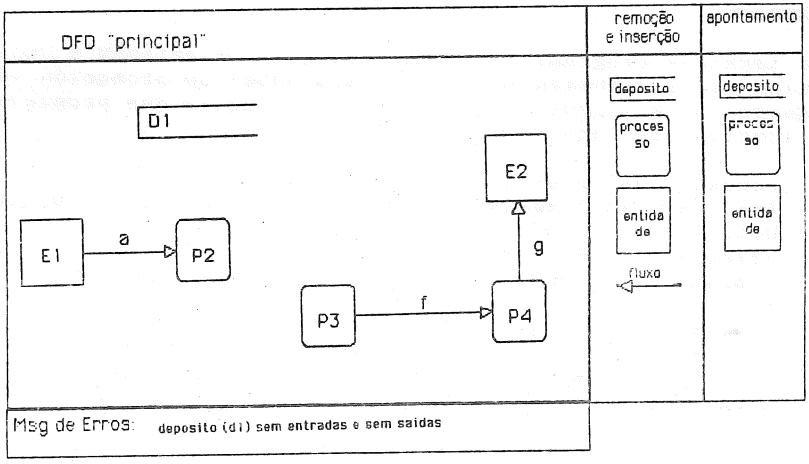


figura 6f: após a remoção do processo "P3"

XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Programación y Metodologías Orientadas a Objetos y las Metodologías avanzadas de Diseño de Soft.

Gustavo A. Pollitzer
Universidad de Belgrano
Facultad de Tecnología
Echeverría 2838
1428 Buenos Aires

Resumen:

Los términos Programación y Metodologías Orientadas a Objetos han tenido mucha difusión en los últimos años. Su promoción viene en algunos casos ligada a productos o a escuelas que promueven su aplicación desplazando otras herramientas.

El propósito del presente trabajo es:
- Analizar las llamadas Metodologías Orientadas a Objetos
- Detectar sus componentes escenciales y conceptuales.
- Rever el proceso de Diseño de Soft y detectar sus necesidades.
- Comparar las MOO con otras metodologías que se aplican a este fin.
- Ampliar las estrategias usadas para diseño incorporando los nuevos conceptos y herramientas, desarrollando así, aún más, el conjunto de Metodologías avanzadas de Diseño de Soft, tomando lo mejor de cada mundo y dejando de lado elementos no fundamentales de una escuela dada.

Metodologías Orientadas a Objetos. Conceptos fundamentales.

Las Metodologías Orientadas a Objetos toman al Objeto como la entidad básica o componente esencial para modelizar el mundo real o el problema a encarar. Ese Objeto tendrá atributos (o ítems de datos) y propiedades y estará sujeto a procesos. Su comportamiento se caracteriza por las acciones que sufre y que requiere de otros objetos. (Booch, 86)

Equivale a una estructura de datos de tipo Record. (Jacky y Kalet, 87).

Un Objeto es una instancia de una Clase a la que pertenece. Esa Clase equivale al Tipo de Datos del Objeto. Asociado con este tipo de datos, habrá un conjunto de operaciones o métodos.

La regla fundamental es que todo atributo de un Objeto sólo es accesible por sus métodos. Se implementa así algo equivalente a una cápsula de Tipo Abstracto de Datos.

De una clase puedo derivar nuevas clases o subclases, como una extensión de la anterior, su superclase. Al declarar que una clase es subclase de otra, la nueva incluye (o hereda) los atributos y los métodos de la superclase.

Podrá además ser ampliada con sus propios atributos y métodos. Si así se desea, los valores de los atributos de la superclase pueden ser redefinidos (overriden) por una nueva declaración de su valor en la subclase o en un objeto dado. En este caso deberíamos tener conciencia de las objeciones conceptuales al respecto.

Los métodos son invocados o activados por mensajes emitidos por otros métodos. El Polimorfismo implica que el programador no necesita ocuparse del tipo de los datos y los objetos, que serán interpretados por los métodos según sean las características de los objetos a que fueron destinados los mensajes.

El análisis del orden de activaciones de los métodos o procedimientos muestra que los mensajes son colocados en una estructura con la política de atención necesaria para las sucesivas invocaciones encadenadas y para la posible recursión. Un despachador se encargará a su turno de activar el método correspondiente al destinatario elegido. Así visto, debemos considerar la equivalencia con el mecanismo normal de secuencia de sucesivas llamadas a subrutinas o procedimientos.

Tipo Abstracto de Datos, Cápsula:

Defino un Tipo Abstracto de Datos por las Entidades que ha de representar, sus Atributos, y las Operaciones que podré efectuar sobre ellos.

Al usar el término Tipo Abstracto de Datos no nos referiremos a temas de especificación formal, sino a la implementación de una entidad representada por una estructura de datos y todas las funciones para acceder o hacer uso de la misma, en una entidad única o en un tipo, que permita generar repetidas instancias de la misma.

Llamamos Cápsula al módulo que, en un lenguaje adecuado implemente un TAD. Tendrá así inicialización y permanencia de los datos de la estructura y las instancias que creo al representar cada una de las entidades de ese tipo, y contendrá la implementación de los procedimientos que me permiten efectuar las operaciones correspondientes.

Proveerá protección contra accesos externos y ocultamiento de información respecto de la estructura y su forma, y sólo comunicará y exportará al exterior los detalles respecto a su forma de uso e invocación, y su comportamiento.

Los lenguajes que apoyan estos conceptos permiten separar los componentes de estos módulos en dos partes: una de ellas con los datos descriptivos que hacen a su uso y vinculación con el resto del sistema, que llamaremos Definición, y otra con la construcción real de los procedimientos y estructuras que los implementan, a la que nos referiremos como Implementación.

En la Definición especificamos las interfases y en ella declararemos el nombre del tipo, que será así un Tipo de Datos Abstracto, opaco, con su contenido y operaciones en la Implementación. Manejamos y damos así la visibilidad deseada a cada item.

Aplicación de metodologías de Diseño:

(Booch, 86) detalla los pasos para aplicar la metodología de diseño por objetos, inicialmente propuestos por Abbott:
- Identificar los Objetos y sus atributos,
- Identificar las operaciones que sufre y que requiere cada Objeto,
- Establecer la visibilidad de cada objeto en relación con los otros,
- Establecer la interfase de cada objeto,
- Implementar cada Objeto.

Esta lista es muy semejante a los pasos a dar cuando uno selecciona y define una cápsula, que implementa un tipo abstracto de datos en un sistema.

Analicemos ahora el uso de las Metodologías derivadas del Diseño de estructura por Descomposición Funcional.

El comienzo es distinto cuando se parte de las funciones que se espera que cumpla el sistema. El primer refinamiento de esas funciones analiza las subfunciones principales. Este es el primer paso fundamental de Diseño. Pero inmediatamente, cuando se analizan funciones, debe considerarse sobre qué entidades se aplican éstas: datos o atributos de entrada o de estado del sistema, de sub-entidades permanentes o de entidades repetitivas o transitorias en el sistema.

Estas entidades pueden tener existencia en el mundo real o ser una creación del sistema de procesamiento.

Se pueden elegir a-priori, para modelizar, los objetos más significativos del mundo real y sus funciones características. Pero para representar éstos, probablemente debamos recurrir a algunos sub-componentes de los mismos.

No querremos representar todos los detalles de estos subcomponentes, sino sólo aquellos elementos y funciones necesarios para implementar las funciones que deseamos que el sistema lleve a cabo. Volvemos así al punto de partida de funciones y sub-funciones del sistema.

Algunas entidades son de puro origen computacional. La organización de entidades múltiples en estructuras como árboles, pilas, heaps, etc, hace a la eficiencia de los procesos. La selección de la estructura más eficiente depende fundamentalmente de las funciones que requeriremos de ella, que sólo podremos detectar en nuestro proceso de diseño.

Otras, como las ventanas o sistemas de menú, son necesidades de la interacción hombre máquina. Y sus atributos dependerán de las estrategias de uso del sistema.

Curiosamente, existen entidades computacionales o de cálculo, no ligadas a las computadoras, que casi parecerían tener existencia en el mundo real, como una "Cuenta Corriente" o un "consumo promedio". Podemos así pensar en dar vida a nuevas entidades de este tipo, si las requiere nuestro diseño.

Una vez definido un objeto o estructura por sus atributos y funciones, podremos comenzar a considerar su implementación, pero deberemos tener en cuenta los requerimientos de uso y respuesta de estas funciones. Para ello, la información esencial es la de frecuencia de uso de cada función, así como otros datos de volumen y eficiencia requerida.

Por supuesto, en su implementación, usaremos todos los recursos disponibles para ocultar y proteger la entidad, cumpliendo con los lineamientos de Parnas y construyendo Cápsulas.

En los objetos simples, cuya estructura fundamental es de tipo RECORD, este análisis no es significativo. Sí puede serlo el de las funciones, si implican procesos más complejos que observación, modificación o ingreso de un valor.

Las entidades computacionales, por otro lado, tienen una estructura básicamente compleja, lo que motiva su creación como entidad (u objeto) aparte. La elección de estrategias organizativas y de su utilización son tema central de diseño y a ellas dedicaremos nuestro esfuerzo.

Continuando con las Estrategias de Diseño, o elementos de metodologías que emplearemos, conviene anotar aquí las Máquinas de Estado, conceptualización que tendremos en cuenta para aquellas entidades de existencia individual y permanente, donde por supuesto, su comportamiento dependerá del estado que alcanzó en su trayectoria previa.

Tendremos que considerar, dentro de esta línea, los Módulos Incrementales y la estrategia de diseño, para los más complejos, de Inversión de Programa (Jackson, 75).

Por supuesto, entre las entidades de existencia individual y permanente debemos considerar al propio sistema y, por lo tanto, tener en cuenta este paradigma para su organización.

También siguiendo a Jackson, podemos dar estructura a los módulos de tratamiento e interpretación de comandos o mensajes con una gramática implícita.

En los refinamientos de arquitectura general, podemos considerar separarnos de la organización puramente jerárquica. Es evidente la racionalidad de esta alternativa para las situaciones de tipo Productor-Consumidor y así deberemos hacerlo si tenemos medios para implementar módulos a igual nivel en forma simple y elegante.

El manejo de Corrutinas permite estas soluciones al igual que da pie a la creación de objetos de tipo actor y su existencia y comportamiento simultáneos.

A este juego de estrategias y herramientas podemos agregar ahora, como derivado de la corriente orientada a objetos, el mecanismo de herencia, implementable como Tipos de Datos Derivados que permite separar entidades y estructuras en una forma no lograble y de dificil conceptualización sin este paradigma.

En el caso de objetos soportados por una estructura, esta separación se logra considerando a los objetos como herderos (es-un) de nodos de la estructura, que por su parte maneja a éstos en forma totalmente abstracta, sin conocer su significado o contenido, y con total separación de código.

Un enfoque distinto en el desarrollo de sistemas:

La metodología JSD (Jackson, 83) cuestiona todo el ciclo de vida tradicional y propone un proceso distinto, que comienza por la modelización del mundo real.

Pero inmediatamente, y en la misma justificación de la metodología, vincula modelo y función del sistema. Plantea como primera funcionalidad la de supervisión del sistema real, lo que permite realizar informes de actividad y comportamiento y comportamiento y detectar situaciones anormales.

Plantea, además, la modelización como forma de caracterizar la abstracción que encararemos del mundo real, y de corroborar su coherencia.

Los pasos que plantea la metodología son los siguientes:
- Caracterización de Entidades y Acciones del sistema
- Descripción de las acciones que ejecuta o sufre cada entidad en su organización temporal.
- Conexión entre los procesos del mundo real y los del modelo.
- Especificación de las funciones del sistema y las condiciones de ocurrencia e interrelación entre eventos.
- Consideraciones de tiempos en el sistema.
- Implementación del sistema.

Estos son pasos que cubren el espectro de la especificación del sistema y su implementación. Es interesante el análisis del paralelismo entre estas etapas y las especificadas por Abbott para el Diseño Orientado a Objetos.

En esta comparación debemos tener presente que la metodología JSD supone cubrir con las primeras etapas de modelización la especificación del sistema y que en las posteriores encara el diseño y la implementación funcional del mismo.

Lenguajes vinculados a Metodologías Orientadas a Objetos e incorporación de los conceptos de éstas en los lenguajes más tradicionales.

Los primitivos lenguajes procedurales disponían, como principal y casi único elemento vinculado a la éstructura, de la subrutina como paradigma de la abstracción algorítmica y la organización jerárquica (además de la localidad de variables y el paso de parámetros por interfase). Se soportaba, así, la descomposición funcional elemental. (Booch, 86)

El primer lenguaje en implementar Objetos fue Simula. Vinculado con el nacimiento de los lenguajes de Simulación (Dahl y Nygaard, 66), reafirma en esto nuestra idea de la vinculación de su uso e implementaciones en:

- La representación de Entidades del Mundo Real y
- La existencia y accionar de cada uno de ellos en forma simultánea e independiente, que no parece ser normalmente indispensable en un proceso definido en forma procedural.

También surgía de este enfoque la caracterización de objetos en Clases.

Por una línea aparte, se desarrollaron los lenguajes para implementar los conceptos de Tipos Abstractos de Datos como Alphard, CLU (Liskov, 81), Modula-2 (Wirth, 71) y como más inclusivo, Ada.

El concepto se hizo central en el enfoque del proyecto de Xerox PARC y la implementación de Smalltalk. (Goldberg y Robson, 83) (Goldstein y Bobrow, 80)

Se profundizó en la elaboración y aplicación de los conceptos de Herencia.

Sin embargo, dentro de esta escuela se promovieron otros conceptos que, por este hecho, quedaron adheridos al de Objetos. En especial, los de activación y paso de información a funciones y procedimientos.

A las características de métodos (procedimientos) ligados a Objetos y Clases (tipos abstractos de datos) y los mecanismos de Herencia, se unieron la activación y transferencia de información por Mensajes y el mecanismo de despacho para ejecución de cada uno de ellos.

Esto permitió dar un aspecto de no procedural al lenguaje, por el simple hecho de no indicarse cuando o en qué orden deben ejecutarse la funciones. Esto tiene mucha similitud con la característica de no determinismo de los mecanismos de iteración de E. Dijsktra y sus comandos con guardia, no tomados en sí como no procedurales.

Los lenguajes más avanzados de la actualidad poseen y soportan mecanismos de estructuración mucho más avanzados y flexibles. Para su uso adecuado deben considerarse otros tipos de abstracciones como la de datos, y la conceptualización como objetos es una forma directa de encararla. Los packages en Ada o módulos en Modula-2 permiten implementar estos enfoques.

En los principales lenguajes de programación, muchos de ellos de tipo procedural, se han incorporado mecanismos ad-hoc para implementar los conceptos de las metodologías de Objetos.

Así, en versiones de los lenguajes como Lisp, Pascal, C, se pueden definir y trabajar con Objetos, Clases, y aprovechar los mecanismos de Herencia con distintos niveles de generalidad.

Modula-2 (Wirth, 83) incorpora los principales mecanismos de localidad y permanencia de datos, ocultamiento de información por separación de la definición y la implementación, compilación por separado y manejo de la visibilidad de todos los componentes por importación y exportación.

Permite así implementar Tipos Abstractos de Datos en Cápsulas. También facilita las estructuras no jerárquicas por un directo manejo de corrutinas, que para nuestro estudio debemos vincular con la implementación de objetos actores.

Sin embargo, no podemos decir que permita en forma directa e integral representar algunos conceptos centrales de la metodología de Objetos.

Por supuesto, Ada incluye un conjunto de mecanismos que permiten implementar los conceptos derivados de las Metodologías Orientadas a Objetos. Esto lo podemos ver especialmente en el caso de los Tipos de Datos derivados.

El análisis conceptual realizado por (N. Wirth, 88) sobre extensión de tipos de datos busca definir nuevos tipos de datos, relacionados con uno existente tomado como base, cumpliendo reglas de compatibilidad.

La importancia de estas construcciones será significativa si el tipo base puede estar implementado por separado en un módulo compilado independientemente, que oculte sus detalles.

Su declaración es:

T´ = RECORD (T) ... declaraciones adicionales ... END

El tipo T´ es extensión del T . T es tipo base de T´.

El tipo T´tiene como componentes los declarados en la declaración extensiva más los que contiene T.

Podemos desconocer cuales son estos últimos, porque sólo se accede a ellos en el módulo que los declara o podemos conocerlos y tener acceso a los mismos.

Se pueden extender tipos ya extendidos previamente. También admite la Herencia múltiple.

Queremos usar este mecanismo para reproducir la relación Clase-Subclase: El Tipo Extendido (subclase) incorpora los atributos del Tipo Base (su clase).

Para ello debemos analizar los mecanismos que realizan el control de tipos y definir nuevas condiciones para los mismos. Nos interesa extender éstos para poder considerar a un elemento de un Tipo Extendido también como un elemento del Tipo Base.

Consideremos la asignación. Un tipo RECORD representa el producto cartesiano de sus tipos componentes. Un Tipo Base tendrá una dimensionalidad menor que la del Tipo Extendido. Para encontrar significación a la asignación de instancias de Tipo Extendido a entidades de Tipo Base debemos considerar la Proyección del primero sobre este último.

Resulta evidente que la asignación inversa no podría realizarse (quedarían componentes indeterminados), lo que concuerda con la asimetría de la relación Clase-Subclase.

(Wirth 88) extiende la definición ampliada a punteros a los Tipos Base y Extendido pero creo interesante analizar si esta extensión es necesaria.

Si los punteros del Tipo Base son sólo manejados dentro del módulo que los define, la única asignación de punteros del Tipo Extendido a los del Tipo Base se realiza en la invocación a funciones de este módulo (cápsula) para operaciones vinculadas con este tipo.

Wirth justifica ese relajamiento en las asignaciones de punteros por la muy importante posibilidad de construir estructuras con datos heterogéneos, por supuesto extensiones del Tipo Base. Pero si a esta estructura la consideramos como perteneciente a (las instancias de) el Tipo Base y protegida por la implementación del mismo, sería una estructura homogénea, que soporta integrantes heterogéneos en su diferenciación del Tipo Base.

Esta interpretación es paradójicamente apoyada por el mismo autor, en el punto en que considera la necesidad de la discriminación del tipo. Esta necesidad surge por el tratamiento de un puntero en el módulo del Tipo Base y la necesidad de interpretación del mismo en su devolución al ambiente más general, que reconoce las diferencias de los tipos extendidos.

Esto implica que, en el dominio de los punteros, cuando aceptamos uno de ellos como perteneciente a una Clase (Tipo Base), pasa a integrar un conjunto y no pierde su individualidad, que podremos reconocer al identificarlo para ser tratado en el entorno de los tipos extendidos.

Otra conceptualización posible es la de que los punteros no poseen tipo propio sino representan al tipo de lo apuntado. Y es esto último lo que se consultará para la discriminación.

Plantea Wirth para esto una operación de prueba de tipo (v IS T : BOOLEAN) y una muy interesante guardia de tipo, que implicará que la aplicación de la operación está condicionada a que los componentes correspondan al tipo especificado o a una extensión de éste.

```
v : T
v := x (T)      condiciona la ejecución a que x sea del tipo T o
                una extensión de éste.

Type
     T = RECORD
                    c : ...        x (T). c  condiciona el acceder
                    .....     al campo  c  a que  x  sea de tipo  T
         END
```

Polimorfismo en procedimientos: Esta propiedad implica poder aceptar para su tratamiento, no sólo parámetros de un único tipo, sino una variedad de éstos.

Se relajan así las reglas de control de tipo. La definición de este relajamiento creada para los tipos extendidos cubre esta necesidad.

Es en esta asignación de valores a los parámetros formales de la interfase, que proponemos la consideración (y relajamiento) de los tipos de datos correspondientes a punteros.

Beneficios de las Metodologías Orientadas a Objetos:

Una característica significativa de estas metodologías es que fuerzan a trabajar de forma tal que se promueven buenas prácticas de Diseño:

- Una correspondencia más directa con los entes reales a simular o representar
- Modularidad
- Encapsulamiento, con bajo acoplamiento y ocultamiento
- Reusabilidad
- Facilidad de extensión.

Los seguidores de estas estrategias enfatizan los siguientes aspectos (Jacky y Kalet, 87):

- Más fácil modelización del mundo real o del comportamiento de los items a ser modelizados y mejor (más directo) diseño del sistema.
- Menos errores, por modularidad y encapsulamiento.
- Más fácil de extender, por facilidad de incorporar nuevos objetos (clases). Cada nuevo objeto (clase) se incorpora con todos sus métodos.

Estas características deseables de las Estructuras de los Sistemas de Soft ya habían sido incorporadas en metodologías y herramientas anteriores de Diseño.

Analicemos un poco más el último argumento. Se supone en él que se querrá incorporar objetos nuevos más frecuentemente que nuevas operaciones que afecten varias Clases existentes. Si este último fuera el caso, habría que tocar cada objeto para definir el método correspondiente para la nueva operación.

Objeta que en la organización "clásica procedural" cada operación tiene un CASE orientado a cada tipo de objetos, que debería ampliarse al incorporar nuevos. Esto no es así si hemos implementado cápsulas por objetos.

En resumen, opinamos que las Metodologías Orientadas a Objetos:
- Ayudan a que los que se inician incorporen buenas costumbres de diseño.
- Son una estrategia más a aprovechar.
- Parte de los conceptos y las herramientas desarrolladas para soportarlos permiten ampliar y dar más fuerza a las buenas estrategias avanzadas de Diseño de Estructura e implementar lo ya concebido conceptualmente.

Aporte significativo al Diseño de Soft: Mayor separación funcional

Separar conceptos equivale a separar código. Las nuevas herramientas y conceptualizaciones pueden ayudar en este sentido.

Si bien corresponden a funciones diferentes, si una entidad está dentro de un árbol (o cualquier otra estructura), las intrucciones para pasar al próximo elemento están en general mezcladas con las de procesar el nodo. Y los datos y punteros también, todos en la misma cápsula.

Tenemos ahora herramientas para separarlas. Puedo ahora implementar operaciones independientes sobre:
- un elemento,
- la estructura que los soporta y organiza (arbol, jerarquía, secuencia)
- los elementos auxiliares de representación: tipos elementales, contenidos o ventanas, etc.

Esta posibilidad, que no sólo promueve el desacoplamiento entre funciones sino que permite crear módulos independientes con funciones esenciales y así dar un sólido material para reusabilidad, puede llegar a ser más importante que los otros frutos de las metodologías orientadas a objetos.

Análisis de los argumentos sobre Modelización del Mundo Real por Objetos:

Las Metodologías Orientadas a Objetos se prestan mucho para expresar o modelizar el mundo real. Un mundo de complejidad baja, en que no haya todavía una significativa tradición de Sistemas de Información (o no la tenga el implementador o usuario).

En los sistemas de complejidad algo mayor o con tradición de Sistemas de Información, hay que hilar más fino e interactuar con archivos, otros sistemas de información, la estructura de los elementos de soporte y cómputo, y crear organizaciones complejas para hacer eficiente el proceso.

Todo esto implica el trato con un conjunto importante de datos, estructuras y procedimientos que podemos seguir manejando con la metodología de objetos, pero que, forzar este enfoque puede ser una imposición pesada, por aferrarse a la metodología, más que una solución práctica y con iguales propiedades de claridad y controlabilidad de la solución que su aplicación a los objetos básicos.

Un ejemplo extremo lo tendríamos si deseáramos hacer un sistema para jugar al ajedrez. Tomaríamos las piezas y los casilleros del tablero como objetos y sus movimientos posibles como métodos. Será así muy facilmente manejable la representación de una posición dada o la visualización de las jugadas. Pero rápidamente notaríamos que el meollo del problema no ha sido atacado. Sabemos que por esta via sólo se tiene una solución impracticable, por la enorme cantidad de alternativas que deben ser analizadas para evaluar una jugada. Y deberemos crear una cantidad de estrategias y conceptos, medidas y evaluaciones para las que la conceptualización como objetos puede ser una carga intelectual más.

Aún en los casos simples, el verse forzado a considerar que 3 + 4 es un mensaje cuyo receptor es el objeto 3, siendo + es el selector del método a aplicar y 4 el argumento, puede resultar muy pesado si se lo toma como algo más que un divertimento conceptual...

Todas las funciones, disponibles:

Una de las propiedades que se promueven en el uso de un lenguaje orientado a objetos es el poder disponer permanentemente de todas las funciones definidas para todos los objetos ya especificados en el sistema.

La imagen de esta característica, desde el punto de vista de un lenguaje procedural equivale a un ambiente donde todas las funciones o procedimientos definidos previamente estan accesibles para ser llamados por el sistema que estoy armando.

Esto equivaldría a un ambiente en Pascal donde no comienzo a realizar un programa desde cero, sino que incluyo inicialmente todas las funciones definidas previamente. Esta visión sería más natural si pensamos en un ambiente como el de APL, donde en un workspace estan realmente presentes todas las funciones definidas en el sistema. Como operación característica existe la consulta de la lista de operaciones disponibles en el espacio.
Algo equivalente encontramos en ambientes LISP.

Rápidamente detectamos la facilidad para disponer de esta propiedad en lenguajes interpretados. En ambientes de lenguajes compilados, la unidad con que se trabaja es el elemento a compilar, que no se sobrecarga con funciones que podrían o no ser utilizadas por el sistema en desarrollo.

Este problema es encarado por los lenguajes más modernos, donde se considera fundamental la compilación por separado, y la posibilidad de invocar funciones ubicadas y provistas por otros módulos, que serán vinculadas al actualmente en desarrollo en momento de linkedición. Tendremos así acceso a todo el conjunto de funciones, que estarán ubicadas en nuestra biblioteca de módulos. Como recordaremos, esta organización era utilizada en la década del 60...

Por otra parte, el concepto de compilación por separado se vincula con el de ocultamiento de información y el de limitación del acceso a las funciones y la descripción de su implementación. Por lo tanto, la existencia de todas las funciones en un mismo ambiente, en un lenguaje en el que quisiéramos imitar la programación orientada a objetos pecaría por falla en este sentido, salvo que el lenguaje proveyera dispositivos para este fin (funciones protegidas, etc.). Otro problema relacionado es el de nombres de las funciones, que deberán tener mecanismos de diferenciación en su incorporación o importación.

Encontramos este mismo tipo de propiedad en ambientes con sistema operativo transparente, es decir, donde el sistema operativo no se hace evidente al usuario, quien considera estar siempre expresándose en lenguaje de programación o comando, en el mismo nivel. Aquí todo está a su alcance, para invocación o uso desde cualquier sistema.

El diseño tradicional no implica el uso generalizado de un Common:

Leemos, en un trabajo donde se promueven metodologías de diseño orientadas a objetos que "Finalmente en los diseños con las metodologías de descomposición en funciones, se termina poniendo todas las variables en acceso global...". Esto es ignorar reglas fundamentales de la metodología propuesta por Constantine, Yourdon y seguidores donde, en las mediciones de bondad de diseño, se penaliza al ubicar las variables y las estructuras en un common o ambiente global con las peores notas en grado de acoplamiento, medida fundamental del diseño. Esta calificación es síntoma evidente de un diseño que debe reverse.

Por supuesto debemos transar con las limitaciones que nos plantea el lenguaje con que podemos trabajar y así, con Fortran, Cobol y aún Pascal, deberemos aceptar el uso real de esas variables y estructuras como globales, cualquiera sea la metodología de diseño que empleemos.

Paralelismo o simultaneidad de ocurrencia:

Esta propiedad se puede implementar con corrutinas, definiendo la sincronización entre ellas (Productor - Consumidor o varias corrutinas en interacción).
Puedo elegir tener algunos módulos a igual nivel y otros en relación jerárquica. Dispondré de un dispatcher diseñado por mi, con mi estrategia, que existirá sólo si lo necesito.

Podemos además hacer consideraciones sobre que la M O O presupone que:
- El sistema será sólo la suma e interconexión de sus objetos.
- Todos sus objetos son totalmente separables.

Conclusiones:

Las Metodologías Orientadas a Objetos
- No parecen un cambio significativo en cuanto a Diseño, respecto de los enfoques de Modularización encarados por Parnas, Constantine, Yourdon y seguidores y de Tipos Abstractos de Datos e implementados en lenguajes como Alphard, CLU y más prácticamente en Ada y Modula-2 y hasta en paquetes comerciales como Turbo Pascal 4.0
- Sí parecen ser un buen vehículo para imponer estos mismos conceptos, por el peso de difusión que han logrado los términos usados. En ésto es equivalente a lo que la terminología Programación Estructurada logró respecto a la disciplina de programación propuesta por N. Wirth.
- Establecen una disciplina a la que en otra forma no se lograba gran éxito de adhesión.

Las herramientas implementadas en los lenguajes procedurales por la corriente generada por las tendencias de adhesión a la moda de programación por objetos parecen proveernos de un herramental muy poderoso para llevar adelante los conceptos e ideas del Diseño Estructurado (o Composite Design según G. Myers).

Tomemos como recomendación el no "encandilarnos" con un "paradigma" deslumbrante. Vale más conocer a fondo los objetivos de nuestro diseño y las métricas que las metodologías disponibles nos proveen, y tomar de cada moda los elementos positivos que aporta y las herramientas que ese esfuerzo permitió implementar.

Conocer los objetivos del diseño implica, muchas veces, conocer el camino recorrido en el desarrollo de los conceptos y técnicas y las razones y méritos de su afirmación o desplazamiento en la búsqueda, a veces recurrente, de soluciones que se apoyan en las previas para construir nuestro herramental.

Bibliografía:

Atkinson M. P. y Morrison R. 1985. Procedures as persistent data objects. ACM TOPLAS 7, 4, Oct. 539-559.

Atkinson M. P. y Buneman P. O. 1987. Types and Persistence in Database programminglanguages. ACM Comp.Surv. 19,2 Jun.105-190.

Booch,G. 1986. Object-Oriented Development. IEEE Trans.Softw.Eng. SE-12, 2 (Feb.86), 211-221.

Dahl, O. y Nygaard, K. 1966. Simula, an Algol-based simulation language. Commun.ACM 9, 9 Sept. 671-678.

Goldberg, A. y Robson, D. 1983. Smalltalk-80: The language and its implementation. Addison Wesley, Reading, Mass. USA.

Goldstein, I. P. y Bobrow, D. G. 1980. Extending Object Oriented programming in Smalltalk. Proceedings of the 1980 Lisp Conference. Ago, ACM, 75-81.

Ichbiah, et al. 1979. Rationale of the design of the programming language Ada. En ACM SIGPLAN Not, 14, 6

Jacky J. P. y Kalet, I. R. 1987. An Object-Oriented Programming discipline for standard Pascal. Commun.ACM 30, 9 Sept. 772-776.

Jackson, M. A. 1975. Principles of program design. Academic Press, Londres.
Jackson, M. A. 1983. System development. Prentice-Hall. Londres.
Liskov, B., Synder, A., Atkinson, R., y Schiffert, C. 1977. Abstraction mechanisms in CLU. Commun.ACM 20, 8 Ago 564-576.
Liskov, B. 1981. CLU Reference manual, Goos and Hartmanis, Eds. Lecture notes in Computer Science, vol. 114. Springer-Verlag, Berlín.
Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. Commun.ACM 15, 12 Dic. 1053-1058.
Soustrup, B. 1986. An overview of C++. SIGPLAN Not.21,10 Oct.7-18.
Yourdon, E.y Constantine, L. 1979..Structured Design. Fundamentals of a discipline of computer programs and systems design. Prentice-Hall,Englewood Cliffs, NJ. USA.
Wirth, N. 1971. Program development by stepwise refinement. Commun.ACM 14, 4. 221-227
Wirth, N. 1983. Programming in Modula-2. 2nda ed. Springer-Verlag, Berlín.
Wirth, N. 1988. Type extensions. ACM TOPLASS 10, 2 Abr. 204-214.