PUC

Series: Monografias em Ciência da Computação

Nº 4/81

VERIFICATION AND TESTING OF SIMPLE ENTITY-RELATIONSHIP REPRESENTATIONS

Antonio L. Furtado
Paulo A. S. Veloso
José M. V. de Castilho

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP-22453
RIO DE JANEIRO - BRASIL

Series: Monografias em Ciência da Computação

Nº 4/81

Series Editor: Marco A. Casanova April 1981

VERIFICATION AND TESTING OF SIMPLE ENTITY - RELATIONSHIP REPRESENTATIONS*

Antonio L. Furtado
Paulo A. S. Veloso
José M. V. de Castilho§

* Research partly sponsored by FINEP and CNPq.

§ On leave from Universidade Federal do Rio Grande do Sul

ABSTRACT

A methodology is proposed for representing a data base application on a simple entity-relationship data model , based on formally specifying both as abstract data types. This methodology inludes the verification and testing of the representation, which are simplified by the usage of procedural specifications. An example data base application is used to illustrate the general discussion.

Key words:

Abstract data type, data base, data model, representation, entity-relationship approach, correctness of representation, verification, testing.

RESUMO:

Propõe-se uma metodologia para representar uma aplicação de banco de dados em um modelo de dados entidades-relacionamentos simplificado, a qual se baseia em especificações formais de ambos como tipos abstratos de dados. Esta metodologia inclui a verificação e o teste da representação, que são simplificados pelo uso de especificações procedurais. Um exemplo de aplicação de banco de dados é usado para ilustrar a discussão geral.

PALAVRAS CHAVES:

Tipo abstrato de dados, banco de dados, modelo de dados, representação, enfoque entidades-relacionamentos, corretude de representação, verificação, teste.

1. INTRODUCTION

Capturing the intended behavior of a data base application from informal descriptions supplied by its prospective users is a critical task. Misunderstandings are often perceived only too late, e.g. when an executable implementation becomes available after lengthy and costly efforts.

Our approach to this problem involves the following steps:

a) Express the informal description of the data base application formally as an abstract data type [GTW,GUT], keeping however the very same terminology of the data base application.
b) Represent the application data type by an abstract data type corresponding to the data model.
c) Verify formally the correctness of the representation.
d) Test the representation against the original specification.

Notice that we stress testing as an opportunity for the users to confirm the faithfulness of the specification to their perhaps vague mental image. Since it is impossible to prove the equivalence between the initial informal specification and the formal ones, the availability of experimental usage is paramount to confirming that the original intentions were captured [SHA,VCF].

In line with the above remarks we shall use the formalism of procedural specifications [FVE]. It presents the advantage of allowing early usage and testing by means of symbolic programs.

As data model we shall employ the entity-relationship data model [CHE]. It is important to note that the entity-relationship view has been praised for its closeness to real-world situations but has been regarded as informal [ULL], whereas we shall present it here in the same precise data type formalism. For the sake of simplicity, we shall confine ourselves to a simplified version of the entity-relationship model presented in section 2.

In section 3 we introduce a simple data base application, which will be used as a running example in order to illustrate the main ideas involved in representing a data base application by the data model as well as verifying and testing the representation.

2. SPECIFICATION OF THE S-ER DATA MODEL

In order to simplify the presentation, we shall confine ourselves to a restricted version of the entity-relationship data model to be denominated the S-ER data model. The S-ER data model supports only binary relationships and allows attributes for entities but not for relationships. The remaining features of the full ER model appear to be easily incorporated.

Its update operations permit to initialize (phi) the data base to an empty state, create and delete (cr,del) entities within entity-sets, modify (mod) values of attributes ('*' stands for the undefined value) and link or unlink (lk,ulk) entities via a relationship. The query operations are predicates referring to the existence (exs) of entities within entity-sets, values (hv) of attributes and relatedness (isr) of entities.

An obvious integrity constraint is that only entities that exist in the data base may have defined values for attributes and may be related to other entities. Dynamically this constraint is enforced by causing the operations that assign values to attributes or establish links to have no effect if the entities involved do not exist. On the other hand, an entity that exists in only one entity-set cannot be deleted until all its attribute-values are set to undefined and all incident links are removed.

Any ser-object can be created through - and can therefore be denoted by - expressions involving applications of the update operations. It is possible to identify sets of expressions that denote the same ser-object, but one may choose representatives for each one of those sets, defining a convenient canonical form containing only some of the update operations. These operations are phi, cr, mod and lk. The canonical terms will therefore contain only these operations, arranged in the sequence

```
lk(...lk(mod(...mod(...cr(...cr(...phi( )...)...)...)...)...)...)
```

Occurrences of the same operation are ordered lexicographically with respect to their arguments (apart from the canonical term argument), in increasing sequence, from left to right; the implied order of execution being inside out: the first is phi and the last is the most external operation.

However, not all expressions conforming to the above syntax correspond to canonical terms. The procedural specification [FVE] given in Figure 1 can be shown to generate only expressions that are valid with respect to the integrity constraints.

Each procedure has commands corresponding to the rewriting rules which transform a data base expressed by a canonical term into another canonical term, when an additional operation (corresponding to the procedure) is applied. Whenever update operations depend on integrity constraints for their applicability, these are checked at the outset and the canonical term given as argument is returned unchanged in case of failure. The remaining part of the procedure bodies is a <u>case-like statement</u>, inside which occurs the recursive scanning of the canonical term argument.

The language features are self-explanatory except perhaps for "?", which stands for any valid value of an argument, and "?<variable>" which, in addition, assigns the value found to a variable, as in PLANNER [HEW]. In order to improve readability, the canonical terms are written using square brackets instead of parentheses and "|" instead of comma.

If the operations in the expression below

```
mod(A,ATTR,10,lk(A,B,REL,del(B,ES2,cr(A,ES1,cr(B,ES2,phi( ))))))
```

are executed, the resulting canonical term is

```
type ser
sorts  ser,ent,eset,attr,val,rel,logical

  op phi( ):ser
     ⇒ phi
  endop

  op cr(x:ent,t:eset,s:ser):ser
     var y:ent,z:ent,u:eset,a:attr,i:val,r:rel,s1:ser
     exs(x,t,s) ⇒ s;
     match s
       LK[y|z|r|s1] ⇒ LK[y|z|r|cr(x,t,s1)]
       MOD[y|a|i|s1] ⇒ MOD[y|a|i|cr(x,t,s1)]
       CR[y|u|s1] ⇒ if x.t > y.u then CR[y|u|cr(x,t,s1)]
                     else CR[x|t|s]
       otherwise ⇒ CR[x|t|s]
     endmatch
  endop

  op mod(x:ent,a:attr,i:(val,{*}),s:ser):ser
     var y:ent,z:ent,b:attr,j:val,r:rel,s1:ser
     ∿ (exs(x,?,s) ∧∿ hv(x,a,i,s)) ⇒ s;
     match s
       LK[y|z|r|s1] ⇒ LK[y|z|r|mod(x,a,i,s1)]
       MOD[y|b|j|s1] ⇒ if x.a = y.b then
                          if i = * then s1
                          else MOD[x|a|i|s1]
                       else if x.a > y.b then
                              MOD[y|b|j|mod(x,a,i,s1)]
                            else MOD[x|a|i|s]
       otherwise ⇒  MOD[x|a|i|s]
     endmatch
  endop

  op lk(x:ent,y:ent,r:rel,s:ser):ser
     var z:ent,w:ent,q:rel,s1:ser
     ∿(exs(x,?,s) ∧ exs(y,?,s) ∧∿ isr(x,y,r,s)) ⇒ s;
     match s
       LK[z|w|q|s1] ⇒ if x.y.r > z.w.q then
                         LK[z|w|q|lk(x,y,r,s1)]
                      else LK[x|y|r|s]
       otherwise ⇒ LK[x|y|r|s]
     endmatch
  endop

  op del(x:ent,t:eset,s:ser):ser
     var y:ent,z:ent,u:eset,v:eset,a:attr,i:val,r:rel,s1:ser
     ∿(exs(x,t,s) ∧ (inothereset(x,?v,t,s) ∨
                     (∿ isr(x,?,?,s) ∧∿ isr(?,x,?,s) ∧∿ hv(x,?,?,s)))) ⇒ s;
     match s
       LK[y|z|r|s1] ⇒ LK[y|z|r|del(x,t,s1)]
       MOD[y|a|i|s1] ⇒ MOD[y|a|i|del(x,t,s1)]
       CR[y|u|s1] ⇒ if x.t = y.u then s1
                    else CR[y|u|del(x,t,s1)]
     endmatch
  endop
```

```
op ulk(x:ent,y:ent,r:rel,s:ser):ser
   var z:ent,w:ent,q:rel,s1:ser
   ∿ isr(x,y,r,s) ⇒ s;
   match s
      LK[z|w|q|s1] ⇒ if x.y.r = z.w.q then s1
                     else LK[z|w|q|ulk(x,y,r,s1)]
   endmatch
endop

op exs(x:ent,t:eset,s:ser):logical
   var y:ent,z:ent,v:eset,a:attr,i:val,r:rel,s1:ser
   match s
      LK[y|z|r|s1] ⇒ exs(x,t,s1)
      MOD[y|a|i|s1] ⇒ exs(x,t,s1)
      CR[y|v|s1] ⇒ if x.t = y.v then true
                   else if x.t > y.v then exs(x,t,s1)
                        else false
      otherwise ⇒ false
   endmatch
endop

op hv(x:ent,a:attr,i:val,s:ser):logical
   var y:ent,z:ent,b:attr,j:val,r:rel,s1:ser
   match s
      LK[y|z|r|s1] ⇒ hv(x,a,i,s1)
      MOD[y|b|j|s1] ⇒ if x.a.i = y.b.j then true
                      else if x.a > y.b then hv(x,a,i,s1)
                           else false
      otherwise ⇒ false
   endmatch
endop

op isr(x:ent,y:ent,r:rel,s:ser):logical
   var z:ent,w:ent,q:rel,s1:ser
   match s
      LK[z|w|q|s1] ⇒ if x.y.r = z.w.q then true
                     else if x.y.r > z.w.q then isr(x,y,r,s1)
                          else false
      otherwise ⇒ false
   endmatch
endop

hidden op inothereset(x:ent,v:eset,t:eset,s:ser):logical
   ⇒ exs(x,v,s) ∧ v ≠ t
endop

endtype
```

FIGURE 1

MOD[A|ATTR|10|CR[A|ES1|PHI]] .

which would also be the result of executing

mod(A,ATTR,10,CR[A|ES1|PHI])

This "backtracking" property of canonical term specifications [GTW] will be found useful in the sequel.

3. SPECIFYING A DATA BASE APPLICATION

As an example of a (simplified) data base application, we shall use the data base of an employment agency, where persons apply for positions, companies subscribe by offering positions, and persons are hired by or fired from companies. A person applies only once, thus becoming a candidate to some position; after being hired, the person is no longer a candidate but regains this status if fired. The same company can subscribe several times, the (positive) number of positions being added up. Only persons that are currently candidates can be hired and only by companies that have at least one vacant position. One consequence of these integrity constraints is that a person can work for at most one company.

Apply, _subscribe_, _hire_ and _fire_, together with _initag_ (which creates an initially empty agency data base) are our _update operations_. As _query operations_ we shall use the predicates _iscandidate_, _worksfor_ and _haspositions_, which refers to the number of unfilled positions in a company.

Any agency data base (_agdb_) object will be created through - and can therefore be denoted by - expressions involving applications of the update operations _initag_, _apply_, _subscribe_ and _hire_. The canonical terms will therefore contain only these operations, arranged in the sequence:

hire(...subscribe (...apply (...initag( )...)...)...)

Occurrences of the same operation are ordered lexicographically with respect to their first argument (person for hire and apply, company for subscribe), in increasing sequence, from left to right; the order of execution is from the inside out: the first is initag and the last is the most external operation.

Figure 2 shows the procedural specification of the agdb data type. If the operations in the expression below

```
fire(E3,C2,hire(E2,C2,hire(E1,C2,subscribe(C2,1,hire(E1,C1,
hire(E4,C1,apply(E1,hire(E3,C2,apply(E2, apply(E4,subscribe(C2,3,
apply(E3,subscribe(C1,2,initag( ))))))))))))))
```

are executed, the resulting canonical term is

```
HIRE[E1|C1|HIRE[E2|C2|HIRE[E4|C1|SUBSCRIBE[C1|2|
SUBSCRIBE[C2|4|APPLY[E1|APPLY[E2|APPLY[E3|APPLY[E4|INITAG]]]]]]]]]
```

Notice that the same result would be obtained, in view of the "backtracking" property, with the execution of, e.g.

```
hire(E1,C1,hire(E2,C2,hire(E4,C1,subscribe(C1,2,
SUBSCRIBE[C2|4|APPLY[E1|APPLY[E2|APPLY[E3|APPLY[E4|INITAG]]]]]))))
```

4. REPRESENTING THE DATA BASE APPLICATION BY THE DATA MODEL

We are viewing both the data application and the data model as abstract data types. Thus, representing the former by the latter consists of implementing one data type by another [GTW,GUT,HOA].

In our example of the employment agency data base, it seems natural to represent persons (candidates and employees) and companies as entities and agdb-objects as ser-objects. This establishes a correspondence from the non-primitive sorts of type agdb into the sorts of type ser (mirrored in the sort definitions of Figure 3). We now have to refine this correspondence to a function rep assigning to each agdb-object an ser-object representing it. This can be done as follows.

```
type agdb
sorts agdb,person,company,natural,logical

op initag():agdb
   ⇒ INITAG
endop

op apply(x:person,s:agdb):agdb
   var z:person, w:company, m:natural, t:agdb
  ~(~iscandidate(x,s) ∧~ worksfor(x,?,s)) ⇒ s;
   match s
      HIRE[z|w|t] ⇒ HIRE[z|w|apply(x,t)]
      SUBSCRIBE[w|m|t] ⇒ SUBSCRIBE[w|m|apply(x,t)]
      APPLY[z|t] ⇒
         if x>z
         then APPLY[z|apply(x,t)]
         else APPLY[x|s]
      INITAG     ⇒ APPLY[x|s]
   endmatch
endop

op subscribe(y:company,m:natural,s:agdb):agdb
   var x:person, t:agdb, w:company, n:natural
   ~(m>0) = s;
   match s
      HIRE[x|w|t] ⇒ HIRE[x|w|subscribe(y,m,t)]
      SUBSCRIBE[w|n|t] ⇒
         if y=w
         then SUBSCRIBE[y|n+m|t]
         else if y>w
              then SUBSCRIBE[w|n|subscribe(y,m,t)]
              else SUBSCRIBE[y|m|s]
      APPLY[x|t] ⇒ SUBSCRIBE[y|m|s]
      INITAG     ⇒ SUBSCRIBE[y|m|s]
   endmatch
endop

op hire(x:person,y:company,s:agdb):agdb
   var m:natural, z:person, w:company, t:agdb, n:natural
   ~(iscandidate(x,s) ∧ (haspositions(y,?m,s) ∧ m>0) ⇒ s;
   match s
      SUBSCRIBE[w|n|t] ⇒ HIRE[x|y|s]
      HIRE[z|w|t] ⇒
         if x > z
         then HIRE[z|w|hire(x,y,t)]
         else HIRE[x|y|s]
   endmatch
endop
```

```
op fire(x:person,y:company,s:agdb):agdb
   var t:agdb, w:company, z:person
   ~worksfor(x,y,s) ⇒ s
   match s
      HIRE[z|w|t] ⇒
         if x>z
         then HIRE[z|w|fire(x,y,t)]
         else t
   endmatch
endop

op iscandidate(x:person,s:agdb):logical
   var z:person, w:company, t:agdb, m:natural
   match s
      HIRE[z|w|t] ⇒
         if x=z
         then false
         else iscandidate(x,t)
      SUBSCRIBE[w|m|t] ⇒ iscandidate(x,t)
      APPLY[z|t] ⇒
         if x=z
         then true
         else iscandidate(x,t)
      INITAG    ⇒ false
   endmatch
endop

op haspositions(y:company,m:natural,s:agdb):logical
   var z:person, w:company, t:agdb, n:natural
   match s
      HIRE[z|w|t] ⇒
         if w=y
         then haspositions(y,m+1,t)
         else haspositions(y,m,t)
      SUBSCRIBE[w|n|t] ⇒
         if w≠y
         then haspositions(y,m,t)
         else if m=n
              then true
              else false
      APPLY[z|t] ⇒ false
      INITAG    ⇒ false
   endmatch
endop

op worksfor(x:person,y:company,s:agdb):logical
   var z:person, w:company, t:agdb, m:natural
   match s
      HIRE[z|w|t] ⇒
         if x=z and y=w
         then true
         else worksfor(x,y,t)
      SUBSCRIBE[w|m|t] ⇒ false
      APPLY[z|t] ⇒ false
      INITAG    ⇒ false
   endmatch
endop
endtype
```

FIGURE 2

For each basic operation op-agdb, an operation op-ser is defined by means of a procedure using the basic operations of the ser data type, so that the following diagram commutes.

$$\begin{tikzcd}
\text{agdb} \arrow[r, "\text{op-agdb}"] \arrow[d, "\text{rep}"'] & \text{agdb} \arrow[d, "\text{rep}"] \\
\text{ser} \arrow[r, "\text{op-ser}"'] & \text{ser}
\end{tikzcd}$$

Figure 3 shows a procedural specification of such a representation. Some points are worth mentioning. Firstly, the representation is independent of any particular implementation of the data model. Secondly, the representation function <u>rep</u> is defined implicitly only by viewing a sequence of agdb operations as calls to the corresponding procedures in the representation. Thirdly, the data model needs only the entity-sets CAND, EMP and COMP, only one attribute NPOS (associated with entities from the COMP set), and only one relationship set, WORKS, linking entities from EMP to entities from COMP. Lastly, the integrity constraints of agdb are to be respected in the representation, the data model having of course its own integrity constraints.

5. VERIFYING THE REPRESENTATION

As stated in section 4 a representation of agdb by ser implicitly defines a map rep:agdb → ser and verifying the correctness of the representation amounts to verifying the commutativity of a diagram for each basic agdb-operation. An expanded version of such a diagram appears in Figure 4, where the upper and the lower paths have been decomposed into three steps each. We have to start with a generic agdb-object C, which will be mapped to a ser-object e, via the upper path, and to a ser-object g, via the lower path. In order to check the equality of e and g (which are given by sequences of ser operations), we use the procedural specification of ser and

```
repr of agdb by ser

sort definitions
     agdb : : = ser
     person ::= ent
     company ::= ent
{sorts natural and logical are primitive}

op initag( ):agdb
   ⇒ phi( )
endop

op apply(x:person,s:agdb):agdb
   ∿(∿ exs(x,CAND,s) ∧∿ isr(x,y,WORKS,s)) ⇒ s;
      ⇒ cr(x,CAND,s)
endop

op subscribe(y:company,m:natural,s:agdb):agdb
   var n:natural
   ∿(m>0) ⇒ s;
   hv(y,NPOS,?n,s) ⇒ mod(y,NPOS,n+m,s);
   ⇒ mod(y,NPOS,m,cr(y,COMP,s))
endop

op hire(x:person,y:company,s:agdb):agdb
   var n:natural
   ∿(exs(x,CAND,s) ∧ hv(y,NPOS,?n,s) ∧ n>0) ⇒ s;
   ⇒ lk(x,y,WORKS,mod(y,NPOS,n-1,cr(x,EMP,del(x,CAND,s))))
{the value of n in the last command is obtained by the ?n construction,
 as in PLANNER}
endop

op fire(x:person,y:company,s:agdb):agdb
   var n:natural
   ∿isr(x,y,WORKS,s) ⇒ s;
   hv(y,NPOS,?n,s) ⇒ ulk(x,y,WORKS,mod(y,NPOS,n+1,
      cr(x,CAND,del(x,EMP,s))))
{the effect of the condition hv(y,NPOS,?n,s) is simply retrieving the
 value of n}
endop

op iscandidate(x:person,s:agdb):logical
   ⇒ exs(x,CAND,s)
endop

op haspositions(y:company,m:natural,s:agdb):logical
   ⇒ hv(y,NPOS,m,s)
endop

op worksfor(x:person,y:company,s:agdb):logical
   ⇒ isr(x,y,WORKS,s)
endop

endrepr agdb/ser
```

FIGURE 3

$$\begin{tikzcd}
C \arrow[r, "1.a"] \arrow[d, swap, "2.a"] & D \arrow[r, "1.b"] & d \arrow[dd, "1.c"] & \text{(agdb)} \\
c \arrow[d, swap, "2.b"] & & & \\
f \arrow[r, "2.c"] & g \arrow[d, swap, "3.b"] & e \arrow[d, "3.a"] & \text{(ser)} \\
& B \arrow[r, phantom, "="] & A & \text{(canonical ser)}
\end{tikzcd}$$

FIGURE 4 Steps to verify the correctness of the representation.

check whether the corresponding canonical terms, A and B, are syntactically identical.

The steps involved are as follows

- upper path

1a: operation op-agdb given by the prodecural specification (Figure 2), giving

D = HIRE[E|C|...|SUBSCRIBE[C|n|...|APPLY[E|...|INITAG...]...]...]

1b: application of the backtracking property for agdb canonical terms, yielding

d = hire(E,C,...,subscribe(C,n,...apply(E,...,initag( )...)...)...)

1c: execution of the procedures according to the representation (Figure 3), giving

e = lk(E,C,WORKS,mod(C,NPOS,n-1,cr(E,EMP,del(E,CAND,...,mod(C,NPOS,n,... ...cr(E,CAND,...,phi( )...)...)...))))

- lower path

2a: application of the backtracking property for agdb canonical terms, yielding

c = hire(P,D,...,subscribe(D,...,apply(P,...,initag( )...)...)...)

2b: execution of the procedures according to the representation (Figure 3) giving

f = lk(P,D,WORKS,mod(D,NPOS,j-1,cr(P,EMP,del(P,CAND,...,..., mod(D,NPOS,k,...,cr(P,CAND,...,phi( )...)...)...)))

2c: execution of the procedure op-ser on f according to the representation (Figure 3), yielding g.

Notice that both e and g are sequences of basic ser operations. By applying to them the procedural specification for the data model (Figure 1) we obtain the canonical terms A and B which are to be identical if the representation is correct.

As an illustration consider the simple case of the operation of firing E2 from C2, on the agdb state given by the canonical term

C = HIRE[E1|C1|HIRE[E2|C2|SUBSCRIBE[C1|2|SUBSCRIBE[C2|3
    APPLY[E1|APPLY[E2|APPLY[E3|INITAG]]]]]]]

By applying to this term the agdb operation fire(E2,C2,.), we obtain

D = HIRE[E1|C1|SUBSCRIBE[C2|2|SUBSCRIBE[C2|3|APPLY[E1|APPLY[E2|
    APPLY[E3|INITAG]]]]]]

which, by the backtracking property, is

d = hire(E1,C1,subscribe(C1,2,subscribe(C2,3,apply(E1,apply(E2,
    apply(E3,initag( )))))))

The above terms pertain to the agdb level. Now, the representation of Figure 3 applied to d, gives an ser term representing d, namely

e = lk(E1,C1,WORKS,mod(C1,NPOS,2-1,cr(E1,EMP,del(E1,CAND,
    mod(C1,NPOS,2,mod(C1,NPOS,3,cr(E1,CAND,cr(E2,CAND,cr(E3,
    CAND,phi( ))))))))))

which, according to the data model procedural specification, corresponds to the canonical term

A = LK[E1|C1|WORKS|MOD[C1|NPOS|1|MOD[C2|NPOS|3|CR[E1|EMP|
    CR[E2|CAND|CR[E3|CAND|PHI]]]]]]

On the other hand, along the lower path, the canonical term C is, by the backtracking property

```
c = hire(E1,C1,hire(E2,C2,subscribe(C1,2,subscribe(C2,3,apply(E1,apply(E2,
    apply(E3,initag( ))))))))
```

which is represented, according to Figure 3, by the ser term

```
f = lk(E1,C1,WORKS,mod(C1,NPOS,2-1,cr(E1,EMP,del(E1,CAND,lk(E2,C2,WORKS,
    mod(C2,NPOS,3-1,cr(E2,EMP,del(E2,CAND,mod(C1,NPOS,2,
    mod(C2,NPOS,3,cr(E1,CAND,cr(E2,CAND,cr(E3 CAND,phi( )))))))))))))
```

By applying to f the procedure fire(E2,C2,.) of Figure 3, we obtain

```
g = ulk(E2,C2,WORKS,mod(C2,NPOS,3-1+1,cr(E2,CAND,del(E2,EMP,f))))
```

whose corresponding canonical term is

```
B = LK[E1|C1|WORKS|MOD[C1|NPOS|1|MOD[C2|NPOS|3|CR[E1|EMP|CR[E2|CAND|
    CR[E3|CAND|PHI]]]]]]]
```

which is identical to the above A

In general, the representation will be verified to be corrected by proving, for each agdb operation (initag, apply, ..., worksfor) and for every agdb canonical term, the syntactical identity of the corresponding A and B, as above. This can be a laborious task, but the very form of the procedure texts suggests how to break it into cases.

6. TESTING THE REPRESENTATION

In order to provide experimental usage and testing of a specified data type, we developed a SNOBOL-based processor, which is a single program with three identifiable parts:

1. initializations and utilities
2. operations
3. interactive handler for test programs

Part 2 varies according to the data type to be tested.

In our present case, two sets of operations where separately included:

- for running the original specification: operations of the agdb data type (Figure 5);
- for running the representation: operations of the ser data type (Figure 6) and operations of the representation of agdb by ser (Figure 7).

Parts 1 and 3 are given in Appendix A. Part 3 is essentially a loop which keeps prompting the user to submit SNOBOL programs, compiles and runs them. One or more programs within a session can utilize the operations for initializing, updating and querying the data base in its canonical term representation. Terms are traversed using the CURSOR [DAT] feature, analogous to the iterators in [LIS].

Of special interest is the ability to introduce and test operations intended for the external schema level [ANS]. Such operations handle the data base by in turn invoking its (conceptual level) data type operations; they are a simple case of enrichment [GTW] and therefore cannot disturb the data type.

Also due to the characteristics of SNOBOL, it is possible to redefine a data type operation and investigate the overall consequences of the redefinition.

Appendix B contains a sample session, involving various test programs which have been executed over the ser representation. We regard the processor as a short-sized and unsophisticated (as compared to OBJ [GTA], for instance) but useful tool. Since it is entirely contained in this paper and given the wide availability of SNOBOL, it becomes easy for others to reproduce and extend the experiment.

7. CONCLUSION

We have presented a methodology for the crucial task of producing, verifying and testing a precise specification of a data base application. After this becomes available, the way is paved for an orderly sucession of design steps, leading to a final implementation with some data base management system (DBMS)

```
      DEFINE('INITAG()')                                :(INEND)
INITAG      INITAG = 'INITAG'                           :(RETURN)
INEND
*
*
      DEFINE('APPLY(X,S)Z,W,N,S1')                      :(APEND)
APPLY       APPLY = ¬(¬ISCANDIDATE(X,S) ¬WORKSFOR(X,ARG,S))
+                 S                                     :S(RETURN)
            S  'HIRE(' ARG . Z ',' ARG . W ',' ARG . S1 ')' :F(AP1)
               APPLY = 'HIRE(' Z ',' W ',' APPLY(X,S1) ')'  :(RETURN)
AP1         S  'SUBSCRIBE(' ARG . W ',' ARG . N ',' ARG . S1 ')' :F(AP2)
               APPLY = 'SUBSCRIBE(' W ',' N ',' APPLY(X,S1) ')' :(RETURN)
AP2         S  'APPLY(' ARG . Z ',' ARG . S1 ')'    :F(AP3)
               APPLY = LGT(X,Z) 'APPLY(' Z ',' APPLY(X,S1) ')' :S(RETURN)
               APPLY = 'APPLY(' X ',' S ')'             :(RETURN)
AP3         APPLY = 'APPLY(' X ',' S ')'                :(RETURN)
APEND
*
*
      DEFINE('SUBSCRIBE(Y,M,S)Z,W,N,S1')                :(SUEND)
SUBSCRIBE   SUBSCRIBE = EQ(M,0)   S                     :S(RETURN)
            S  'HIRE(' ARG . Z ',' ARG . W ',' ARG . S1 ')' :F(SU1)
               SUBSCRIBE = 'HIRE(' Z ',' W ','
+                          SUBSCRIBE(Y,M,S1) ')'        :(RETURN)
SU1         S  'SUBSCRIBE(' ARG . W ',' ARG . N ',' ARG . S1 ')' :F(SU2)
               SUBSCRIBE = IDENT(Y,W) 'SUBSCRIBE(' Y ','
+                          N + M ',' S1 ')'             :S(RETURN)
               SUBSCRIBE = LGT(Y,W) 'SUBSCRIBE(' W ',' N ','
+                          SUBSCRIBE(Y,M,S1) ')'        :S(RETURN)
               SUBSCRIBE = 'SUBSCRIBE(' Y ',' M ',' S ')' :(RETURN)
SU2         SUBSCRIBE = 'SUBSCRIBE(' Y ',' M ',' S ')'     :(RETURN)
SUEND
*
*
      DEFINE('HIRE(X,Y,S)Z,W,S1')                       :(HIEND)
HIRE        HIRE = ¬(ISCANDIDATE(X,S) HASPOSITIONS(Y,ARG . NUM,S)
+                GT(NUM,0))   S                         :S(RETURN)
            S  'HIRE(' ARG . Z ',' ARG . W ',' ARG . S1 ')' :F(HI1)
               HIRE = LGT(X,Z)  'HIRE(' Z ',' W ','
+                     HIRE(X,Y,S1) ')'                  :S(RETURN)
               HIRE = 'HIRE(' X ',' Y ',' S ')'         :(RETURN)
HI1         HIRE = 'HIRE(' X ',' Y ',' S ')'            :(RETURN)
HIEND
*
*
      DEFINE('FIRE(X,Y,S)Z,W,S1')                       :(FIEND)
FIRE        FIRE = ¬WORKSFOR(X,Y,S)   S                 :S(RETURN)
            S  'HIRE(' ARG . Z ',' ARG . W ',' ARG . S1 ')'
               FIRE = IDENT(X,Z)   S1                   :S(RETURN)
               FIRE = 'HIRE(' Z ',' W ',' FIRE(X,Y,S1) ')'  :(RETURN)
FIEND
*
*
```

```
      DEFINE('ISCANDIDATE(X,S)Z')                          :(ISEND)
ISCANDIDATE  S  ARB 'APPLY(' X $ Z
+                 *(¬WORKSFOR(Z,ARG,S))                    :S(RETURN)F(FRETURN)
ISEND
*
*
      DEFINE('HASPOSITIONS(Y,M,S)W,N')                     :(HAEND)
HASPOSITIONS  S  ARB 'SUBSCRIBE(' Y $ W ',' ARG $ N
+               *COMPARE(((N - NHIRED(W,S))  ',' ),M)  :S(RETURN)F(FRETURN)
HAEND
*
*
      DEFINE('NHIRED(Y,S)S1')                              :(NHEND)
NHIRED      S  ARB 'HIRE(' ARG ',' Y ',' ARG . S1 ')'   :F(NH1)
               NHIRED = NHIRED(Y,S1) + 1                   :(RETURN)
NH1         NHIRED = 0                                     :(RETURN)
NHEND
*
*
      DEFINE('COMPARE(S,P)')                               :(CMEND)
COMPARE     S  P                                           :S(RETURN)F(FRETURN)
CMEND
*
*
      DEFINE('WORKSFOR(X,Y,S)')                            :(WOEND)
WORKSFOR    S  ARB 'HIRE(' X ',' Y                         :S(RETURN)F(FRETURN)
WOEND
```

FIGURE 5

```
         DEFINE('PHI()')                                         :(PEND)
PHI           PHI = '$'                                          :(RETURN)
PEND
*
*
         DEFINE('CR(X,T,S)Y,Z,U,A,I,R,S1')                       :(CREND)
CR            CR = EXS(X,T,S)  S                                 :S(RETURN)
              S  'LK(' ARG . Y ',' ARG . Z ',' ARG . R ','
+                        ARG . S1 ')'                            :F(C1)
                 CR = 'LK(' Y ',' Z ',' R ',' CR(X,T,S1) ')' :(RETURN)
C1            S  'MOD(' ARG . Y ',' ARG . A ',' ARG . I ','
+                        ARG . S1 ')'                            :F(C2)
                 CR = 'MOD(' Y ',' A ',' I ',' CR(X,T,S1) ')' :(RETURN)
C2            S  'CR(' ARG . Y ',' ARG . U ',' ARG . S1 ')' :F(C3)
                 CR = LGT(X T,Y U) 'CR(' Y ',' U ','
+                                            CR(X,T,S1) ')'      :S(RETURN)
                 CR = 'CR(' X ',' T ',' S ')'                    :(RETURN)
C3            CR = 'CR(' X ',' T ',' S ')'                       :(RETURN)
CREND
*
*
         DEFINE('MOD(X,A,I,S)Y,Z,B,J,R,S1')                      :(MEND)
MOD           MOD = ¬(EXS(X,ARG,S) ¬HV(X,A,I,S))  S              :S(RETURN)
              S  'LK(' ARG . Y ',' ARG . Z ',' ARG . R ','
+                        ARG . S1 ')'                            :F(M1)
                  MOD = 'LK(' Y ',' Z ',' R ',' MOD(X,A,I,S1) ')' :(RETURN)
M1            S  'MOD(' ARG . Y ',' ARG . B ',' ARG . J ','
+                        ARG . S1 ')'                            :F(M3)
                  IDENT(X A,Y B)                                 :F(M2)
                   MOD = IDENT(I,'*')  S1                        :S(RETURN)
                   MOD = 'MOD(' X ',' A ',' I ',' S1 ')'         :(RETURN)
M2                MOD = LGT(X A,Y B) 'MOD(' Y ',' B ',' J ','
+                                              MOD(X,A,I,S1) ')' :S(RETURN)
                  MOD = 'MOD(' X ',' A ',' I ',' S ')'           :(RETURN)
M3            MOD = 'MOD(' X ',' A ',' I ',' S ')'               :(RETURN)
MEND
*
*
         DEFINE('LK(X,Y,R,S)Z,W,Q,S1')                           :(LEND)
LK            LK = ¬(EXS(X,ARG,S) EXS(Y,ARG,S) ¬ISR(X,Y,R,S)) S :S(RETURN)
              S  'LK(' ARG . Z ',' ARG . W ',' ARG . Q ','
+                        ARG . S1 ')'                            :F(L1)
                 LK = LGT(X Y R,Z W Q) 'LK(' Z ',' W ',' Q ','
+                                              LK(X,Y,R,S1) ')'  :S(RETURN)
                 LK = 'LK(' X ',' Y ',' R ',' S ')'              :(RETURN)
L1            LK = 'LK(' X ',' Y ',' R ',' S ')'                 :(RETURN)
LEND
```

```
        DEFINE('DEL(X,T,S)Y,Z,U,A,I,R,S1')                          :(DEND)
DEL          DEL = ¬EXS(X,T,S)   S                                  :S(RETURN)
             V1 = T
             EXS(X,ARG $ V2 *DIFFER(V2,V1),S)                       :S(D1)
             DEL = ¬(¬ISR(X,ARG,ARG,S) ¬ISR(ARG,X,ARG,S)
+                  ¬HV(X,ARG,ARG,S))   S                            :S(RETURN)
D1           S   'LK(' ARG . Y ',' ARG . Z ',' ARG . R ','
+                      ARG . S1 ')'                                 :F(D2)
                DEL = 'LK(' Y ',' Z ',' R ',' DEL(X,T,S1) ')' :(RETURN)
D2           S   'MOD(' ARG . Y ',' ARG . A ',' ARG . I ','
+                       ARG . S1 ')'                                :F(D3)
                DEL = 'MOD(' Y ',' A ',' I ',' DEL(X,T,S1) ')'  :(RETURN)
D3           S   'CR(' ARG . Y ',' ARG . U ',' ARG . S1 ')'
                DEL = IDENT(X T,Y U)   S1                           :S(RETURN)
                DEL = 'CR(' Y ',' U ',' DEL(X,T,S1) ')'  :(RETURN)
DEND
*
*
        DEFINE('ULK(X,Y,R,S)Z,W,Q,S1')                              :(UEND)
ULK          ULK = ¬ISR(X,Y,R,S)   S                                :S(RETURN)
             S   'LK(' ARG . Z ',' ARG . W ',' ARG . Q ','
+                      ARG . S1 ')'
                ULK = IDENT(X Y R,Z W Q)   S1                       :S(RETURN)
                ULK = 'LK(' Z ',' W ',' Q ',' ULK(X,Y,R,S1) ')'  :(RETURN)
UEND
*
*
        DEFINE('EXS(X,T,S)')                                        :(EEND)
EXS          S   ARB 'CR(' X ',' T                     :S(RETURN)F(FRETURN)
EEND
*
*
        DEFINE('HV(X,A,I,S)')                                       :(HEND)
HV           S   ARB 'MOD(' X ',' A ',' I              :S(RETURN)F(FRETURN)
HEND
*
*
        DEFINE('ISR(X,Y,R,S)')                                      :(IEND)
ISR          S   ARB 'LK(' X ',' Y ',' R               :S(RETURN)F(FRETURN)
IEND
```

FIGURE 6 S-ER data model operations.

```
      DEFINE('INITAG()')                                     :(INEND)
INITAG      INITAG = PUI()                                   :(RETURN)
INEND
*
*
      DEFINE('APPLY(X,S)')                                   :(APEND)
APPLY       APPLY = ¬(¬ISCANDIDATE(X,S) ¬WORKSFOR(X,ARG,S))
+                 S                                          :S(RETURN)
            APPLY = CR(X,'CAND',S)                           :(RETURN)
APEND
*
*
      DEFINE('SUBSCRIBE(Y,M,S)')                             :(SUEND)
SUBSCRIBE   SUBSCRIBE = EQ(M,0)   S                          :S(RETURN)
            SUBSCRIBE = HV(Y,'NPOS',ARG . N,S)
+            MOD(Y,'NPOS',N + M,S)                           :S(RETURN)
            SUBSCRIBE = MOD(Y,'NPOS',M,CR(Y,'COMP',S))       :(RETURN)
SUEND
*
*
      DEFINE('HIRE(X,Y,S)')                                   :(HIEND)
HIRE        HIRE = ¬(ISCANDIDATE(X,S) HASPOSITIONS(Y,
+                 (ARG . N *GT(N,0)),S))  S                    :S(RETURN)
            HIRE = LK(X,Y,'WORKS',DEL(X,'CAND',CR(X,'EMP',
+                MOD(Y,'NPOS',N - 1,S))))                     :(RETURN)
HIEND
*
*
      DEFINE('FIRE(X,Y,S)')                                  :(FIEND)
FIRE        FIRE = ¬WORKSFOR(X,Y,S)   S                      :S(RETURN)
            HV(Y,'NPOS',ARG . N,S)
            FIRE = ULK(X,Y,'WORKS',DEL(X,'EMP',CR(X,'CAND',
+                MOD(Y,'NPOS'.N + 1.S))))                    :(RETURN)
FIEND
*
*
      DEFINE('ISCANDIDATE(X,S)')                             :(ISCEND)
ISCANDIDATE   HAS(X,'CAND',S)                       :S(RETURN)F(FRETURN)
ISCEND
*
*
      DEFINE('HASPOSITIONS(Y,M,S)')                          :(HAEND)
HASPOSITIONS   HV(Y,'NPOS',M,S)                     :S(RETURN)F(FRETURN)
HAEND
*
*
      DEFINE('WORKSFOR(X,Y,S)')                              :(WOEND)
WORKSFOR    ISR(X,Y,'WORKS',S)                      :S(RETURN)F(FRETURN)
WOEND
```

FIGURE 7

manipulating a file structure convenient in terms of efficiency considerations.

The transition from the purely behavior-oriented specification to one where data structuring comes to the fore is facilitated by the properties of the entity-relationship view, whose translation into the more data-structure oriented models has been for a long time under investigation [CHE]. It is also to be expected that DBMSs directly based on the entity-relationship model, as proposed in [POO], will become available.

REFERENCES

[ANS] ANSI/X3/SPARC - Interim Report of the Study Group on Data Management Systems. FDT Bulletin, ACM, 1975.

[CHE] Chen, P. - The entity-relationship model - toward a unified view of data. ACM-TODS v.1, n.1, 1976.

[DAT] Date, C.J. - An introduction to data base systems. Addison-Wesley, 1977.

[FVE] Furtado, A.L. and Veloso, P.A.S. - Procedural specifications and implementations for abstract data types. SIGPLAN Notices to appear.

[GTA] Goguen, J.A. and Tardo, J.J. - An introduction to OBJ: a language for writing and testing formal algebraic specifications. Proc. Specifications of Reliable Software, IEEE Computer Society, 1979.

[GTW] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. - An initial algebra approach to the specification, correctness and implementation of abstract data types, in R.T. Yeh (ed.) Current trends in programming methodology, vol. IV, Prentice-Hall, 1978, p.81-149.

[GUT] Guttag, J. - Abstract data types and the development of data structures. CACM, vol.20, n.6, 1977.

[HEW] Hewitt, C. - PLANNER: a language for proving theorems in robots. Proc. IJCAI, 1971.

[HOA] Hoare, C.A.R. - Proof of correctness of data representations. Acta Informatica, vol.1, 1972, p.271-281.

[LIS] Liskov, B.H., Snyder, A., Atkinson, R. and Schaffert, C. - Abstraction mechanisms in CLU. CACM, vol.20, n.8, 1977.

[POO] Poonen, G. - CLEAR: a conceptual language for entities and relationships, Proc. of International Conference on Management of Data, 1978.

[SHA] Shaw, M. - Presentation in the workshop on data abstraction, databases and conceptual modelling, SIGMOD Record, vol.11, n.2, 1981, p.43-46.

[ULL] Ullman, J.D. - Principles of database systems. Computer Science Press, 1980.

[VCF] Veloso, P.A.S., Castilho, J.M.V. and Furtado, A.L. - Systematic derivation of complementary specifications for data base applications. Res. Rept. DB038101, PUC/RJ, 1981.

```
*                 APPENDIX A: THE SNOBOL-BASED PROCESSOR
*
*
*                        INITIALIZATIONS AND UTILITIES
*
*
        &FULLSCAN = 1
        &ANCHOR = 1
        INPUT(.INTERM,'INTERM',80)
        OUTPUT(.OUTERM,'OUTERM','(1X,72A1)')
        ARG = BREAK(',(') '(' BAL ')' | BREAK(',)')
*
        DEFINE('CURSOR(C)')                                    :(CEND)
CURSOR        $(C 1) = 0
              $C = ARG @CUR *GT(CUR,CUR1) @($(C 1))    :(RETURN)
CEND
*
        DEFINE('NEXT(C)')                                      :(NXEND)
NEXT          CUR1 = $(C 1)
              NEXT = $C                                        :(RETURN)
NXEND
*
        DEFINE('DISPLAY(VAR)T,V')                              :(DIEND)
DISPLAY       T = ''
DI1           VAR BREAK(',') . V ',' = ''                      :F(DI2)
              T = T V ' = ' $V '  '                            :(DI1)
DI2           OUTERM = T  VAR ' = ' $VAR                       :(RETURN)
DIEND
        OPSYN('!','DISPLAY',1)




*
*                 INTERACTIVE HANDLER FOR TEST PROGRAMS
*
        NPRG = 0
CONTSESS  NPRG = NPRG + 1
STRT    OUTERM = '???'
        TEST = ''
CONTRQ        TEST = TEST TRIM(INTERM)
        TEST ARB 'CANCEL'                                   :S(STRT)
        TEST ARB 'FINIS'                                    :S(ENDSESS)
        TEST ARB 'OLD(' ARG . P ')'                         :F(TSV)
        NPRG = NPRG - 1                                     :<$('PRG' $P)>
TSV     TEST ARB . LD 'SAVE(' ARG . SV ')' = LD             :F(NWRQ)
        $SV = NPRG
NWRQ    TEST ARB . TS '.' RPOS(0) = TS '; :(CONTSESS)'  :F(CONTRQ)
        P = CODE(TEST)                                      :F(CFAIL)
        $('PRG' NPRG) = P                                   :<P>
CFAIL OUTERM = 'COMPILATION FAILED'                         :(STRT)
ENDSESS
*
*
END
```

APPENDIX B: SAMPLE SESSION

```
 ???

"processes a series of updates
recording the corresponding canonical term";

      z = hire('e3','c2',hire('e2','c2',hire('e1','c2',
          subscribe('c2',1,hire('e1','c1',hire('e4','c1',
          apply('e1',hire('e3','c2',apply('e2',apply('e4',
          subscribe('c2',3,apply('e3',subscribe('c1',2,
          initag()))))))))))))) ;
      !'z'.

 Z = LK(E1,C1,WORKS,LK(E2,C2,WORKS,LK(E4,C1,WORKS,
     MOD(C1,NPOS,0,MOD(C2,NPOS,3,CR(C1,COMP,CR(C2,
     COMP,CR(E1,EMP,CR(E2,EMP,CR(E3,CAND,CR(E4,EMP,
     $)))))))))))


 ???

"queries
 1. finds a candidate";

      iscandidate(arg . candidate,z) ; !'candidate'.

 CANDIDATE = E3

 ???

 "2. finds a company with no vacant positions";

      haspositions(arg . company,0,z) ; !'company'.

 COMPANY = C1

 ???

 "3. lists for each company its vacant positions and employees";

        cursor('comp');
contcomp    haspositions(next('comp') . company,arg . vacant,z)
            !'company,vacant'                    :f(emp.comp.end);
            cursor('emp');
contemp            worksfor(next('emp') . employee,company,z)
                   !'employee'                  :s(contemp) f (contcomp);
emp.comp.end.

 COMPANY = C1  VACANT = 0
 EMPLOYEE = E1
 EMPLOYEE = E4
 COMPANY = C2  VACANT = 3
 EMPLOYEE = E2
```

```
   ???

  "defines and uses an external schema operation
  allowing company c2 to hire people who have not applied yet
  provided that at least 10 openings remain";

         define('c2.hire(x,s)')                            :(c2hend);
c2.hire        c2.hire = haspositions('c2',(arg   np *gt(np,10)),s)
                         hire(x,'c2',apply(x,s))  :s(return);
               c2.hire = hire(x,'c2',s)           :(return);
c2hend;
         z = subscribe('c2',17,z);
         z = c2.hire('e5',z);
         worksfor('e5',arg . company,z);
         output = 'e5 works for ' company.

  E5 WORKS FOR C2



  ???

  "tries to hire a person already working for a company and fails";

        save(try);
        z = hire('e4','c2',z);
        status = worksfor('e4','c2',z)  'hired';
        status = ¬worksfor('e4','c2',z)  'not hired';
        output = status.

  NOT HIRED

  ???

  "redefines a conceptual schema operation
  now a person who has applied always remains a candidate";

        define('iscandidate(x,s)')        :(iscend);
iscandidate exs(x,('cand' | 'emp'),s)     :s(return)f(freturn);
iscend.

  ???

  "tries again to hire the already employed person";

        old(try)

  HIRED

  ???

  "terminates the session";

        fini;
```