# Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

153

# Graph-Grammars and Their Application to Computer Science

2nd International Workshop

Edited by
Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg



Springer-Verlag
Berlin Heidelberg New York Tokyo

# SPECIFICATION OF DATA BASES THROUGH REWRITING RULES

A.L. Furtado
P.A.S. Veloso
Pontificia Universidade Católica do R.J.
Brasil

ABSTRACT

Formalisms based on graph transformations are used to specify data base applications. Starting from an informal description, two formal specifications - one query-oriented and the other update-oriented - are successively derived.

## 1. INTRODUCTION

Formal specifications of data bases present some well-know advantages. Unfortunately they also present some problems. Paramount among these are difficulties in constructing and in understanding them, as well as in finding modularization strategies able to cope with the size and complexity of data bases. The use of grammars within a sequence of complementary specifications can help alleviating the above problems. Grammatical formalisms over strings [12] or graphs [3,4] have been proposed for data base specification.

Starting from a verbal description of a data base application, one can visualize each state as a graph representing the real-world facts. With each class of facts we can associate some query operation, to ascertain whether a fact holds at a given state. Next we select some application-oriented update operations, characterizing them by what facts they cause to be asserted or denied (or, equivalently, by their effect on the result of queries). This characterization can be done under the intuition-appealing form of graph-grammar productions.

Then we change our perspective, recognizing that, since each state is obtainable by some sequence of updates, terms consisting of such sequences can represent the states. This leads to the specification of a canonical term algebra $\mathfrak{C}$, where each operation corresponds to a transformation on trees. Now we can program a (hopefully confluent and Noetherian) term-rewriting system [10,14] whose normal forms are exactly the elements of $\mathfrak{C}$.

Section 2 introduces the example to be used to illustrate the methodology, giving first an informal description and then showing a representation of states as

graphs (more exactly, intersecting two-level trees). Section 3 gives the first formal specification, which is query-oriented and uses a graph-grammar formalism. In section 4, an update-oriented specification, under the form of a term-rewriting system, is derived. Section 5 contains the conclusions and references to complementary work.

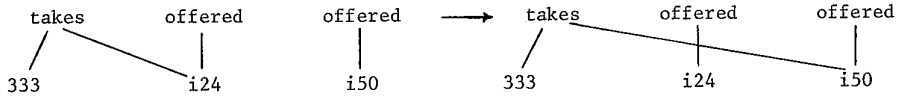
## 2. INFORMAL DESCRIPTION

The contents of a particular data base at some instant of time is called a state. To be more precise, a state is a possibly empty unordered collection of ground positive litterals, here called facts. For specific data base applications only certain kinds of facts are admitted. In our example, referring to an academic world, a fact may be that a course is being offered or that a student is taking a course.

Not all states containing these two kinds of facts will be valid, however. Here we impose the static constraint that students can only be taking currently offered courses.

States as seen at different instants of time can differ. The passage from a state to another is a transition. Valid transitions must involve pairs of states which are both valid (with respect to static constraints), and, in addition, must obey the required transition constraints. In our example, the only transition constraint is that, once a student starts to take some course, the number of courses that he takes cannot drop to zero (in the academic term being recorded). In other words, at any subsequent state he must be taking some course, which in particular may be the same one that he is taking in the current state.

The figure below shows a valid transition between two valid states, using intersecting two-level trees:




One may assert or deny a fact. If a fact is denied it simply vanishes from the data base, since only positive facts are kept. Assert, deny and create (an initial empty state) are primitive operations. Systems where primitive operations can be used directly, at the risk of possibly violating some constraint, are called open systems. As an alternative more apt ro preserve constraints, we shall consider systems with encapsulation, where only certain application-oriented update operations are directly available.

The operations in our example are: initiate academic term, offer course, cancel course, enroll student in course and transfer student from a course to another.

## 3. QUERY-ORIENTED SPECIFICATION

The names of the chosen application-oriented update operations clearly indicate what should be their intended effects. However, in view of the declared constraints, we still must determine the applicability conditions and possible side-effects of the operations. Intended effects, conditions and side-effects are all expressible as the result of queries on facts holding in the original or in the resulting state. In our example we have:

a. initiate acedemic term
   intended effects : create (an empty state)
   conditions : none
   side-effects : none

b. offer course
   intended effects : assert that the course is offered
   conditions : none
   side-effects : none

c. cancel course
   intended effects : deny that the course is offered
   conditions : no student is taking the course
   side-effects : none

d. enroll student in course
   intended effects : assert that the student takes the course
   conditions : the course is being offered
   side-effects : none

e. transfer student from a course to another
   intended effects : deny that the student takes the first course and assert that he takes the other one
   conditions : the other course is being offered
   side-effects : none

Certain assumptions should be made explicit:

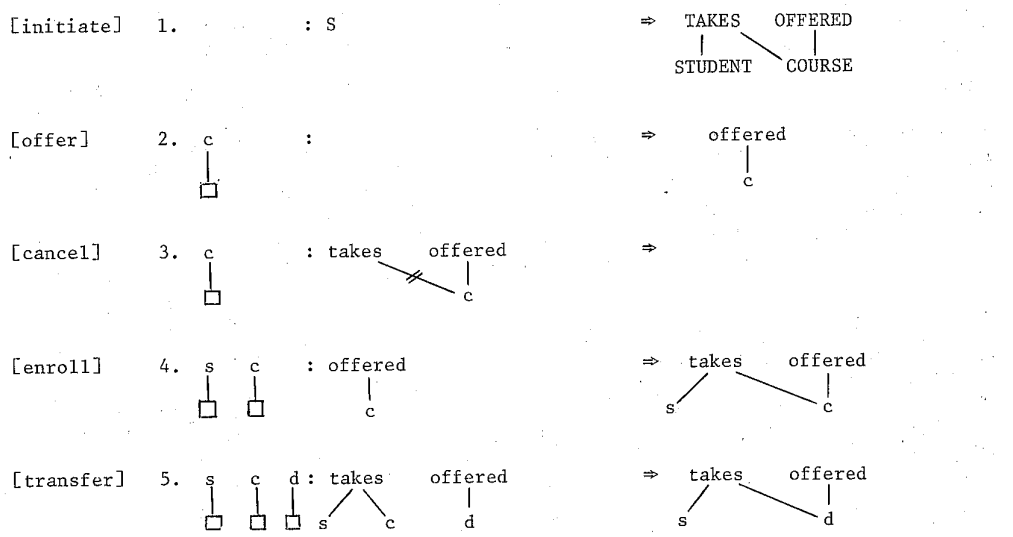
- if the conditions for an operation fail, the current state remains unchanged;

- if the intended effects already hold, the state also remains unaffected (e.g. enroll a student in a course that he is already taking or cancel a course that

is not being offered);

- each fact is represented only once.

We give below the specification of the operations as graph-grammar productions, using the notation explained in [4]. The first production originates the initial empty state, taking the start symbol S into the data base schema; the data base schema is implicitly present on both sides of all the other productions.




It is easy to show that the two constraints will be enforced if the data base is handled only through the above operations. The case of the transition constraint is trivial because none of the operations reduces the number of courses that a student is taking. The static constraint motivated the conditions for applying cancel, enroll and transfer.

Some freedom of choice is given by the possibility of strengthening either conditions or side-effects. For example, we might replace c by:

c'. cancel course
intended effects : deny that the course is offered
conditions : no student is taking the course exclusively
side-effects : deny that any student is taking the course

This example also shows how the enforcement of constraints depends on the interplay of conditions and effects of operations. The condition in c' is motivated

by the transition constraint, which could now be violated by the execution of the side-effects.

## 4. UPDATE-ORIENTED SPECIFICATION

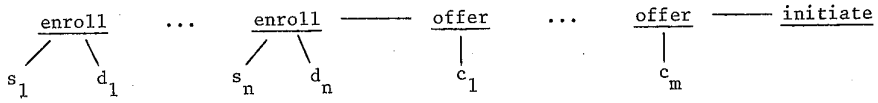
So far we have specified a data base by describing each state by means of its properties expressed by the facts that hold. In section 3 we encapsulated a particular data base application to be handled only through a fixed set of update operations. Then only states reachable by sequences of these operations will be generated and it becomes natural to represent each such state by a sequence of operations creating it.

Each such sequence can be regarded as a trace [1,6]. We can then describe the effects of each operation as a transformation on traces.

### 4.1 - Canonical Terms

It is convenient to choose as representatives a set of terms closed under subterms. Then we shall have canonical terms [7]. Moreover, in order to have a unique term representing each state, only certain terms are to be elected.

In the case of our example of a data base application it is clear that the operations offer and enroll, besides the initialization, suffice to generate all the valid reachable states (these are then the constructor operations [8]). In fact, we can be even more selective: a state where courses $c_1,\ldots,c_m$ are offered and the enrollments consists of the pairs $(s_1,d_1),\ldots,(s_n,d_n)$ can be represented as



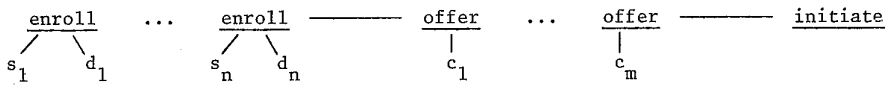

where whenever c occurs in an enroll then it also occurs in some offer.

In order to have uniqueness of representatives we fix a particular ordering (say lexicographic) among courses and demand $c_1 <\ldots< c_m$. Analogously we also require $(s_1,d_1) <\ldots< (s_n,d_n)$. These terms will constitute our canonical representatives.

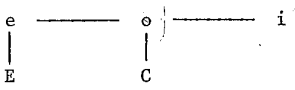
### 4.2 - Transformations

Now that we have a one-to-one correspondence between states and canonical representatives we can specify each application operation by describing the corresponding tree transformations on canonical representatives. In order to describe the

transformations on a term X of the form




as above, we use the notation $C = \{c_1,\ldots,c_m\}$ , $E = \{(s_1,d_1),\ldots,(s_n,d_n)\}$. Notice that both sets can be read off from X. Also, we abbreviate X as

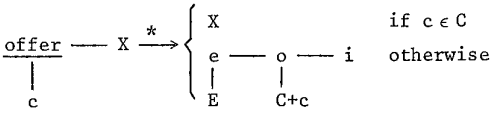



With these considerations we can describe the updates as follows:

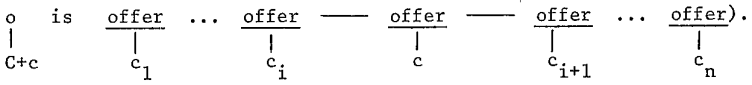
initiate academic term:

$\xrightarrow{*}$ initiate

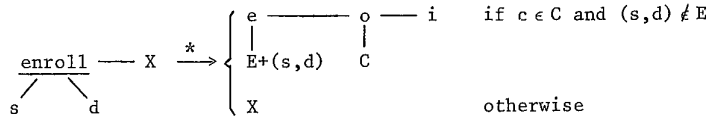
offer c at X:




(Here, o means o with offer inserted in its proper place according to the ordering; e.g. if $c_i < c < c_{i+1}$ then



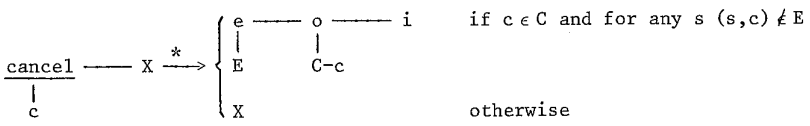

enroll s in d at X:




(Here e is analogous to o )

E+(s,d) C+c

cancel c at t:



(Here $o_{C-c}$ denotes the result of removing $\underline{offer}_{c}$ from $o_{C}$ ; e.g.

if $c = c_j$ then $o_{C-c}$ is $\underline{offer}_{c_1}$ ... $\underline{offer}_{c_{j-1}}$ —— $\underline{offer}_{c_{j+1}}$ ... $\underline{offer}_{c_n}$).

transfer s from c to d at X:

$$\underline{transfer}_{s\ c\ d} \text{ —— } X \xrightarrow{*} \begin{cases} e_{E-(s,c)+(s,d)} \text{ —— } o_{C} \text{ —— } i & \text{if } (s,c) \in E \text{ and } d \in C \\ X & \text{otherwise} \end{cases}$$

(Here $e_{E-(s,c)}$ is the analogous of $o_{C-c}$ )

Similarly the queries can be described:

is c offered at X:

$$\underline{offered}_{c} \text{ —— } X \xrightarrow{*} \begin{cases} \underline{True} & \text{if } c \in C \\ \underline{False} & \text{otherwise} \end{cases}$$

does s take d at X:

$$\underline{takes}_{s\ d} \text{ —— } X \xrightarrow{*} \begin{cases} \underline{True} & \text{if } (s,d) \in E \\ \underline{False} & \text{otherwise} \end{cases}$$

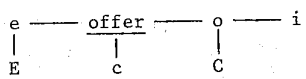
The above transformations can be regarded as the specification of the "input-output behavior" of a rewriting system. Our task now is to produce such a rewriting system.

## 4.3 - Strategy

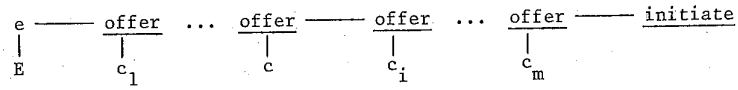
One way to arrive at a rewriting system performing the desired transformations consists in decomposing them into simpler transformations achieving some subgoals.

For instance, consider the transformations of offer c at X. We can:

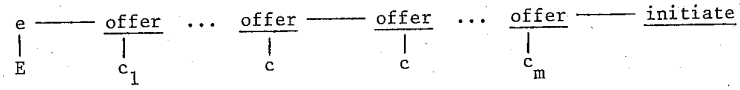
- first, move the initial $\underline{offer}_{c}$ inwards over all the enroll's , obtaining




- Now, we continue moving offer-c inwards over the $c_i$'s while c $\neq$ $c_i$ , until either we reach a $c_i$ with c = $c_i$ or initiate, obtaining:




in which case we stop, or else we reach a $c_i$ with c = $c_i$ , i.e.




in which case we stop after deleting the extra occurrence of offer(c).

These sub-transformations suggest which rules to write. The rules can then be checked to be sound and powerful enough to actually perform the required sub-transformations. A detailed presentation of a complete example appears in [14].

## 4.4 - Rewrite Rules

We can now specify our example data base application by means of a system of rewrite rules that performs the required transformations. This amounts basically to design a (possibly non-deterministic) program performing symbolic manipulations on trees. In fact, we shall present our rewriting system in a procedural notation [15], which consists of rewrite rules with a superimposed order of application (resembling, e.g., programmed grammars [13]).

For each application operation we shall a procedure that performs the required transformation on a canonical representative, passed as parameter. Traces will correspond to syntactically correct sequences of procedure calls, resulting in the corresponding canonical representatives. Procedures for queries should inspect the canonical representative to return True or False.

Each procedure consists of a heading followed by an optional pre-condition testing and then by a match statement. The latter recursively analyses the tree structure of the canonical representative X passed as parameter. The value returned by a particular invocation of the procedure is the value of the right-hand side of the first $\Rightarrow$-statement whose left-hand side happens to be satisfied.

In all procedures we assume the following type declarations:

```
      s , t: student ; c , d , e : course ; X , Y : state.

op initiate academic term!
  ⇒ initiate
endop

op offer c at X!
  match X :
    initiate ⇒ offer ―― initiate
                 |
                 c

    offer ―― Y ⇒ case c * d
      |                c < d : offer ―― X
      d                                |
                                       c

                     c = d : offer ―― Y
                                |
                                d

                     c > d : offer ―― offer c at Y!
                                |
                                d

                   endcase

    enroll ―― Y ⇒ enroll ―― offer c at Y!
     /  \          /  \
    s    d        s    d

  endmatch
endop

op enroll s in c at X!
  ¬ is c offered at X? ⇒ X;
  match X
    enroll ―― Y ⇒ case (s,c) * (t,d)
     /  \                (s,c) < (t,d) : enroll ―― enroll ―― Y
    t    d                                 /  \     /  \
                                          s    c   t    d

                         (s,c) = (s,d) : X

                         (s,c) > (t,d) : enroll ―― enroll s in c at Y!
                                          /  \
                                         t    d

                   endcase

    otherwise ⇒ enroll ―― X
                 /  \
                s    c

  endmatch
endop
```

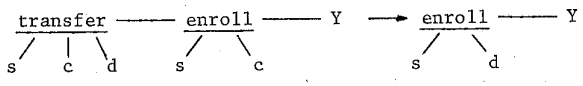
```
op cancel c at X!
  match X
    initiate ⇒ X
    offer ——— Y ⇒ case c * d
      |                 c = d : Y
      d                 c ≠ d : offer ——— cancel c at Y!
                                  |
                                  d
                      endcase
    enroll ——— Y ⇒ case c * d
     /   \                c = d : X
    t     d               c ≠ d : enroll ——— cancel c at X!
                                   /   \
                                  t     d
                      endcase
  endmatch
endop
```

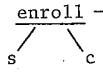
```
op transfer s from c to d at Y!
  match X
    enroll ——— Y ⇒ case (s,c) * (t,e)
     /   \                (s,c) < (t,e) : X
    t     e               (s,c) = (t,e) : enroll s in d at Y!
                          (s,c) > (t,e) : enroll ——— transfer s from c to d at Y!
                                           /   \
                                          t     e
                      endcase
    otherwise ⇒ X
  endmatch
endop
```

```
query is c offered at Y?
  match X
    initiate ⇒ False
    offer ——— Y ⇒ case c * d
      |                 c < d : False
      d                 c = d : True
                        c > d : is c offered at Y?
                      endcase
    enroll ——— Y ⇒ is c offered at Y?
     /   \
    t     d
  endmatch
endquery
```

```
query does s take c at X?
  match X
    initiate ⇒ False
    offer ——— Y ⇒ False
      |
      d
    enroll ——— Y ⇒ case (s,c) * (t,d)
    / \                 (s,c) < (t,d) : False
   t   d                (s,c) = (t,d) : True
                        (s,c) > (t,d) : does s take c at Y?
                  endcase
  endmatch
endquery
```

These procedures taken together form the analogue of a CLU-like cluster [11]. In fact, it can be verified that this cluster will generate exactly the trees corresponding to the canonical representatives. That is the reason why the match-statements contain at most three patterns corresponding to the constructors. It is worthwhile remarking that this fact allows the form of the rules to be simpler. For instance, the procedure for transfer contains the following rewrite rule:




which is not necessarily sound if  is not guaranteed to be canonical.



## 5. CONCLUSIONS AND FURTHER COMPLEMENTARY WORK

We have started from intuition-oriented specifications to obtain specifications where some problems are more amenable to formal treatment. The usage of a graph representation throughout all stages further contributed to making the formal specifications understandable.

Both formalisms - query-oriented and update-oriented - were based on transformations. We might have used instead generative grammatical formalisms, able to either generate or parse (depending on the direction according to which the rules are used) valid instances of states and transitions.

Modularization appears to be connected with some grammatical aspects. Indeed, one can view a module as generated from a nonterminal. Also operations on languages

can be used to combine grammars for diverse modules.

Expressing constraints across separately generated modules, i.e. non-local or "context-sensitive" constraints, becomes a simpler task when the grammatical formalism is based on two-level grammars, which also encompass in a natural way the notion of parameterization [16]. Using W-grammars [2,9,17], we have been able to formalize a number of fundamental data base concepts, including mappings between schemas [5].

Finally, [15] contains an example showing how to obtain an algebraic specification (under the form of conditional axioms) from a rewriting system in procedural notation.

## ACKNOWLEDGEMENT

Financial support from the Conselho Nacional de Desenvolvimento Científico e Tecnológico is gratefully acknowledged.

## REFERENCES

[1] BARTUSSEK,W. and PARNAS,D. "Using traces to write abstract specifications for software modules" ; Technical Report 77-012; University of North Carolina (1977)

[2] CLEAVELAND,J.C. and UZGALIS,R.C "Grammars for programming languages"; Elsevier North-Holland (1977).

[3] EHRIG,H. and KREOWSKI,H.J. "Applications of graph grammar theory to consistency, synchronization and scheduling in data base systems"; Information Systems, vol. 5 (1980) 225-238.

[4] FURTADO,A.L. "Transformations of data base structures"; In 'Graph-Grammars and their Application to Computer Science and Biology'; Claus,V., Ehrig,H. and Rozenberg,G. (eds.); Springer Verlag (1979) 224-236.

[5] FURTADO,A.L. "A W-grammar approach to data bases"; Technical Report 9/82; Pontifícia Universidade Católica do Rio de Janeiro (1982).

[6] FURTADO,A.L and VELOSO,P.A.S. "On multi-level specifications based on traces"; Technical Report 8/81; Pontifícia Universidade Católica do Rio de Janeiro (1981).

[7] 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 'Current Trends in Programming Methodology', Vol. IV, Yeh,R.T. (ed.); Prentice-Hall (1978).

[8] GUTTAG,J. "Abstract data types and the development of data structures"; Comm. of the ACM, 20 (1977) 397-404.

[9] HESSE,W. "A correspondence between W-grammars and formal systems of logic and its application to formal language description"; Technical Report TUM-INFO-7727, Technische Universität München (1977).

[10] HUET,G and OPPEN,D.C. "Equations and rewrite rules: a survey"; Technical Report STAN-CS-80-785, Stanford University (1980).

[11] LISKOV,B. et al. "Abstraction mechanisms inCLU"; Comm. of the ACM, 20 (1977) 564-576.

[12] RIDJANOVIC,D. and BRODIE,M.L. "Defining database dynamics with attribute grammars"; Information Processing Letters, Vol. 14, nº 3 (1982) 132-138.

[13] ROSENKRANTZ,D.J. "Programmed grammars and classes of formal languages"; Journal of the ACM, vol. 16 (1969).

[14] VELOSO,P.A.S. "Methodical specification of abstract data types via rewriting systems"; Technical Report 7/81, Pontifícia Universidade Católica do Rio de Janeiro, (1981).

[15] VELOSO,P.A.S., CASTILHO,J.M.V. and FURTADO,A.L. "Systematic derivation of complementary specifications"; Proc. Seventh International Conference on Very Large Data Bases; (1981) 409-421.

[16] WAGNER,E.G. "Lecture notes on the algebraic specification of data types"; Technical Report RC 9203 (#39787), IBM Thomas J. Watson Research Center (1981).

[17] WIJNGAARDEN,A. van et al (eds.). "Revised report on the algorithmic language ALGOL 68"; Acta Informatica, 5 (1975) 1-236.