PUC

Series: Monografias em Ciência da Computação
Nº 9/82

A W-GRAMMAR APPROACH TO DATA BASES

A. L. Furtado

Departamento de Informática

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

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciência da Computação
Nº 9/82

Editor: Marco A. Casanova

September 1982

A W-GRAMMAR APPROACH TO DATA BASES*

A. L. Furtado

* This work has been sponsored in part by FINEP and by CNPq 402090/80

Abstract

A generative formalism for the characterization of fundamental data base concepts and for the specification of data base applications is proposed. The formalism is based on W-grammars, which are two-level grammars whereby a possibly infinite set of context-free productions can be derived; W-grammars can cope with with non-local constraints as found in data base applications.

The formalism is used to precisely characterize the concepts of state, transition, execution of primitive operations, execution of application-oriented operations and traces. Involved, in turn, in the characterization are facts, static and transition constraints, pre-conditions, transactions and encapsulation.

The use of W-grammars to formalize mappings between schemas is outlined.

Keywords: data bases, generative specifications, W-grammars.

Resumo

Propoe-se um formalismo gerativo para caracterizar conceitos fundamentais em bancos de dados e para a especificacao de aplicacoes de bancos de dados. O formalismo e baseado em gramaticas-W, que sao gramaticas de dois niveis pelas quais se pode derivar um conjunto potencialmente infinito de producoes livres de contexto; gramaticas-W comportam restricoes nao-locais, que sao encontradas em aplicacoes de bancos de dados.

O formalismo e usado para a caracterizacao precisa dos conceitos de estado, transicao, execucao de operacoes primitivas, execucao de operacoes orientadas para aplicacoes e tracos. A caracterizacao envolve fatos, restricoes estaticas e de transicao, pre-condicoes, transacoes e encapsulamento.

O uso de gramaticas-W para formalizar mapeamentos entre esquemas e delineado.

Palavras chaves: bancos de dados, especificacoes gerativas, gramaticas-W.

1. Introduction

In this paper we propose a formalism with the double purpose of giving a precise characterization of certain fundamental concepts in the area of data bases, and of developing a generative methodology [12] for the conceptual design of data base applications.

The need to formalize concepts pertaining to the conceptual schema has been recognized and the subject has received considerable attention recently [6].

Generative data base specifications, using some form of grammar, have been proposed in [3,4,13]. Such formalisms are particularly appropriate to the validation (testing) of the specifications, since the grammars can be used to concretely generate instances of the specified language or to parse what the specifier believes to be legitimate instances. Furthermore, translations between different representations can be done in a convenient way with grammars structured in two levels; informally speaking, one level is used to parse instances of the given representation and the other level to generate the corresponding instances in the new representation.

The formalism to be used here is based on W-grammars [16]. W-grammars are two-level grammars, containing meta-rules and hyper-rules. By applying meta-rules to hyper-rules one can generate a possibly infinite set of production rules. It has been shown that W-grammars can generate/accept general phrase-structured languages.

For our purposes, they offer two features of vital importance:

a. Uniform replacement - Suppose we apply the meta-rule X :: a ; b. (X becomes a or b) to the hyper-rule z : X , p , X. (z becomes X followed by p followed by X). This can originate exactly two production rules:

z : a , p , a.

z : b , p , b.

Uniform replacement means that the meta-notion X must be replaced by the same right-hand side alternative wherever it appears in the target hyper-rule. However, if the hyper-rule were z : X1 , p , X2. (noting that meta-rules like X1 :: X., X2 :: X., ... , Xn :: X. are assumed for every meta-symbol such as X) then X1 and X2 might be replaced independently by a or b, giving origin to the previous productions and, in addition, to:

z : a , p , b.

z : b , p , a.

b. Predicates and derivations ending in blind alleys - Predicates are sets of one or more production rules which eventually produce either the empty symbol (ε) (i.e. contribute nothing to the generated terminal) or a non-terminal string to which no further production rule can be applied, this situation being known as a "blind alley". If we want to modify the example, so that the first and the second occurrences of X are required to be distinct, we may replace the hyper-rule by:

z : X1,p,X2 , unless X1 is X2.

Additional rules not shown here (see [11], for example) will cause the non-terminal unless X1 is X2 to vanish in the cases of apb and bpa, but no rules can be applied to eliminate it in the cases of apa or bpb.

Uniform replacement corresponds very closely to the notion of parameterization [15]. Predicates are a convenient way to handle static or transition constraints involving different elements in the data base (non-local or "context-sensitive" constraints). This favors modular specification, since the various elements can be specified separately and later related via uniform replacement and predicates.

All rules in W-grammars are context-free. The reader should note however that symbols can have any length; unless commas are inserted, a succession of characters including blanks must be understood as a single symbol. In this paper, terminals will be distinguished by being enclosed between quotes, square brackets or angular brackets. The symbols ->, // and <= will also be terminal.

A detailed discussion of W-grammars would take considerable space and hence is outside the scope of the present paper. The reader is referred to [10,11] for an introduction and to [2] for a rather complete treatment.

Our presentation starts with the so-called open systems, which are characterized first in terms of (valid) states and transitions. Primitive operations are added and it is argued that they may lead to non-valid states or transitions. Next, application operations are introduced, associated with the encapsulation strategy, as an effective way to enforce validity. Finally, we consider mappings between schemas [6].

It should be stressed that, even before operations are introduced, formal and precise specifications in terms of data base states and transitions will be obtained. We contend that, unlike the data types arising in the programming languages area, data bases should go through a level of specification based exclusively on what kinds of valid information they will contain and what the valid transformations are. This level does not depend on a particular repertoire of operations, and remains invariant if the repertoire is changed. Constraints on states and transitions are here explicitly declared and are therefore clearly understood and documented, whereas they will be built into the pre-conditions and effects of application

operations at the next level. The ability to handle instances of states and transitions without resorting to operations is again a useful feature of generative formalisms.

The formal methodology involves a succession of W-grammars (see [2,8,9]), where each W-grammar is included in the next one. Rules are divided into general and specific rules, the specific ones belonging to a given data base application.

2. Open systems

We shall characterize open systems by means of three W-grammars, which are designed in an incremental way. The fundamental concepts, corresponding to the start symbol of each grammar, will be state, transition and execution.

A state is a possibly empty unordered collection of ground positive litterals, here called facts. For a particular data base application - referring to an academic world, in our example - only certain kinds of facts are admitted. In the example a fact may be that a course is being offered or that a student is taking a course.

The first W-grammar, G1, starts giving the format imposed on any collection of facts, corresponding to the meta-notion DB. Next, the specific kinds of facts for the example application are enumerated, together with the definition of their domains (student number and course identification). Note that the application facts follow the syntax of general facts but, for the the time being, nothing in the formalism forces this compliance.

```
GRAMMAR G1
 Meta-rules
   general
     DB :: FACTS ; &.
     FACTS :: FACTS , FACT ; FACT.
     FACT :: FACTNAME , ( , VALUES , ).
     FACTNAME :: FACTNAME , ALPHA ; ALPHA.
     VALUES :: VALUES , , VALUE ; VALUE.
     VALUE :: VALUE , SYMBOL ; SYMBOL.
     SYMBOL :: ALPHA ; DIGIT.
     ALPHA :: a ; b ; c ; d ; e ; f ; g ; h ; i ; j ; k ; l ; m ; n;
              o ; p ; q ; r ; s ; t ; u ; v ; w ; x ; y ; z.
     DIGIT :: 0 ; 1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9.
   specific
     APPFACT :: offered , ( , COURSEID , ) ;
                takes , ( , SNUMB , | , COURSEID , ).
     SNUMB :: DIGIT , DIGIT , DIGIT.
     COURSEID :: ALPHA , DIGIT , DIGIT.
(to be continued)
```

We are now in a position to define a state. A state is a DB where:

a. there are only facts pertaining to the chosen application;

b. each fact is valid with respect to the imposed static constraints.

In our example, the only static constraint is that students can only be taking currently offered courses.

Static constraints are checked by selecting some fact and comparing it to the remaining facts in the state; checking continues recursively with the collection of the remaining facts.

The first hyper-rule requires that the general syntax of facts be obeyed and the second one, which can be used only after the first (because the start symbol is state), makes sure that only the application facts are involved; this is how we impose that application facts follow the syntax for general facts. Application facts are converted, one by one, into the terminal notation (by convention, strings between quotes are terminal, as in [11]), and checked by way of the where and unless predicates which use the pre-defined contains predicate (again, see [11]).

```
GRAMMAR G1 (continued)
 Hyper-rules
   general
     state :: DB.
     DB1 APPFACT DB2 : "APPFACT" , DB1 DB2 , where valid APPFACT
       in DB1 DB2.
   specific
     where valid offered(COURSEID) in DB : &.
     where valid takes(SNUMB | COURSEID) in DB : where DB contains
       offered(COURSEID).
```

An instance of a terminal string accepted/generated by G1 is:

"offered(i50)" "takes(333 | i24)" "offered(i24)"

The next W-grammar, G2, includes all meta-rules and hyper-rules of G1, although the first hyper-rule (with the start symbol of G1) will never be used.

The start symbol of G2 is transition, defined as a pair of states. In the second hyper-rule, an arrow (here by convention, a terminal symbol) replaces the word becomes. In addition:

a. the two states must be valid with respect to the static constraints, which will be checked by the meta-rules and hyper-rules inherited from G1;

b. the specified transition constraints must be obeyed.

Here the only transition constraint is that, once a student begins to take some course, the number of courses that he takes cannot drop to zero. 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 (checked by the second specific hyper-rule).

Transition constraints are verified by taking each fact in the current state and checking it against the rest of the transition (i.e. the transition between the remaining facts in the current state and the entire next state). Conversely, each fact in the next state must be checked against the rest of the transition (between the entire current state and the remaining facts in the next state). This checking in two directions is not redundant; recall that states contain only positive facts, and if a fact appears only in the current (or in the next) state, its negation will not appear explicitly in the next (respectively, current) state.

```
GRAMMAR G2
 Meta-rules
   Those of G1
 Hyper-rules
   general
     Those of G1 plus:
     transition : DB1 becomes DB2.
     DB1 becomes DB2 : DB1 , -> , DB2 , where valid DB1 to DB2 ,
       where valid DB2 from DB1.
     where valid DB1 FACT DB2 to DB3 : where valid FACT in DB1 DB2
       to DB3 , where valid DB1 DB2 to DB3.
     where valid DB2 FACT DB3 from DB1 : where valid FACT in DB2 DB3
       from DB1 , where valid DB2 DB3 from DB1.
     where valid & to DB : &.
     where valid & from DB : &.
   specific
     where valid offered(COURSEID) in DB1 to DB2 : &.
     where valid takes(SNUMB | COURSEID1) in DB1 to DB2 :
       where DB2 contains takes(SNUMB | COURSEID2).
     where valid offered(COURSEID) in DB2 from DB1 : &.
     where valid takes(SNUMB | COURSEID) in DB2 from DB1 : &.
```

A valid transition in terminal notation is:

```
"offered(i50)" "takes(333 | i24)" "offered(i24)" ->
"offered(i50)" "offered(i24)" "takes(333 | i50)"
```

One may wonder whether a transition wherein the next state contains a student taking a course that is not offered is a valid transition, since there is no transition constraint excluding this possibility. Indeed it is not a valid transition and it will not be accepted by G2 ; this happens because, by incorporating the static constraints from G1, G2 ensures that valid transitions can only be defined between valid states. So we need only consider as transition constraints those that cannot be checked statically, i.e. by inspecting a single state.

W-grammar G3 has execution as its start symbol. An execution is defined as a state and a sequence of primitive operations applied to the state. The primitive operations are: create an initially empty state, assert a fact (which is added to the collection) or deny a fact (which is removed, since the states will only contain positive facts).

The second hyper-rule puts the sequence between square brackets (yielding a terminal, by convention) and generates a transition from the given state to some new state, making sure, afterwards, that this new state is really the final one determined by applying the sequence of operations.

Each single operation will lead to some intermediate state. We do not require that the intermediate states and transitions be valid. However we do require validity with respect to the initial and final states as well as to the transition between them, which will be checked through the meta-rules and hyper-rules inherited from G2 (which in turn contains those from G1). These requirements correspond to the concept of transactions in the data base literature.

As before, the first hyper-rule from G1 will not be used because the start symbol is now execution. The hyper-rule marked with an asterisk will also not be used and is included in anticipation of yet another grammar to be given in section 3, where the need for the meta-notion TAIL will be justified.

```
GRAMMAR G3
 Meta-rules
   general
     Those of G2 plus:
     OPERATIONS :: OPERATIONS , OPERATION ; OPERATION.
     OPERATION :: create , ( , ) ; assert , ( , FACT , ) ;
       deny , ( , FACT , ).
     TAIL :: halt ; intermediate DB.
   specific
     Those of G2
 Hyper-rules
   general
     Those of G2 plus:
     execution : DB OPERATIONS halt.
     DB1 OPERATIONS TAIL : [OPERATIONS] , DB1 becomes DB2 TAIL ,
       where DB2 achieved from DB1 by OPERATIONS.
     DB1 becomes DB2 halt : DB1 becomes DB2.
 *   DB1 becomes DB2 intermediate DB2 : DB1 becomes DB2.
```

```
where DB3 achieved from DB1 by OPERATION OPERATIONS :
  where DB2 achieved from DB1 by OPERATION , where DB3
  achieved from DB2 by OPERATIONS.
where DB1 FACT DB2 achieved from DB1 DB2 by assert(FACT) :&.
where DB1 DB2 achieved from DB1 FACT DB2 by deny(FACT) :
  unless DB1 DB2 contains FACT.
where & achieved from & by create() : &.
```

An execution in terminal notation is:

```
[deny(takes(333 | i24)) assert(takes(333 | i50))]

"offered(i50)" "takes(333 | i24)" "offered(i24)" ->

"offered(i50)" "offered(i24)" "takes(333 | i50)"
```

3. Systems with encapsulation

The idea of encapsulation comes from abstract data types. Instead of allowing the use of primitive operations, which may result in states or transitions that are not valid, we require that the data base application be handled only through certain previously defined application operations.

The pre-conditions and effects of such operations are used to specify them and must suffice to guarantee that, if the data base is handled exclusively by them, only valid states and transitions will arise.

Application operations are selected considering the anticipated usage of the data base. Suppose, in our example, that courses can be offered or cancelled and that students are allowed to enroll in courses and transfer to other courses but not to drop courses. These considerations suggest what application operations are needed, an operation to obtain the initial empty state being also required.

Pre-conditions and effects are established in view of the constraints. Pre-conditions are logical expressions which must hold in the class of states to which an operation is said to be applicable; they involve some of the facts in such states. The effects of the application operations are expressed as sequences of primitive operations.

From the discussion it follows that, if the pre-conditions and effects of each application operation are correctly specified and only the application operations are ever used:

a. all constraints will be preserved;

b. the sequences of primitive operations corresponding to the effects will be a subset of those (transactions) which lead to the valid terminals accepted by G3;

c. given a sequence of application operations, all intermediate states and transitions will be valid.

W-grammar G4, which has appexecution (execution using application operations) as start symbol, incorporates G3, which would seem to be highly redundant in view of the considerations above. However, notice that checking the constraints is redundant only if the pre-conditions and effects have been correctly designed and the encapsulation strategy is assumed.

The second hyper-rule takes one of the application operations, puts it between angular brackets (yielding a terminal, once again by convention), generates a transition to the state reached by the operation and continues, recursively, considering the other application operations. The hyper-rule ensures that the new intermediate state must be the same state to which the next operation will be applied. Pre-conditions and effects are the object of the where predicate at the end. The second hyper-rule is to be used for the last operation (or if the sequence contains only one operation).

The reader can now turn back to the TAIL meta-notion and to the hyper-rule with an asterisk in G3. This is the only case in our succession of grammars in that a provision in a grammar for a need arising in the next grammar had to be made. We wanted to ensure that each intermediate state is effectively achieved by the sequence of primitive operations corresponding to the application operation involved, which can only be done by the rules from G3.

All meta-rules and hyper-rules from G3 will be used, except the first, again because appexecution is the start symbol of G4. The is predicate is defined in [11]. In contrast, the auxiliary void predicate, which is specific to the present application, is defined in G4 itself.

```
GRAMMAR G4
 Meta-rules
   general
     Those of G3 plus:
     APPOPERATIONS :: APPOPERATIONS , APPOPERATION ;
       APPOPERATION.
   specific
     Those of G3 plus:
     APPOPERATION :: initiate academic term ;
       offer , COURSEID ; cancel , COURSEID ; enroll , SNUMB , in ,
       COURSEID ; transfer , SNUMB , from COURSEID , to , COURSEID.
 Hyper-rules
   general
     Those of G3 plus:
     appexecution : DB APPOPERATIONS.
     DB1 APPOPERATION APPOPERATIONS : <APPOPERATION> ,
       DB1 OPERATIONS intermediate DB2 , // , DB2 APPOPERATIONS ,
       where APPOPERATION applicable to DB1 as OPERATIONS.
     DB APPOPERATION : <APPOPERATION> , DB OPERATIONS halt ,
       where APPOPERATION applicable to DB as OPERATIONS.
   specific
```

```
Those of G3 plus:
where initiate academic year applicable to & as create() : &.
where offer COURSEID applicable to DB as
  assert(offered(COURSEID)) : unless DB contains
  offered(COURSEID).
where cancel(COURSEID) applicable to DB as
  deny(offered(COURSEID)) : where DB contains
  offered(COURSEID) , where void COURSEID in DB.
where enroll SNUMB in COURSEID applicable to DB as
  assert(takes(SNUMB | COURSEID)) : unless DB contains
  takes(SNUMB | COURSEID) , where DB contains
  offered(COURSEID).
where transfer SNUMB from COURSEID1 to COURSEID2
  applicable to DB as
  deny(takes(SNUMB | COURSEID1))
  assert(takes(SNUMB | COURSEID2)) : where DB contains
  takes(SNUMB | COURSEID1) , unless DB contains
  takes(SNUMB | COURSEID2) , where DB contains
  offered(COURSEID2).
where void COURSEID1 in takes(SNUMB | COURSEID2) DB :
  unless COURSEID2 is COURSEID1 , where void COURSEID1 in DB.
where void COURSEID1 in offered(COURSEID2) DB :
  where void COURSEID1 in DB.
where void COURSEID in & : &.
```

The cancel application operation illustrates an important issue in passing from static and transition constraints to pre-conditions and effects: several choices are possible. Recall that courses can only be taken if they are being offered (static constraint); consequently we cannot cancel a course if any student is taking it, and we decided to impose this as a pre-condition to the cancel operation. We might prefer, instead, to always permit a course to be cancelled, expanding the effects part to remove (deny) all enrollments in the course. Also recall that, because of our transition constraint, students who are taking only the cancelled course must be transferred to some other course rather than simply having their enrollment undone.

In our example, we have imposed the pre-condition that the desired effect of each operation should not already be present in the current state. For instance, we prescribed that enroll is not applicable if the given student is already taking the given course. This is another choice left to the designer, who might prefer to make the exclusion of "vacuous" operations a general requirement, rather than leaving the decision to be made for each application as we did here.

An appexecution in terminal notation is:

<transfer 333 from i24 to i50>

[deny(takes(333 | i24)) assert(takes(333 | i50))]

"offered(i50)" "takes(333 | i24)" "offered(i24)" ->

"offered(i50)" "offered(i24)" "takes(333 | i50)" //

<cancel(i24)> [deny(offered(i24))]

"offered(i50)" "offered(i24)" "takes(333 | i50)" ->

"offered(i50)" "takes(333 | i50)"

Another W-grammar, G5, is of interest for our discussion of encapsulation as in the theory of abstract data types. As before it includes the previous grammars and, in addition, has the following general hyper-rule:

trace : & APPOPERATIONS.

Traces [1,5] denote those states that are reachable from the initial empty state through some sequence of application operations. We usually do our specifications in such a way that all valid states are reachable. We may or may not require that all valid transitions be realized by some sequence of application operations. Notice in our example that there is no operation for students to drop courses, even though this would be legal (except when the course is the only one that the student is taking).

Given two states s1 and s2, we can say that s1 can be transformed into s2 if and only if there is a trace of s1 contained in some trace of s2.

4. Mappings between schemas

Grammars in general have been used for parsing, generating and translating. Mappings between schemas, as in the ANSI/X3/SPARC architecture [6], are essentially translations between the schemas involved.

Thus far the discussion has been confined to the conceptual schema, describing the entire data base. For each user there may be an external schema, where only part of the information may be available, perhaps re-organized according to the different needs of users.

Here we shall show the mapping from conceptual states to the external states of students (more precisely, this will be a family of mappings, there being one mapping for each student). We assume that a student is authorized to see what courses are offered and what is his particular study-list (the set of courses that he is taking), but cannot see what courses the other students are taking. The mapping is described by W-grammar G6. To make sure that external states can correspond only to valid conceptual states, the right-hand side of the

first hyper-rule also yields a "copy" of the the original conceptual state to be reduced to terminal notation by the rules inherited from G1.

```
GRAMMAR G6
   Meta-rules
     general
       Those of G1
     specific
       Those of G1 plus:
       COURSES :: COURSES , COURSEID ; &.
   Hyper-rules
     general
       Those of G1
     specific
       Those of G1 plus:
       mapping to student SNUMB schema :
         student SNUMB DB , <= , DB.
       student SNUMB COURSES DB1 takes(SNUMB | COURSEID) DB2 :
         student SNUMB COURSES COURSEID DB1 DB2.
       student SNUMB COURSES DB : "study-list({COURSES})" ,
         student SNUMB DB.
       student SNUMB1 DB1 takes(SNUMB2 | COURSEID) DB2 :
         student SNUMB1 DB1 DB2 unless SNUMB2 is SNUMB1.
       student SNUMB DB1 offered(COURSEID) DB2 :
         "offered(COURSEID)" , student SNUMB DB1 DB2.
       student SNUMB : &.
```

A mapping in terminal notation, showing the external state of student 333 (on the left-hand side of the arrow) obtained from a valid conceptual state (on the right-hand side), is:

"study-list({i24 i50})" "offered(i50)" "offered(i24)"

<=

"offered(i50)" "takes(333 | i24)" "offered(i24)"

"takes(297 | i50)" "takes(333 | i50)"

5. Conclusion

Generative specifications, like equational or axiomatic specifications, can be used in the formal verification of the intended properties. For instance, if the language generated is empty the specification is inconsistent. Yet their ability to parse or generate instances makes them particularly adequate for experimentation and testing. Among the aspects that can be checked are validity of states and transitions, ability of operations (primitive or application-oriented) to achieve transitions, satisfaction of pre-conditions for application operations (at given states) to be executed as some sequence of primitive operations, sufficiency of pre-conditions and effects of application operations to enforce the static and transition constraints, reachability of valid states from the initial

empty state via sequences of application operations and transference from conceptual to external states of the authorized information.

Intuition is favored by the incremental style used in introducing the W-grammars, especially in the systematic way whereby a specification based on application operations was derived, starting from a purely declarative specification based on states and transitions.

By distinguishing specific rules from general rules, we can correspondingly reduce and discipline the task of the designer of particular data base applications, whose concern will be to supply appropriate specific rules.

The natural characterization of mappings is another contribution of the formalism. Looking at mappings as translations, suggests an analogy with the idea of implementing a data type by another [7]. A second useful analogy, taken from programming languages, can be established between the occurrence of meta-notions in hyper-rules and pattern-matching. The reader familiar with SNOBOL will notice that the little W-grammar, shown in the introduction, induces the two SNOBOL statements below, where the first (a pattern definition) corresponds to a meta-rule and the second (a pattern-matching statement, with S containing the string to be examined) to a hyper-rule:

```
X = 'A' | 'B'
S    X $ V    'P'      V
```

noticing that the "immediate assignment" to variable V and its sbsequent usage corresponds to uniform replacement. It may be interesting to pursue with the comparison, showing how we might require in SNOBOL that the second occurrence of X should be different from the first one:

```
S     X $ V    'P'     X $ W     *DIFFER(V,W)
```

remarking that DIFFER is a predicate returning a null string (like &, in W-grammars) in case the result is true, and otherwise causing the entire pattern-matching to fail (blind alleys, in W-grammars).

As shown in [14], other complementary specifications can be systematically derived once a specification using pre-conditions and effects of application operations has been obtained (as done here). The benefits of complementary specifications (definitional, denotational, operational) are argued in the same reference.

Further research might consider W-grammars as the basis for a software tool for semi-automatic data base specification.

Acknowledgements

The author is grateful to P. A. S. Veloso and M. A. Casanova for useful discussions and suggestions and to M. L. Brodie for reading the manuscript and for helpful comments.

References

[1] Bartussek, W. and Parhas, 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 (1979) 224-236.

[5] Furtado, A. L. and Veloso, P. A. S. - "On multi-level specifications based on traces" - technical report 8/81 - Pontificia Universidade Catolica do R. J. (1981).

[6] Griethuysen (ed.) - Concepts and terminology for the conceptual schema and the information base" - report from the ISO TC97/SC5/WG3 group (1982).

[7] Guttag, J. V., Horowitz, E. and Musser, D. R. - "The design of data type specifications" - in "Current trends in programming methodologies" - Yeh, R. (ed.) - vol. IV - Prentice-Hall (1978).

[8] 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 Universitaet Muenchen (1977).

[9] Hesse, W. - "Two-level graph grammars" - in "Graph-grammars and their application to computer science and biology" - Claus, V., Ehrig, H. and Rozenberg, G. (eds.) - Springer (1979) 255-269.

[10] Koster, C. H. A. - "Two-level grammars" - in "Compiler construction" - Bauer, F. L. and Eickel, J. (eds.) - Springer (1974) 146-156.

[11] Peck, J. E. L. - "Two-level grammars in action" - in "Information processing 74" - Rosenfeld, J. L. (ed.) - North-Holland (1974) 317-321.

[12] Pereda B., A., A. - "Description methods for data types and data structures" (in Portuguese) - D. Sc. dissertation - Pontificia Universidade Catolica do R. J. (1979).

[13] Ridjanovic, D. and Brodie, M. L. - "Defining database dynamics with attribute grammars" - Information Processing Letters, vol. 14, n. 3 (1982) 132-138.

[14] 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.

[15] Wagner, E. G. - "Lecture notes on the algebraic specification of data types" - technical report RC 9203 (#39787), IBM Thomas J. Watson Research Center (1981).

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