PUC

Series : Monografias em Ciência da Computaçao

No. 1/84

FORMAL DATA BASE SPECIFICATION - AN ECLECTIC PERSPECTIVE

P. A. S. Veloso, M. A. Casanova and 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

Series : Monografias em Ciência da Computaçao

No. 1/84

Series Editor: Antonio Luz Furtado January 1984

FORMAL DATA BASE SPECIFICATION - AN ECLECTIC PERSPECTIVE*

P. A. S. Veloso, M. A. Casanova +, A. L. Furtado

* Research partly sponsored by FINEP and CNPq

+ Centro Científico de Brasilia - IBM do Brasil

ABSTRACT

Logical, algebraic, programming language, grammatical and denotational formalisms are investigated with respect to their applicability to formal data base specification. On applying each formalism for the purpose that originally motivated its proposal, it is shown that they all have a fundamental and well integrated role to play in different parts of the specification process. An example is included to illustrate the methodological aspects.

KEYWORDS

data base specification, integrity constraints, logic, algebra, abstract data types, programming languages, grammars, denotational methods.

RESUMO

Formalismos de lógica, álgebra, linguagens de programaçao, gramaticais e denotacionais sao investigados quanto a sua aplicabilidade para especificaçao formal de bancos de dados. Ao aplicar cada formalismo com o fim que originalmente motivou sua proposiçao, mostra-se que todos eles tem um papel fundamental e bem integrado a desempenhar em diferentes partes do processo de especificaçao. Um exemplo è dado para ilustrar os aspectos metodológicos.

PALAVRAS CHAVE

especificaçao de bancos de dados, restriçoes de integridade, lógica, álgebra, tipos abstratos de dados, linguagens de programaçao, gramáticas, métodos denotacionais.

1. Introduction

Different groups of researchers have been using different formalisms in connection with data bases. Here we shall consider the use of formalisms with the following primary purpose:

To specify data base applications subjected to integrity constraints

Initially, we would like to concentrate on the characteristics of the data base application being specified, with no concern for its eventual computer implementation. Such concern would be brought to the foreground after those characteristics are well understood. To guide the phase of implementation-oriented specification, the auxiliary purpose below must also be served:

To specify features of data models, to be used for adapting data base applications to computing environments

What aspects should be covered in this double-purpose specification process? A data base application is first of all a repository of time-varying information. Secondly, there must be functions whereby the information will be used, i.e. interrogated or changed. Thirdly, as we move towards an implementation, we must provide a representation for the data base according to some chosen data model, which involves expressing how the information will be structured and the functions programmed. In turn, the data model specification has syntactical and semantical aspects.

Although data base theory has been largely influenced by concepts derived from first-order logic, either in pure form or adapted to the particular needs of data base research, there have been many attempts to use algebra, high-level programming language constructs, grammars and denotational semantics to capture data base concepts. One might claim that each formalism is powerful enough to cover many (or perhaps all) aspects listed. However it seems more reasonable to conjecture that a single formalism will probably not be equally convenient for all aspects. This position, which we also take, has led to the notion of complementarity.

The major contribution of the paper lies in selecting the correct variation of each formalism for each level of specification, in the style of organizing the formalisms together into a coherent conceptual design framework and in the formal notion of refinement binding the different levels. Thus, contrarily to most published literature, we neither limit ourselves to just one formalism at just one level nor force the use of the same formalism at different levels, which often creates distortions. Finally, although the paper is not intended to be a survey of the area, it may serve as a guide to different approaches to data base theory.

We divide the design process into three levels of specification. Before embarking on their rigorous characterization, we explore

them informally in section 2, where the overly simplified example to be used as illustration is also introduced. In the next paragraphs we briefly say what are these levels of specification, as we indicate how the rest of the paper is organized.

The first level, the information level, characterizes the data base by its information contents independently of how the information will be used or represented. It gives a high-level description of the set of consistent data base states and the set of state transitions and typically involves a language to talk about the data base and a set of static constraints indicating which states are considered consistent, and a set of transition constraints indicating in turn which transitions are acceptable. In this paper, we will adopt an extension of first-order languages, as described in section 3.

At the second level, the functions level, we add to the characterization of a data base a repertoire of functions, establishing how we intend to use the information. These functions indicate how the data base will be queried or updated and depend on the applications the designer anticipates for the data base. We will use in this paper an algebraic formalism related to abstract data types, which is described in section 4.

The third and final level, the representation level, specifies the data base with the help of a data model. A representation of the data base in terms of the data structures supported by the data model must be found and the functions defined at the second level must be mapped into procedures using a Data Manipulation Language (DML) associated with the model. The third level therefore brings us close to the implementation of the data base application on top of a Data Base Management System (DBMS). A programming language, described in section 5, will be used to specify the data base at the third level. The syntax of the language is given by a grammatical formalism, W-grammars, and its semantics is described using a denotational formalism.

Each level of specification must be a refinement of the previous one, in the sense that the second-level update functions must preserve the first-level static and transition constraints, and the third-level procedures defining second-level functions must satisfy the second-level equations. This is further discussed in sections 4.3 and 5.3.

Section 6 contains the conclusions. The bibliography gives a sample of the publications for each formalism, including both the fundamentals and data base research; also included are papers on complementary specifications.

2. An informal outline

2.1. Information level

At the first level, we consider that data bases will contain instances of facts, defined as positive assertions about the application area. Usually, negative facts are not stored (such as what courses are not taken by a student). A state is the collection of facts that are true at a given instant of time; therefore, a state denotes the entire contents of the data base at that instant. Static constraints are restrictions defined on states. A valid state is one that conforms to all specified static constraints.

A transition is a state transformation. A transition can be denoted by a pair of states. Again we may want to impose restrictions on transitions. Thus a valid transition, besides being required to involve only valid states, must conform to the declared transition constraints.

We now begin to present the simple academic data base example to be used throughout the discussion. Using the terminology of the first level, we have:

FACTS: - courses are offered
- students take courses

STATIC CONSTRAINT: students can only be taking currently offered courses

EXAMPLE OF A VALID STATE: c1 is offered
c2 is offered
John takes c1

TRANSITION CONSTRAINT: the number of courses taken by a student cannot drop to zero (during the academic term)

EXAMPLE OF A VALID TRANSITION:

c1 is offered	⟶	c1 is offered
c2 is offered		c2 is offered
John takes c1		John takes c2

Certain points not explicit in our unrealistically simple example must now be stressed. The first point is time, implicitly involved in the transition constraint, which applies only during the current academic term. Time appears in many ways: simply as a criterion to order states, as a duration, as a date, etc. Next we observe that the size and complexity of realistic data bases make their direct specification in one piece an impracticable task. Modularization is in order. Also, we must have ways to verify properties of specifications, such as consistency, non-redundancy, etc., both within and across the various modules.

In an "intelligent" data base, one should be able to infer

certain facts, which then would not have to be stored. For example, if we are confident that the current state of our academic data base is valid and that John takes c1, we could deduce that c1 is offered, taking our static constraint as a general law. Inference becomes more complex when states are allowed to include alternative facts such as John takes c1 or c2, or facts with indeterminate values, such as John takes some course, whose name is presently unknown (an undefined value).

2.2. Functions level

We now turn to the second level, where functions are introduced. To each fact there correspods a query function to check whether or not the fact holds; update functions provide the means to effect transitions, taking states into states. To indicate how a new state is obtained from the current one, we shall assert the facts that become true and deny those that cease to be true. Because of the integrity constraints, the application of certain functions becomes dependent on pre-conditions.

The choice of the set of functions is dictated by the needs of the specific data base application that one can anticipate. In our example the query functions will be:

is course being offered?
is student taking course?

with the obvious meaning. The update functions will be:

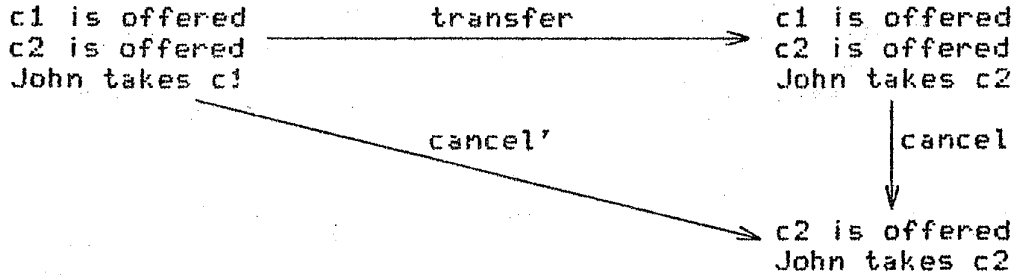
initiate academic term: the data base is "empty" (all facts false)
offer course: assert that the course is offered
cancel course: if no student takes the course
then deny that the course is offered
enroll student in course: if the course is offered
then assert that the student takes the course
transfer student from course1 to course2:
if the student takes course1
and does not take course2
and course2 is offered
then deny that the student takes course1
and assert that he takes course2

Note that here no formal meaning is attached to words like "assert", "deny", "if ... then"; they are used only to favor more concise and structured natural language descriptions.

When we associate application-oriented functions with actions happening in the real world, we are assuming that reality is changed only when the corresponding functions succeed in updating the data base in the intended way. In other words, the segment of interest of the real world is indistinguishable from the data base. It becomes physically impossible to perform an action that

violates some policy of the organization, expressed as a constraint. Also, we can automate certain actions, achieving the so-called active systems, where actions can be triggered by the occurrence of events (possibly involving stored facts, time, etc.).

There may be more than one way to design functions that effectively preserve the declared constraints, which implies that more than one second level specification would be compatible with (and therefore not entirely determined by) the first level specification. Part of this freedom of choice comes from the existence of different ways to combine pre-conditions and effects and also to decide between actions initiated by users and triggered actions. To discuss these possibilities we shall employ the example below.



With the present definition of the functions we could not execute cancel c1 at the state where John takes this course, whereas the use of cancel is legal at the state reached by transferring John to c2. This suggests that, in order to make cancel applicable at the first state shown, we might redefine cancel (see cancel' above) by expanding its effects, thereby being able to weaken its pre-conditions.

Besides redefining a function we may want to create additional ones. For instance, we might add an operation allowing a student to drop a course, if it is not the only one that he is currently taking. Finally, if we had both drop and transfer functions, we could achieve the modified effect of cancel' indicated above without redefining the function: we would merely add a trigger causing either drop or transfer to be invoked for each student taking c1.

The order of execution of operations, on the other hand, is not entirely free because of the interplay of pre-conditions and effects. In our example, enroll John in c1 can only be executed after offer c1 has been executed. Functions whose effects are necessary to fulfill pre-conditions of other functions entail serial execution. On the contrary, offer c1 and offer c2, for instance, can be executed in parallel.

Also from the study of the interplay of pre-conditions and effects, one can conclude that different sequences of functions can accomplish the same net result. Calling a trace any sequence of update functions starting at the initial empty state, the two

traces:

offer c1 . offer c2 . enroll John in c1 .
transfer John from c1 to c2 . cancel c1

and

offer c2 . enroll John in c2

lead to the same state and are in this sense equivalent. Traces clearly provide an alternative way to denote the states that they generate.

2.3. Representation level

To run a data base application on a machine we must adapt the data base application to the machine environment. Such environments involve file structures and facilities to declare, access and manipulate them, usually constituting a Data Base Management System (DBMS).

This adaptation can be specified independently from actual machine details if we employ the abstract structure provided by some data model. The current data models offer trees, graphs, tables, etc. as abstract structures. To each data model there corresponds a family of DBMSs.

Using the relational model, we could represent the information in our academic data base by way of two tables, OFFERED and TAKES, as shown below:

OFFERED

C#
c1
c2
..

TAKES

S#	C#
John	c1
...	..
...	..

Executing a query function will now correspond to inspecting tuples of the appropriate tables whereas, for an update function, certain tuples will be inserted, deleted or modified. For instance, transferring John from c1 to c2 involves (after an inspection that would show the presence of tuple (c1) in OFFERED, the absence of (John,c2) from TAKES and the presence of (John,c1) in TAKES) the deletion of (John,c1) from and the insertion of (John,c2) in TAKES.

One would expect a DBMS to handle constraints in at least one of the following ways:

a. The constraint may be implicit in the data model. An example is that relational tables, by definition, do not admit duplicate tuples.
b. The constraint may be declared. For example, we may declare that certain columns of a table constitute a key, in which case no two tuples with the same values in such columns are

permitted.

c. The constraint may be enforced procedurally. This can be accomplished, among other ways, by restricting tuple-update operations to be invoked only from within procedures corresponding to the previously defined update functions. An example is the description above of how transfer would be executed on the OFFERED and TAKES tables.

Since strategy (a) is not enough and most DBMSs are weak in terms of features to enable strategy (b), strategy (c), known as encapsulation, becomes an important option and is assumed here. A characteristic of encapsulation is that it does not prevent that the execution of tuple-update operations violate constraints; however this occurs only at intermediate states. For example, when transferring John from c1 to c2, as we first delete (John,c1) from TAKES the transition constraint is violated, but the violation is immediately corrected by the ensuing insertion of (John,c2).

Similarly to the relationship between levels one and two, there can be more than one third level specification compatible with a second level one, in the sense that all functions are correctly realized. Here the freedom results from the possibility of choosing among different data models and also from the various ways whereby a data base application can be structured within the same data model.

2.4. Placing the formalisms

We have investigated, in some detail, the nature of what we want to specify formally. In the introduction, five formalisms were mentioned. We now propose that each formalism be used essentially for the purpose that originally motivated its conception:

- with respect to data base applications
  - for the information level - logical formalism
  - for the functions level - algebraic formalism
  - for the representation level - programming language formalism

The basic formalism is logic. For a long time, first-order logic has been regarded as a paradigm of formalization.

The first level of specification characterizes data bases by their information contents, independently of how the information will be used and also independently of representation. We did not see any need to depart from logic at this level, where types of facts will quite naturally correspond to predicate symbols and integrity constraints to axioms. Yet, we note that considerations, such as the presence of components of different types and transition constraints, may prescribe the use of many-sorted first-order logic and temporal logic, rather than strict first-order logic.

At the second level, we add to the characterization of a data base a repertoire of functions, establishing how we intend to use

the information. Representation considerations are still absent. Algebraic formalisms have been conceived precisely to specify objects by the collection of functions defined on them. Functions will be defined via conditional equations expressing their input/output properties.

At the third level we introduce representation, following a data model, and thus pave the way to an eventual computer implementation. When indicating what kinds of information the data base will contain (at the first level) no thought is given to how it could be structured for efficient access. Similarly, the functions defined (at the second level) take entire data base states both as domain and range, whereas, if the information were organized, inspections and changes could be circumscribed to small parts of the data base. The formalisms that more directly express computing phenomena are programming languages. With the commands of a programming language, functions can be programmed as procedures, acting on data structures containing the information. (see also [We2], page 158, for three-level specifications).

A programming language to be employed for formal specification must, of course, be a very high-level and theoretically sound language, which excludes most languages coming with the currently available DBMSs. Moreover, it must itself be formally specified, and thus we have:

- with respect to the language associated with the data model
  . for the syntactical aspects - grammatical formalism
  . for the semantical aspects - denotational formalism

Grammatical formalisms have indeed been created for specifying syntax. They describe it by way of production rules. Certain "context-sensitive" syntactical aspects have been misleadingly labelled as belonging to semantics, simply because of the inability of formalisms, such as BNF, to cope with them. Here, we shall use two-level grammars, which have enough power, for instance, to exclude syntactically commands that manipulate undeclared data structures.

Denotational formalisms purport to explain in mathematical terms the semantics of computer-oriented constructs. They sometimes also cover "abstract syntax", resorting to BNF productions. Since a fully comprehensive treatment of syntax can be provided as indicated in the previous paragraph, we can concentrate on the semantical aspects, to be described mainly by semantic equations.

We should pass from one level to the next in a constructive manner, and should also be able to verify, afterwards, that we have done so in a correct way, i.e., among other requirements:

a. that the second-level functions preserve the first-level integrity contraints;

b. that the third-level procedures realize the second-level

functions.

The second requirement implies that a formal specification of the language used at the third level should be available beforehand.

3. The information level - the use of logical formalisms

3.1. Logical formalisms

In this section we briefly indicate how a data base can be specified, at the information level, using a logical formalism. We assume familiarity with first-order logic at the level, say, of [En], so that the presentation of the formalism will be very terse.

To set up basic terminology and notation, we recall that the alphabet of a first-order language consists of: (1) a set of logical symbols, which are the variables, the usual connectives and quantifiers, the equality symbol (if necessary), and parenthesis; (2) a set of non-logical symbols, which are a set of constant symbols and, for each positive integer n, a set of n-ary predicate symbols and a set of n-ary function symbols. Terms are built from variables, constants and function symbols using the familiar formation rules. Well-formed formulas (wffs) are constructed out of terms, predicate symbols (including equality) and connectives and quantifiers, again using the familiar formation rules. A literal is either an atomic wff or a negated atomic wff. A ground term is a term with no variables and, likewise, a ground wff is a wff with no variables.

A structure I for a first-order language L consists of a non-empty set D, the domain of I, and assigns to each constant c of L an element of D, to each n-ary function symbol f of L an n-ary function I(f) over D, and to each n-ary predicate symbol p of L an n-ary relation over D. A valuation of L for I is a function v assigning to each variable of L an element of D. We use !=I P[v] to indicate that I satisfies P with v, and use !=I P to indicate that I satisfies P with any valuation v of L for I [En]. In the last case, we say that P is valid in I. A model of a set W of wffs of L is a structure I of L such that all wffs in W are valid in I. We say that W logically implies a wff w iff w is valid in any model of W.

The first-order predicate calculus defined on a first-order language consists of a set of axiom schemes, the logical axioms, and two inference rules: modus ponens and generalization.

A first-order axiomatic theory is a pair T=(L,A), where L is a first-order language and A is a set of wffs of L, called the non-logical or proper axioms of the theory. A model of the theory is an interpretation of L in which all axioms are valid.

To conclude this brief refresher about first-order languages, we say that the language is many-sorted when each variable and constant is assigned to a sort, each n-ary predicate symbol p is associated with an n-tuple of sorts <T1,...,Tn> (Ti is the sort of the i-th argument of p), and each n-ary function symbol is associated with an (n+1)-tuple of sorts <T1,...,Tn+1> (Ti indicates the sort of the i-th argument, 0<i<n+1, and Tn+1, called the target sort, indicates the sort of the result).

Moreover, the formation rules of terms and wffs are changed so that sorts are respected (for the details see [En]).

At the information level, a data base can in principle be adequately represented as a first-order theory T1=(L1,A1), where L1 is a first-order language used to talk about the data base and A1 is a set of non-logical axioms essentially defining the set of consistent data base states.

A very rich vein of research during the past years has centered around special classes of first-order sentences that capture important facts about data bases and yet have special properties not shared by the full version of first-order logic. The various classes of data dependencies offer the best example.

In another direction, variations of first-order logic have been used to express data base concepts that cannot be readily expressed by ordinary first-order languages. Aggregation operations, such as SUM and AVERAGE that map relations into scalar objects, is one example. They require a special treatment to avoid talking about higher-order functions of higher-order logics [CB].

Another example is precisely the notion of transition constraints that impose restrictions on data base state transitions and not just on data base states. Most of the research, with a few notable exceptions, ignored this type of constraints, although they are equally important and interesting. We now explain a possible extension of first-order languages to cover aspects related to transitions. The extension we describe is perhaps the simplest one and depends on the introduction of two modal operators. Other sets of modal operators can be adapted to enhance the expressive power of the language. A different approach could also be taken by selecting a many-sorted first-order language with a special sort interpreted as time (see [CF,CCF,MWJ,BADW] for extensive discussions).

Given a (many-sorted) first-order language L, its temporal extension, LT, is defined as follows. The symbols of LT are those of L, plus one modal operator , the possibility operator denoted by '<>'. The modal operator $\Box$ of necessity is the dual of <> in that it can be introduced by definition as $\Box P \equiv -\Diamond -P$. The terms of LT are those of L and the set of wffs of L is defined using the familiar formation rules, plus one new rule:

if P is a wff of L or LT, then <>P is also a wff of LT

The semantics of LT is defined as follows. A universe U for LT is a pair (S,R), where S is a set of structures of L, all with the same domain D (this restriction can be relaxed, but it simplifies the treatment of quantifiers), and R is a binary relation over S, called the accessibility relation. Given a wff P of LT, a structure I in S and a valuation v over the common domain D, we define the notion that I satisfies P with v in U (denoted !=UI P[v]) using rules identical to those of first-order

languages, plus one additional rule:

!=UI (<>P)[v] iff there is J in S such that R(I,J) and !=UJ P[v]

A wff of LT without any modal operator is called a static wff. A fully temporal wff is conveniently viewed as consisting of static subformulas to which modal operators have been applied.

The notions of model, logical implication and theory are as for first-order languages.

Thus, to account for transition constraints, a data base is specified at the information level by defining a theory T1=(L1,A1), where L1 is a temporal extension of a (many-sorted) first-order language L and A1 is a set of axioms. The non-logical symbols of L1 describe the data base data structures and all ordinary symbols, such as "less than", used to express facts about the data base. Data base structures are represented by special predicate symbols, called db-predicate symbols. The axioms in A1 define static constraints, if they do not involve modal operators (i.e. are static wffs), or transition constraints, otherwise. The semantics of the data base is fixed by selecting a universe U=(S,R) for L1. The structures in S play the role of data base states and the relation R over S is interpreted as indicating that, if (I,J) is in R, then J is a future state with respect to I. A structure I in S corresponds to a consistent state iff it is a model of T1.

We note that the semantics of a data base, as explained above, is only loosely fixed by the theory T1, especially the relation R. This situation is modified when the functions level (i.e., algebraic) specification of the data base is fixed (section 4).

3.2. An example

We are now in a position to present our example data base and formalize it at the information level.

The example data base is defined by a theory T1=(L1,A1), where L1 is a many-sorted temporal language with two sorts, course and student, and two predicate symbols, **offered**, of sort <course>, and **takes**, of sort <student,course>. The intended interpretation of **offered**(c) is that course c is offered, and of **takes**(s,c) is that student s takes course c. The set A1 of axioms consists of two formulas:

(1) $-\exists s \exists c(\mathbf{takes}(s,c) \wedge -\mathbf{offered}(c))$

(2) $-\exists s \exists c(\Diamond(\mathbf{takes}(s,c) \wedge \Diamond(-\exists c' \mathbf{takes}(s,c'))))$

The first formula formalizes the static constraint: "a student cannot take a course that is not being offered". The second formula formalizes the transition constraint: "the number of courses taken by a student cannot drop to zero" (i.e., he cannot be taking a course in (some) current state and no course in a

future state).

To summarize, formalisms based on logic are best viewed as tools to describe data bases at the first level of specification since the set of consistent data base states and the set of consistent state transitions can be formalized by sets of axioms.

4. The functions level - the use of algebraic formalisms

Recall that the goal of a second level specification is to define a set of query and update functions that preserve the static and transition constraints listed at the first-level specification, provided that only such functions be used (the encapsulation strategy). This can be achieved by giving the data base application an algebraic specification [VF,DMW].

At the functions level, a data base is still specified as a first-order theory T2=(L2,A2). However, the similarities with a first-level specification fade out with a closer look at T2.

4.1. Algebraic formalisms

An algebraic specification is a first-order theory T=(L,A), where L is a many-sorted first-order language and A is a set of axioms obeying the following restrictions.

The set of sorts of L must include a Boolean sort and a designated sort state (also called sort-of-interest). The remaining sorts are called parameter sorts. The only predicate symbols of L are equality symbols of sort <s,s>, for each sort s. For simplicity, and since no ambiguity arises, they are all denoted by '='. We shall also use $t \neq t'$ as an abbreviation for $\neg\ t = t'$. The parameter sorts of L are endowed with their own function symbols (not involving the sort state either as a domain or range sort, and not including Boolean as domain sort). which have the effect of generating a set of ground terms called parameter names.

The Boolean sort will be equipped with two constants, True and False, and with function symbols standing for the usual connectives, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\equiv$ written in infix notation.

The language L may also have other function symbols as long as state occurs as one of the domain sorts and the range sort is Boolean or state. To simplify the notation, we assume that state is always the last one in the list of domain sorts. Thus, if f is an n-ary function symbol in this group, it must have a sort of the form <s1,...,sn-1,state,sn+1> (recall that sn+1 is the target sort). If sn+1 is the sort state then f is an update function (intuitively, it maps states into states according to some arguments); otherwise, f is a query function (it interrogates the current state (according to some arguments) and returns a value). Let f be an n-ary query function. Whenever terms of sorts other than state are irrelevant, we will write f(δ) instead of f(t1,...,tn-1,δ).

A term of the form q(t1,...,tn) where q is a query function and t1,...,tn contain no occurrences of update functions is called a simple observation. We will construct the language L2 to be sufficiently rich with queries so that states can be identified by means of simple observations. More precisely, if δ and δ' are state variables such that for all simple observations f we have

f(δ)=f(δ'), then δ=δ'. This observability condition is often fulfilled by data base applications due to their purpose.

The type of axioms allowed in algebraic specifications will be conditional equations, which are wffs of the form P -> t = t' where P is a wff and t and t' are terms of the same sort s. If s is state then we call the axiom an U-equation, otherwise we call the axiom a Q-equation. Often term t' is "simpler" than t and we can view an axiom as a conditional term-rewriting rule, namely, if condition c is fulfilled then t can be rewritten as t'. This operational interpretation has an intuitive appeal to it, which can be conveniently exploited.

An algebraic specification, being a theory, defines a set of structures, the models of the theory. (In the context of algebraic specifications, structures are called (many-sorted) algebras.) As usual, we further restrict this set to be the set of all finitely generated algebras (i.e., those in which every element is the value of a variable-free term) which are models of the axioms. Thus we can employ the principle of structural induction (on terms) as a proof rule.

We call an algebraic specification T = (L,A) sufficiently complete iff for every ground term of the form q(t1,..,tn), where q is a query function (with target sort s, say), there exists a parameter name p (of sort s) such that A != q(t1,...,tn) = p. Intuitively, a sufficiently complete algebraic specification is one enabling the evaluation of all queries.

Returning to the beginning of our discussion, we can concisely say that data base applications are specified at the functions level by algebraic specifications. The next section outlines in general terms the methodology to obtain such data base specifications.

4.2. Obtaining a functions level specification - an example

We now outline the methodology we employ to obtain an algebraic specification T2=(L2,A2) of a data base application at the functions level.

Consider again the data base application described at the information level by the theory T1=(L1,A1) of section 3.2. For simplicity, we take the parameter sorts of L2 as the sorts of L1. Moreover, we correlate the db-predicate symbols of L1 describing data base structures with query function symbols. So, L2 will contain two query function symbols, offered and takes, of sorts <course,state,Boolean> and <student,course,state,Boolean>, respectively. The intended interpretation of offered(c,δ), for example, is that it is True iff c is a course offered in state δ.

The update function symbols (with their intended interpretation) are: initiate of sort <state>, with initiate understood as an operation that initializes the data base; offer of sort <course,state,state>, where offer(c,δ)=ζ indicates that c is

added as a new course to state σ, creating state τ; cancel of sort <course,state,state>, where cancel(c,σ)=τ means the inverse of the previous operation; enroll of sort <student,course,state,state>, where enroll(s,c,σ)=τ creates a new state τ by enrolling student s to course c on state σ; transfer of sort <student,course,course,state,state>, with transfer(s,c,c',σ)=τ understood as creating state τ from state σ by transferring student s from course c to course c'.

Our task now is to write a set of conditional equations from which we can obtain the correct result of every query and, at the same time, guarantee that consistency is always preserved. In other words, for every query function q, for all parameters p and for all ground terms t of sort state, we should be able to derive from the axioms the equality q(p,t) = b where the Boolean value b is the correct answer according to the given description. Now, the set T of ground terms of sort state is the smallest set of terms containing initiate and closed under symbolic application of the other update functions. Thus, we shall strive for Q-equations of the form (perhaps with some condition)

q(p,u(p',σ)) = "simpler expression"

for all query functions q, update functions u and parameter lists p and p', σ being a variable of sort state.

We start from the informal description of the operations. As already mentioned the effect of each update function is changing the state as observed by the query functions. As a first step, we give a more structured description for each update function by identifying its intended effects, preconditions for state change, possible side effects, and simple observations that are not affected.

The general outlook of such a structured description by means of effects for an update u is

τ = u(parameters,σ)

intended effects: some simple observations q at state τ give specific values
pre-conditions: some simple observations q' at state σ have given values
side effects: some simple observations q may have their values altered by passing from state σ to state τ
not-affected: other simple observations q" maintain at state τ the value they had at state σ

For example, the informal description of the update function cancel would be structured as follows

τ = cancel(c,σ)
/* course c is cancelled at τ, provided that no student is taking it at state σ */

intended effects: offered(c,τ) = False
pre-conditions: ∀s(takes(s,c,δ) = False)
side-effects: none
not-affected: all other queries, including offered(c',.)
with c' ≠ c

As an example of the method, let us consider the update function cancel, whose structured effects description has been given above. We shall examine in detail the case of the query offered. In other words, we want (conditional) equations enabling us to derive the correct results of queries of the form

offered(c',cancel(c,δ))

We shall divide our task into two cases depending on the comparison of c' with c.

For the first case (c' ≠ c) the not-affected part of the structured description tells that the value of offered(c',.) is not affected by the update, i.e.

offered(c',cancel(c,δ)) = offered(c',δ)

We can put this into the form of a conditional equation

c' ≠ c -> offered(c',cancel(c,δ)) = offered(c',δ)

Notice that the antecedent of the conditional equation does not involve terms of sort state, only parameters. Also, the righthand side of the consequent is "simpler" than its lefthand side, for the term δ is "simpler" than the term cancel(c,δ). Thus, we can view this conditional equation as reducing the problem of determining the value of offered(c',cancel(c,δ)) to the simpler problem of determining offered(c',δ) in case c' ≠ c.

Now let us examine the case c' = c. According to the structured description, the value of offered(c,cancel(c,δ)) will depend on the precondition. If the precondition holds then we have the intended effect False. Otherwise the value remains unchanged. Thus, we have:

offered(c,cancel(c,δ)) =

False	if ∀s(takes(s,c,δ) = False)
offered(c,δ)	if ∃s(takes(s,c,δ) = True)

Now, in view of the static constraint, we have

∃s(takes(s,c,δ) = True) -> offered(c,δ) = True

So, we can write

offered(c,cancel(c,δ)) =

False	if (∃s takes(s,c,δ)) = False

True if (∃s takes(s,c,ó)) = True

which can be simplified to

offered(c,cancel(c,ó)) = True ≡ ∃s (takes(s,c,ó) = True)

This wff can be rewritten as two conditional equations:

∃s (takes(s,c,ó) = True) -> offered(c,cancel(c,ó)) = True

and

¬∃s (takes(s,c,ó) = True) -> offered(c,cancel(c,ó)) = False

Three remarks are in order. First, in obtaining this equation we used the static constraint (assumed to hold; we shall later have to verify that it does hold). Second, the antecedents of the above conditional equations do not involve quantification over states, only over parameters. Third, we may regard these equations as reducing the problem of determining offered(c,cancel(c,ó)) to that of determining whether there exists a student s such that takes(s,c,ó) = True, which may be viewed as a problem somewhat simpler than the original one. However we must be careful, for some other equation might reduce the problem of determining takes(s,c,ó) to that of determining offered(c,ó), thereby creating a circularity. This is the reason why we later verify termination.

By applying the general methodology outlined above we obtain the following set of Q-equations for our example

1. offered(c,initiate) = False
2. takes(s,c,initiate) = False
3. offered(c,offer(c,ó)) = True
4. c ≠ c' -> offered(c,offer(c',ó)) = offered(c,ó)
5. takes(s,c,offer(c',ó)) = takes(s,c,ó)
6. offered(c,cancel(c,ó)) = True ≡ ∃s (takes(s,c,ó) = True)
7. c ≠ c' -> offered(c,cancel(c',ó)) = offered(c,ó)
8. takes(s,c,cancel(c',ó)) = takes(s,c,ó)
9. offered(c,enroll(s,c',ó)) = offered(c,ó)
10. takes(s,c,enroll(s,c,ó)) = offered(c,ó)
11. s ≠ s' ∨ c ≠ c' -> takes(s,c,enroll(s',c',ó)) = takes(s,c,ó)
12. offered(c,transfer(s,c',c'',ó)) = offered(c,ó)
13. takes(s,c',transfer(s,c,c',ó)) = (offered(c',ó) ∧ takes(s,c,ó)) ∨ takes(s,c',ó)
14. takes(s,c,transfer(s,c,c',ó)) = (¬offered(c',ó) ∨ takes(s,c',ó)) ∧ takes(s,c,ó)
15. s ≠ s' ∨ (c ≠ c' ∧ c ≠ c'') -> takes(s,c, transfer(s',c',c'',ó)) = takes(s,c,ó)

4.3. First to second level refinements

The information and functions level specifications of a data base

application are bound by a notion of refinement we describe in this section.

Let T1=(L1,A1) and T2=(L2,A2) be the information and functions level specifications of the data base application. Intuitively, we say that T2 refines T1 iff all equations in A2 are sufficient to guarantee that all updates preserve consistency with respect to the static and transition constraints in A1. Although this condition is on the surface simple, it creates some technical difficulties to be formalized, mainly because the two languages, L1 and L2, are of different types. In particular, wffs of L1 may contain modalities, which are not part of L2.

For simplicity, we assume that every sort of L1 is a parameter sort of L2 and every variable of sort s is also a variable of L2. Of course L2 has two new sorts, state and Boolean, as well as new variables ranging over them.

The notion of refinement is formally defined by specifying an interpretation I mapping the non-logical symbols of L1 into terms of L2 with the following characteristics:

(1) for each n-ary db-predicate symbol r of sort <s1,...,sn> of L1, I(r) must be a term of L2 of sort Boolean and free variables x1,...,xn,y of sorts s1,...,sn,state

(2) for each other n-ary predicate symbol p of sort <s1,...,sn> of L1, I(p) must be a wff of L2 with free variables x1,...,xn of sorts s1,...,sn

(3) for each function symbol f of sort <s1,...,,sn,sn+1> of L1, I(f) must be a term of L2 of sort sn+1 and free variables x1,...,xn of sorts s1,...,sn

In our running example, we might define an interpretation I that assigns to the db-predicate symbol **offered** the term offered(c,ó) and to **takes** the term takes(s,c,ó).

Thus, the notion of interpretation defined above follows closely the idea of first-order interpretation. The differences are basically that some symbols of L1 are associated with terms of sort Boolean of L2, and not wffs of L2 as one would expect, and the addition of new sorts (and variables).

If t is a term of L2 with free variables x1,...,xn of sorts s1,...,sn and t1,...,tn are terms of L2 also of sorts s1,...,sn, let t[t1/x1,...,tn/xn] denote the term of L2 obtained by replacing xi by ti, i=1,...,n.

Given an interpretation I, we extend I to map wffs of L1 into wffs of L2. However, in order to do so, we must extend L2 by adding a predicate symbol F of sort t <state,state>, which will stand for the reachability relation R of the semantics of L1. The extension of I is defined as follows:

(1) for any wff P of L1, I(P) = (∀sJ(s,P)), where s is a variable of sort state of L2

The mapping J in turn maps pairs (s,P), where s is a variable of L2 of sort state and P is a wff of L1, into wffs of L2. J is defined as follows:

(2) J(s,x) = x, if x is a variable of L1

(3) J(s,f(t1,...,tn)) = I(f)[J(s,t1)/x1,...,J(s,tn)/xn], if f is a function symbol of L1

(4) J(s,p(t1,...,tn)) = I(p)[J(s,t1)/x1,...,J(s,tn)/xn], if p is a predicate symbol of L1 (other than a db-predicate symbol)

(5) J(s,r(t1,...,tn)) = I(r)[J(s,t1)/x1,...,J(s,tn)/xn,s/y] = True if r is a db-predicate symbol

(6) J(s,-P) = -J(s,P)

(7) J(s,P1 ^ P2) = J(s,P1) ^ J(s,P2)

(8) J(s,∃xP) = ∃xJ(s,P)

(9) J(s,[]P) = ∀s'(F(s,s') => J(s',P)), where s' is a variable of L2 of sort state not used before

(10) J(s,<>P) = ∃s'(F(s,s') ^ J(s',P)), where s' is a variable of L2 of sort state not used before

Thus, at this point we know how to map wffs of L1 into wffs of L2. Therefore, we can check if indeed the axioms of T2 are enough to guarantee that all updates of T2 preserve consistency. More precisely, we say that T2 is a correct refinement of T1 under a given interpretation I iff for any axiom P of T1, I(P) is a theorem of T2.

As for first-order languages, our notion of interpretation can also be used to induce a mapping from structures of L2 into universes of L1, which permits us to give an alternative (semantical) characterization of correct refinement. Indeed, given an interpretation I from L1 into L2, we define a mapping M from structures of L2, which must be finitely generated algebras by assumption, into universes of L1 as follows.

Let A be a structure of L2 and assume that A is finitely generated. Extend A to assign meaning to the predicate symbol F added to L2 as follows:

(e,e') ∈ A(F) iff there exists a term t(ó) of L2 of sort state, whose only variable is ó also of sort state, such that

!=A (t(ó)=ó')[e/ó,e'/ó']

where ó' is another variable of sort state. Intuitively, e' is F-

related to e in A iff some trace constructing e' will pass by e as an intermediate step.

Then, M(A) = (S,R) is the universe of L1 induced by A, where S is a set of structures for L1 differing only on the values of the db-predicate symbols of L1 and R is a binary relation over S. The set S is defined by taking each element e in the domain of A of sort state and constructing a structure E of L1 as follows:

(1) for each sort s of L1, the domain of E of sort s is the domain of A of sort s

(2) if f is a function symbol of L1 of sort <s1,...,sn,sn+1>, E(f) is the function defined by I(f) in A, that is,

$$E(f) = \{(a1,...,an,b)/\ A(I(f))(a1,...,an) = b\}$$

(3) if p is a predicate symbol of L1 of sort <s1,...,sn>, E(p) is the relation defined by I(p) in A, that is,

$$E(p) = A(I(p))$$

(4) if r is a db-predicate symbol of L1 of sort <s1,...,sn>, E(r) is the relation defined by I(r) and e, the element of the domain of A of sort state associated with E, that is,

$$E(r) = \{(a1,...,an)/\ A(I(r))(a1,...,an,e) = A(\underline{True})\}$$

The relation R is in turn defined as follows:

(5) (E,E') is in R iff (e,e') is in A(F) where e, e' are the elements of the domain of A of sort state associated with E and E', respectively.

We can prove one basic property of M. We say that a wff P of L1 is valid in a universe U=(S,R) of L1 iff P is valid in every structure E in S. Also, we call U a model of a set W of wffs iff every wff in W is valid in U.

THEOREM 4.1:

For any wff P of L1 and any structure A of L2, P is valid in M(A) iff I(P) is valid in A.

As a consequence of this theorem, we have an alternative definition of correct refinement:

THEOREM 4.2:

T2 is a correct refinement of T1 under a given interpretation I iff for any model A of T2, M(A) is a model of T1, where M is the mapping induced by I.

In addition to the above result, the notion of refinement also has a second semantic interpretation. Assume that T2 is a correct

refinement of T1. Then, T2 in fact refines the semantic specification of the data base by explicitly defining the reachability relation R in terms of the repeated application of update operations (which is the meaning assigned to the function symbol F).

4.4. Proof of correctness of the refinement - an example

Let T2=(L2,A2) be the algebraic specification of the data base application obtained in section 4.2. We must guarantee that T2 has the following properties:

- it is sufficiently complete and correct with respect to the structured description;
- it is a refinement of the first-level specification given in section 3.2.

By construction our equations are already correct with respect to the structured description. We proceed by proving:

(a) sufficient completeness
(b) static consistency, i.e. every reachable state is valid
(c) transition consistency

Parts (b) and (c) are equivalent to saying that the refinement is correct.

We outline below how these properties can be proven.

(a) Sufficient completeness

We can view our set of Q-equations as a system of mutually recursive equations defining the query functions. From this viewpoint, sufficient completeness amounts to termination of this system of recursive definitions. There are several criteria for checking termination of such term rewriting systems. However, the basic idea is checking the absence of circularity in these definitions. This basic idea will do for cases simple as our example, as we now illustrate.

We consider a graph whose nodes are the terms occurring in the equations disregarding their parameters. For each equation with term t0 on its lefthand side and t1,...,tn on its righthand side we draw directed edges from the node corresponding to t0 to those corresponding to t1,...,tn. For instance, equations 6 and 7 will contribute the following partial graph

The graph for our example will have 12 nodes and 14 edges. It is easy to check that it has no cycles.

Since this graph has no cycles it corresponds to a well-founded relation on ground queries. Therefore, the recursive definitions do terminate.

(b) Every reachable state is valid

Consider the set V of all valid states, i.e. the set defined by

$\forall c \forall s$ (takes(s,c,δ) = True -> offered(c,δ) = True)

The set G of reachable states is the least set of states containing the value of initiate and closed under all the other update functions. So in order to show that the static constraint is satisfied at the functions level, i.e. $G \subseteq V$, it suffices to show that V contains initiate and is closed under all the other update functions.

Clearly, by equation 2, initiate is in V. To show that V is closed under applications of cancel, we have to check that cancel(c',δ) ∈ V whenever δ ∈ V. For this purpose we have to show that for all courses c and students s if takes(s,c,cancel(c',δ)) = True then offered(c,cancel(c',δ)) = True. We consider two cases:

case 1: $c \neq c'$

Then, by equations 7 and 8, we have

offered(c,cancel(c',δ)) = offered(c,δ)
takes(s,c,cancel(c',δ)) = takes(s,c,δ)

whence, as δ ∈ V, we have the desired implication.

case 2: c = c'

Then, by equation 6,

offered(c,cancel(c',δ)) = True = $\exists s'$ takes(s',c,δ) = True

This suggests considering two subcases according to the value of takes(s,c,δ). First, in case takes(s,c,δ) = True, then

$\exists s'$ takes(s',c,δ) = True, so

offered(c,cancel(c',δ)) = True

On the other hand, if takes(s,c,δ) = False, then, by equation 8 we obtain

takes(s,c,cancel(c,δ)) = takes(s,c,δ) = False

Therefore, the implication always holds and we can conclude

cancel(c',δ) ∈ V.

By proceeding similarly with the other update functions and invoking the appropriate equations we check the closure of V, whence G $\subseteq$ V.

Notice that we have illustrated with our example a perfectly general method to show that the functions level of a data base application given by an algebraic specification satisfies the declared static constraints. Namely, by invoking the equations show that the set of all valid states is closed under all update functions (including constants), which can be done syntactically, as above.

(c) Transition consistency

The transition constraint of our example (see section 3.2) is logically equivalent to

$$\forall s \forall c\ [\Box(\underline{takes}(s,c) \rightarrow \Box(\exists c'\ \underline{takes}(s,c')))]$$

which can be rewritten, by applying the notion of refinement as

$$\forall \delta 0 \{\forall s \forall c \forall \delta [F(\delta 0,\delta) \rightarrow [\underline{takes}(s,c,\delta) = \underline{True} \rightarrow \forall \zeta (F(\delta,\zeta) \rightarrow \exists c'\ \underline{takes}(s,c',\zeta) = \underline{True})]]\}$$

where F corresponds to the accessibility relation.

We shall first check

$$\forall s \forall c \forall \delta [\underline{takes}(s,c,\delta) = \underline{True} \rightarrow \exists c'\ \underline{takes}(s,c',u(\delta)) = \underline{True}]$$

for each update function u other than initiate.

We illustrate this checking with the case of cancel. For this purpose, notice that, by equation 8

$$\underline{takes}(s,c',\underline{cancel}(c'',\delta)) = \underline{takes}(s,c',\delta).$$

So, if $\underline{takes}(s,c,\delta) = \underline{True}$ then there exists c' = c such that $\underline{takes}(s,c',\underline{cancel}(c'',\delta)) = \underline{True}$.

The case of offer is entirely similar. For the update functions enroll and transfer the checking can be performed by breaking into cases depending on the comparison of the values of the parameters.

Thus we have that every single-update transition obeys the transition constraint. It follows readily, by induction, that every transition (effected by a sequence of updates) also obeys the transition constraint.

5. The representation level - the use of a programming language formalism

As the name indicates, the representation phase, the last phase of the specification process, provides a representation of the objects of the data base and a description of the functions using some appropriate language. However, since we remain at the specification level, the representation of the objects should be based on abstract data structures, such as those underlying the current data models, and the functions should be programmed as procedures written in a programming language supporting the data structures of the data model. Moreover, the programming language should be simple and theoretically sound and must be formally specified so as to permit verifying if the procedures indeed realize the functions.

We note that, by specifying a language associated with a data model, we are in a sense providing a formal specification of the data model itself.

5.1. Programming language formalism

This section describes the syntax and semantics of what we call a data base schema at the representation level. The language to be used, RPR, is based on an extension of the concept of regular programs over relations described in [CB], to which the reader is referred for a fuller discussion.

5.1.1. Syntax - the use of a grammatical formalism

Briefly, the syntax of a data base schema is defined as follows. Let L be a many-sorted first-order language with a set of distinguished constants, called scalar program variables. If P is a wff of L with free variables x1,...,xm, then we call an expression of the form {(x1,...,xm)/P} a relational term of type <s1,...,sm>, if si is the type of xi.

A data base schema has the following format:

schema SCL ; OPL end-schema

SCL is a list of statements of the form R(A1,...,An) where R is a predicate symbol of L and A1,...,An are unary predicate symbols of L such that, if <s1,...,sn> is the sort of R, then Ai has sort <si>, for each i = 1,...,n. Each predicate symbol R in SCL is called a relation name or relational program variable. OPL is a list of operation declarations of the form "proc I(Y1,...,Yn) = S" where I is an operation identifier, Yi is either a scalar or a relational program variable, and S is a statement, called the operation body.

The set of statements (based on L), is defined inductively as follows:

(1) For any scalar program variable x of L and any variable-free term t of L of the same type as x, the expression x := t is an assignment statement.

(2) For any relational program variable R of L and any relational term F of the same type as R, the expression R := F is a relational assignment statement.

(3) For any closed wff P of L, P? is a test statement.

(4) For any statements p and q, the expressions (p U q), (p ; q) and p* are statements called the union of p and q, the composition of p and q and the iteration of p, respectively.

We may also introduce some familiar constructs by definition as follows:

(5) if P then r else s = (P?;r) U (-P?;s)

(6) if P then r = (P?;r) U -P?

(7) while P do r = (P?;r)*;-P? U -P?

(8) insert R(x1,...,xn) = R := {(y1,...,yn)/R(y1,...,yn) V
(y1=x1 ^ ... ^ yn=xn)}

(9) delete R(x1,...,xn) = R := {(y1,...,yn)/R(y1,...,yn) ^
-(y1=x1 ^ ... ^ yn=xn)}

If we use only the constructs in (5) to (9) in lieu of (3) and (4) we obtain the set of deterministic programs.

The formal definition of the syntax of data base schemas is given using W-grammars (see also [FVC]). W-grammars (as also other comparable formalisms, such as attribute grammars and affix grammars) go beyond BNF in that they can express context-sensitive restrictions (e.g. that all relational program variables in the OPL part of a schema have been declared in the SCL part), and can be used to build compiler generators. A correspondence between W-grammars and logic has been established in [He].

A W-grammar is an 8-tuple G = (M,S,T,z,MT,H,RM,RH), where

- M is a finite set of metanotions (which are denoted by sequences of capital letters);

- S is a finite set of s-notions (which are denoted by sequences of lower-case letters);

- T is a finite set of terminals (which are denoted by sequences of lower-case letters between double quotes (following [Pe]) or by indicated special symbols);

- z in S+ is the start symbol;

- MT C S is a set of metaterminals;

- H C (M U S)+ is a set of hypernotions;

- RM C M x (M U MT)* is a finite set of metarules (which are written as X0 :: X1 X2 ... Xm .);

- RH C H x (H U T)* is a set of hyperrules (which are written in the form x0 : x1 , x2 , ... , xn .).

notes:

(1) Sets M, S and T are pairwise disjoint;

(2) Metarules with the same left-hand side may be combined using slashes to separate the right-hand side alternatives; the same convention also applies to hyperrules. The null sequence is denoted by '&'.

By applying metarules just like ordinary context-free rules, one can generate from a metanotion X sequences of metaterminals called metaproductions of X. In this paper, the language generated by a metanotion X, L(X), consists of all metaproductions derived from X by such production rules.

Consider a hyperrule x0 : x1 , ... , xk . . Since each xi is an element of (M U S)+, it may contain occurrences of X. By taking a metaproduction y of X and consistently replacing each occurrence of X in the hyperrule by y we obtain a new rule x0' : x1' , x2' , ... , xk' . . If we perform this process of uniform replacement on all metanotions occurring in the hyperrule we obtain the context-free production rule x0" : x1" , x2" , ... , xk" . where each xi" is a sequence without metanotions. The language generated by a hypernotion h, L(h), consists of all sequences of terminals derived from h by such production rules and the language generated by the W-grammar G is L(z).

Whenever possible, a W-grammar will be defined by exhibiting just its meta and hyperrules, leaving implicit the other elements of the grammar.

We now turn to the definition of the syntax of RPR. To simplify the discussion, we assume that we are given a many-sorted first-order language L whose syntax is defined by a W-grammar GL with the following metanotions:

- V such that L(V) is the set of variables of L;

- P such that L(P) is the set of predicate symbols of L;

- R such that L(R) is a set of distinguished predicate symbols of L (these will be called relational program variables):

- F such that L(F) is the set of function symbols of L;

- X such that L(X) is a set of distinguished constants (0-ary function symbols) of L (these will be called scalar program variables);

- T such that L(T) is the set of terms of L;

- W such that L(W) is the set of wffs of L;

and, for each w in L(W) and each list $\vec{x}$ of variables, the hypernotions below (These are cases of what is called a predicate in the W-grammar terminology. We say that a predicate succeeds if through the application of appropriate production rules, it eventually vanishes (i.e. the symbol & is generated); otherwise a blind alley situation is reached.):

- 'where w is closed' such that L(where w is closed) is {&} if w is a closed wff, and Ø otherwise;

- 'where {$\vec{x}$/w} is well formed' such that L(where {$\vec{x}$/w} is well formed) = {&} if $\vec{x}$ is a list of all variables that occur free in w and is Ø otherwise;

and, for each n-ary relational program variable R and for each list A1,...,An of unary predicate symbols, the following hypernotion:

- 'where A1,...,An matches r' such that L(where A1,...,An matches R) is {&}, if Ai is of sort <Ti>, $1 \leq i \leq n$, and R is of sort <T1,...,Tn>, and is Ø otherwise;

and, for each n-ary program variable R, each relational term E and each list of schemes SCL, the following hypernotion:

- 'where E relational program variables are in SCL' such that L(where E relational program variables are in SCL) is {&}, if each program variable appearing in E also appears (i.e. has been declared in SCL, and is Ø otherwise; for the transformation of the hypernotion into {&}, the general predicate 'where ... contains ... ' [Pe] will be applied to check if each relational program variable R encountered in E is contained in SCL (notice a similar application of the same general predicate in the last hyperrule of the W-grammar below);

and, for each term t (relational or not) and each program variable x (relational or scalar), the following hypernotions:

- 'where t is ground' such that L(where t is ground) = {&}, if t is a ground term, and Ø otherwise.

- 'where t agrees with x' such that L(where t agrees with x) is {&} if t and x are of the same type and Ø otherwise.

This concludes what we assume about the W-grammar GL. The W-grammar defining the syntax of data base schemas is shown below (the start symbol is p, the first hyperrule to be invoked being indicated by '->').

```
Metarules
   First-Order Objects
      (inherited from W-grammar GL)
   Auxiliary objects
      /*
        for each i = 1,...,p
      */
      I  :: I1 ! ... ! Ik .          /* identifiers                */
      E  :: { VL / W } .              /* relational terms           */
      Y  :: X ! R .                   /* generic program variable   */
      F  :: T ! E .                   /* generic term               */
      YL :: Y YL ! & .                /* list of generic variables  */
      FL :: F FL ! & .                /* list of generic terms      */
      VL :: V VL ! V .                /* list of variables          */
   First-Order Rules
      (inherited from GL)
   Programming-Language Rules
      /*
        context-free aspects of the syntax of RPR
      */
      Q :: schema SCL ; OPL end-schema . /* schema declaration */
      SCL :: RL ; SCL ! RL .          /* list of schemes            */
      RL :: R(PL) .                   /* schemes definition         */
      PL :: P ; PL ! P .              /* list of attributes         */
      OPL :: OP ; OPL ! OP .          /* list of operations         */
      OP :: proc I (FL) = S .         /* operation declaration      */
      S :: S ; S ! S U S ! S* !       /* statements                 */
           W? ! X := T ! R := E .
Hyperrules
   First-Order Rules
      (inherited from GL essentially to define closed wffs and
       relational terms)
      c : W , where W is closed . /* L(c) is the set of all
                                     closed wffs                 */
      g : T , where T is ground . /* L(g) is the set of all
                                     ground (variable-free) terms */
   Programming-Language Rules
   -> p : repr Q , where Q defined .
      /*
        terminal representation of programs -
        metaterminals are enclosed in quotes
      */
      repr schema SCL ; OPL end-schema :
           "schema" , repr SCL , ";" , repr OPL , "end-schema" .
      .                              /* further rules enclosing
      .                                 syntactical objects
      .                                 within quotes           */
      repr & : & .
      /*
        checking procedure declarations and other statements
```

```
*/
where schema SCL ; OPL end-schema defined :
      where SCL defined ,
      where <SCL> OPL defined .
where R[PL] SCL defined : where PL matches R ,
                          where SCL defined .
where <SCL> proc I (YL) = S ; OPL defined :
      where <SCL> S defined , where <SCL> OPL defined .
where <SCL> & defined       : & .
where <SCL> S1 ; S2 defined : where <SCL> S1 defined ,
                              where <SCL> S2 defined .
where <SCL> S1 U S2 defined : where <SCL> S1 defined ,
                              where <SCL> S2 defined .
where <SCL> S* defined      : where <SCL> S defined .
where <SCL> W? defined      : where W is closed ,
       where W relational program variables are in SCL .
where <SCL> X := T defined  : where T agrees with X ,
                              where T is ground .
where <SCL> R := E defined  : where E agrees with R ,
                              where SCL contains R ,
       where E relational program variables are in SCL .
```

5.1.2. Semantics - the use of a denotational formalism

Again, let us briefly discuss the semantics of data base schemas before giving the formal definitions.

Let L be the underlying many-sorted first-order language. For a given structure A of L and a given non-logical symbol s of L, let A(s) denote the value of s in A. Likewise, let A(t) be the value of a variable-free term t of L in A and let A(F) be the relation denoted by F, if F is a relational term.

A universe U for L is a set of structures of L satisfying three conditions:

(i) any two structures in U differ only on the values of the scalar or relational program variables;
(ii) for any A in U, any scalar program variable x and any element e of the domain of the sort of x, there is B in U such that A and B differ only on the value of x, which is e in B;
(iii) for any A in U, any relational program variable R of sort (s1,...,sn) and any n-ary relation r C Ds1 x ... x Dsn, where Dsi is the domain of sort si, there is B in U such that A and B differ only on the value of R, which is r in B.

These conditions guarantee that, for example, if the value of x is changed to e, the resulting structure is in U, that is, the universe is closed under assignment, so to speak. Note that, by (i), all structures in U have the same domain and the same value on all symbols, except on the scalar and relational program

variables.

For a fixed universe U of L, the meaning of statements is given by a function m assigning to each statement in RPR a binary relation in U as follows:

(1) m(x:=t) = {(A,B) / B is equal to A, except that B(x) = A(t)}
(2) m(R:={(x1,...,xn) / P}) = {(A,B) / B is equal to A, except that B(R) is the n-ary relation defined by P in A}
(3) m(P?) = {(A,A) / P is true in A}
(4) m(p U q) = m(p) U m(q) (union of both binary relations)
(5) m(p;q) = m(p) . m(q) (composition of both binary relations)
(6) m(p*) = (m(p))* (reflexive-transitive closure of m(p))

The meaning of procedure declarations is given by a function k assigning to each procedure declaration d of the form proc I(Y1,...,Ym) = S a function from Ds1 x ... x Dsm into the set of all binary relations over the universe, where Dsi is the domain of type si and Yi is of type si. The function k is defined as follows:

(7) k(d) = f iff for any (c1,...,cm) in Ds1 x ... x Dsm, f(c1,...,cm) is the set of all pairs (A,B) in UxU such that (A[c1/Y1,...,cm/Ym],B) is in m(S).

We now sketch a formal definition of the semantics of data base schemas using the denotational approach. A certain familiarity with this approach at the level of, e.g. [Pa], is assumed.

The semantics will be specified by defining functions that assign to each element in a set of syntactical objects, called a <u>syntactical domain</u>, a value taken from a <u>semantic domain</u>.

Each syntactical domain will coincide with the language associated with a non-terminal of the W-grammar introduced in section 5.1.1. For example, the language associated with V is the <u>domain of variables</u>. The table below lists all syntactical domains of interest. By convention, N : S indicates that S is the name of the syntactical domain associated with the non-terminal N, that is, of the language L(N) associated with N by the W-grammar.

<u>Syntactical domains</u>

<u>First-Order Domains</u>

V	: Var	variables
P	: Pred	predicate symbols
F	: Func	function symbols
T	: Term	terms
g	: G-term	variable-free (ground) terms
W	: Wff	well-formed formulas (wffs)
c	: C-wff	closed wffs

<u>Programming-Language Domains</u>

OP	: Oper	operations
S	: Stmt	statements

X : S-var	scalar program variables
R : R-var	relational program variables
E : R-term	relational terms
I : Iden	identifiers
Auxiliary Domains	
Y : G-var	generic variables
F : G-term	generic terms
VL: L-var	list of variables

The semantic domains are defined in the next table. Some comments come in order before introducing the table, though. The notation v : S this time indicates that v is a variable (of the metatheory) taking values from the set S. As usual in the denotational approach, T and ⊥ indicate 'overdetermined' and 'undetermined', respectively. The set ST consists of states or functions assigning a scalar value to each scalar program variable and relation to each relational program variable, respecting their types.

Semantic Domains

t : TR = {True,False}	truth values
vi : Dsi	a domain of values for type si including T and ⊥
s : ST	a universe of states

The semantic functions are just five and are defined in the sequel. We have functions m and k, as discussed before, which are defined in terms of three other functions taken from first-order logic. We have a function A that is a structure with domain Ds1,...,Dsm for the underlying first-order language L (with sorts s1,...,sm), except that A does not assign meaning to the scalar or relational program variables (which are distinguished constants and predicate symbols of the language). These objects have their meaning fixed by a state. Or, putting it differently, a state s together with A determine a structure A(s) for L. We have a function I which, for each such structure A(s), acts as the interpretation of the closed formulas and variable-free terms of L based on A(s). Finally, we have a function J which, for each structure A(s), assigns a relation over the appropriate domains to each relational term.

The following notational conventions will be used. If o is a syntactical object and f is a semantic function, then f⟦o⟧ will denote the value of o assigned by f; if f:A->(B->C) is a semantic function, o is in A and q is in B, then we use f⟦o⟧q to denote the value assigned to o and q by f; if s is a state, x is a scalar program variable of sort t and b is an element of the domain of t, then s[b/x] is the state that is equal to s, except that the value of x is b; likewise, s[r/R] denotes the state that is equal to s, except that the value of a relational program variable R is a relation r over the correct domains. Finally, if A and B are binary relations, then A . B, A ∪ B and A* denote the composition of A to B, the union of A and B and the reflexive and transitive closure of A, respectively, and xAy denotes a

pair in A.

Semantic Functions and Equations

Semantic Functions

k : Oper -> (**U** Ds -> P(STx ST))
where s=(si1,...,sij) ranges over the set of all sequences of types and Ds indicates Dsi1x...xDsij, and where P(C) denotes the powerset of C

m : Stmt->P(STxST)

A : Pred U Func -> (**U** P(Ds) U **U** Ds -> Dsi)
(same remark as above)

I : G-Term U C-Wff -> ST -> (**U** Dsi U TR U **U** P(Ds))
(same remark as above)

J : R-Term -> ST -> **U** P(Ds)
(same remark as above)

Semantic Equations

First-Order Logic Equations

- for each s in ST, s U A is a structure for L, the underlying many-sorted first-order language
- for each s in ST, s U A induces an interpretation for L, which fixes the value of I[a]s when a is in G-Term U C-Wff
- J[E]s = {(d1,...,di) ∈ Dsj1x...xDsji / I[W[d1/v1,...,di/vi]]s = true}
  with E = {(v1,...,vi)/W}, where W has free variables v1,...,vi and vi is of sort sji.

Programming Language Equations

m[S1 ; S2] = m[S1] . m[S2]
m[S1 U S2] = m[S1] U m[S2]
m[S*] = (m[S])*
r m[W?] q iff r = q and I[W]r = True
r m[X:=T] q iff q = r[I[T]r/X]
r m[R:=E] q iff q = r[J[E]r/R]
k[proc I(Y1,...,Ym)=S] = {((c1,...,cm),(C,B)) ∈ (Dsi1x...xDsim) x (UxU) / (C[c1/Y1,...,cm/Ym],B) ∈ m[S]}
where Yi is of sort si

5.2. Obtaining a representation level specification - an example

As we did when passing from level 1 to level 2 of the specification of our example data base application, we pass from level 2 to level 3 by first using a constructive and systematic strategy; at a later stage we shall prove the passage correct.

Obtaining the third level specification means to express in the programming language introduced in the previous section both the kinds of predicates to be used, under the guise of relations, and the query and update functions that will act upon them. The query functions are trivially introduced, by noting that the language allows logical-valued expressions of the form R(t), which yield True if t is in R, and False otherwise.

In order to obtain in a constructive manner procedures that implement the desired update functions, we first correlate the four parts of our structured (semi-formal) description of update functions with the semantics of the statements of the programming language. The parts of the structured description are:

a. intended effects
b. pre-conditions
c. side-effects
d. not-affected elements

In turn, the main statements of the programming language are:

1. assignment
2. test

which can be put together in a procedure body by

3. composition
4. union
5. iteration

From the semantic definitions, one readily sees that, in the simpler cases, an update function f will follow the pattern:

```
proc f(x) = (pre-conditions?; effects; side-effects) U
            -pre-conditions?
```

or, using the if-then construct:

```
proc f(x) =
    if pre-conditions
    then (effects;
          side-effects)
```

The assignment statement is the only way to achieve effects and side-effects, since it alone can modify any values in a state; moreover, all other values in the state are not affected. Relational assignment can be specialized to the insert and delete statements, which handle a single tuple. From the definition of those statements, it is clear that, after inserting a tuple t in a relation R, R(t) is true regardless of the situation at the previous state, and that if t is deleted from R then R(t) is false independently again of the previous state; in both cases nothing else is affected, i.e. except for the value of R(t) the new and the previous states are identical.

We look again at the structured description of the function cancel:

```
ϒ = cancel(c,ϕ)
intended effects: offered(c,ϒ) = False
pre-conditions: ¬∃s (takes(s,c,ϕ) = True)
side-effects: none
```

not-affected: all other queries, including offered(c',..) with c'≠c

Using the pattern above, we are led to write:

```
proc cancel(c) =
  if -∃s TAKES(s,c)
  then delete OFFERED(c)
```

More complex updates may require (possibly nested) tests and iterations. The latter are useful, in particular, to check a universally-quantified pre-condition. Explicitly quantified pre-conditions and the general form of assignment lead to a more "set-oriented" style of programming, whereas the use of iteration and insert/delete statements favors a "tuple-oriented" style.

The complete programming language specification for the example is given below:

```
schema

OFFERED(Students);
TAKES(Students,Courses);

  proc initiate() =
    (TAKES := 0;
     OFFERED := 0)

  proc offer(c) =
    insert OFFERED(c)

  proc cancel(c) =
    if -∃s TAKES(s,c)
    then delete OFFERED(c)

  proc enroll(s,c) =
    if OFFERED(c)
    then insert TAKES(s,c)

  proc transfer(s,c,c') =
    if TAKES(s,c) ^ -TAKES(s,c') ^ OFFERED(c')
    then (delete TAKES(s,c);
          insert TAKES(s,c'))

end schema
```

5.3. Second to third level refinements

We repeat in this section the exercise of section 4.3, this time showing what it means for a representation level specification of a data base application to be a refinement of a functions level specification of the same application.

Let T2 = (L2,A2) and T3 be the functions and representation level specifications of the same data base application. Then, the

operations defined by procedures in T3 must satisfy all equations in A2. Again, we must face the fact that T2 and T3 use different formalisms so we do not have a notion of interpretation readily available.

Recall that T3 uses a programming language, which is in turn based on a first-order language, say, L3.

For simplicity, we assume that every parameter sort of L2 is a sort of L3 and every variable of sort s of L2 is also a variable of L3. For simplicity we assume that every parameter sort of L2 is a sort of L3 and every variable of sort s of L2 is also a variable of L3.

The notion of refinement is again formally defined by specifying a mapping K from the non-logical symbols of L2 into non-logical symbols of L3, wffs of L3 and procedure declarations of T3. The mapping K must satisfy the following requirements:

(1) for each n-ary update function symbol u of L2 of sort <s1,...,sn-1,state,state>, K(u) is a procedure declaration proc U(y1,...,yn-1) = S in T3 such that yi is of sort si, for i = 1,...,n-1.

(2) for each n-ary query function symbol q of L2 of sort <s1,...,sn-1,state,Boolean>, K(q) is a wff of L3 with free variables x1,...,xn-1 of sorts s1,...,sn-1

(3) for each n-ary function symbol f of L2 of sort <s1,...,sn,Boolean>, except those in (2) and those representing logical connectives, K(f) is a wff of L3 with free variables x1,...,xn of sorts s1,...,sn

(4) for each n-ary function symbol f of L2 of sort <s1,...,sn,sn+1>, with sn+1 not equal to Boolean or state, K(f) = f

note: the requirement in (4) could be generalized to K(f) being a wff of L3 with free variables x1,...,xn,y of sorts s1,...,sn+1, if we could force the wff K(f) to define a function as for first-order interpretations.

We now pause for a comment from our formalisms department.

If the reader remembers section 4.3, the next natural step would be to extend K to map wffs of L2 into wffs of L3. However, L3 is not powerful enough to permit us to carry on such extension. In order to do so, we would need a full programming logic, such as Dynamic Logic (a separate paper will explore this possibility).To circumvent this difficulty, we adopt a semantic definition of correct refinement.

Thus, using an interpretation K, we define a mapping N from finitely generated universes of L3 into finitely generated structures of L2, defined as follows.

Let U be a finitely generated universe of L3. That is, U is a set of structures of L3 differing only on the relation names declared in T3 and on the scalar program variables of L3 such that U is generated by the procedures declared in T3. We also assume that each procedure p declared in T3 is deterministic. Thus, if p has parameters of sorts s1,...,sn, then the semantic equations associate with p a function

k⟦p⟧: Ds1 x ... x Dsn x U -> U

Now, N(U) is a structure A of L2 defined as follows. Let E be any structure of L3 in U:

(1) The domain of A of each parameter sort s coincides with the domain of E of sort s (this is well-defined because every parameter sort of L2 is also a sort of L3, by assumption, and two structures in U have the same domains).

The domain of sort state is U itself, the domain of sort Boolean is {true,false};

(2) if u is an n-ary update function symbol of L2, A(u) = k⟦K(u)⟧;

(3) if q is an n-ary query function symbol of L2, A(q) is the function defined by K(q), that is,

A(q) = {(a1,...,an-1,E,true)/ E != K(q)[a1/x1,...,an-1/xn-1]}
U {(a1,...,an-1,E,false)/ E !≠ K(q)[a1/x1,...,an-1/xn-1]}

(4) if f is an n-ary function symbol of L2 of sort <s1,...,sn;Boolean>, except the query function symbols and those representing logical connectives, A(q) is the function defined by K(q), that is,

A(q) = {(a1,...,an,true)/ E != K(q)[a1/x1,...,an/xn]}
U {(a1,...,an,false)/ E !≠ K(q)[a1/x1,...,an/xn]}

(5) if f is a function symbol of L2 of sort <s1,...,sn,sn+1> with sn+1 not equal to boolean or state, A(f) = E(K(f))

(6) if c is a constant of L2, A(c) = E(K(c))

(7) if = is the equality symbol of L2 of sort <s,s,Boolean>,

A(=) = {(a1,a2,true)/ E != a1 = a2} U {(a1,a2,false)/
E != a1 ≠ a2}

(8) if f is a function symbol of L2 of sort <Boolean,Boolean,Boolean> or <Boolean,Boolean> standing for one of the logical connectives, A(f) is a representation of the truth table of the connective

We can prove that, since U is a finitely generated universe of

L3, N(U) is a finitely generated structure of L2.

Since it is not possible to map wffs of L2 into wffs of some formal language connected to T3, theorem 4.1 has no counterpart here.

Now, using N, we precisely characterize when T3 is a correct refinement of T2.

Definition - We say that T3 is a correct refinement of T2, under a given interpretation K, iff for every finitely generated universe U of L3, N(A) is a model of T2.

This concludes our discussion about second to third level refinements.

5.4. Proof of correctness of the refinement - an example

On analysing the constructive strategy (section 5.2) we observe that the semi-formal considerations that resulted in the algebraic equations of the second level were used but not the equations themselves. Similarly, our understanding of the semantics of the programming language constructs described denotationally was used but, again, the formal denotational description does not appear directly in the process. Finally, we insisted on writing the specification strictly as imposed by the syntactical description of the language, without however making explicit usage of the grammar productions. The formal machinery is necessary, and thereby justified, when we proceed to the verification of correctness, to be developed in the sequel.

We have to verify that the representation level is a correct refinement of the functions level. This amounts to checking, basically, that the procedures define update functions satisfying the equations of the algebraic specification, once the syntactical correctness of the programming language specification is ascertained.

In order to verify that the above programming language specification is syntactically correct we have to guarantee that it can be generated by the W-grammar in section 4.1. Since p is the start symbol, we have to check that the specification is, so to speak, of the form

repr Q , where Q defined

In view of the metarule for Q, this amounts to

(*) repr schema SCL ; OPL end-schema ,
where schema SCL ; OPL end-schema defined

Now, using the metarule for SCL together with the rules inherited from the underlying grammar GL we can generate the following metaproduction of SCL

OFFERED(S) , TAKES(S,C)

Similarly, using the metarules for OPL, OP and S together with rules inherited from GL we can generate the following metaproduction of OPL

```
proc initiate() = ..............;
proc transfer(s,c,c') =
 ([TAKES(s,c) ^ -TAKES(s,c') ^ OFFERED(c')]? ;
 [TAKES := {(y1,y2)/TAKES(y1,y2) ^ -(y1=s ^ y2=c)} ;
 TAKES := {(y1,y2)/TAKES(y1,y2) V (y1=s ^ y2=c')}]) U
 (-[TAKES(s,c) ^ -TAKES(s,c') ^ OFFERED(c')])?
```

We can now uniformly replace each occurrence of SCL and OPL in (*) by the corresponding metaproduction. Then application of the "repr" programming language hyperrules will convert the "repr" part of (*) into terminals.

It remains to check that the "where" part (a predicate) of (*) reduces to &. This amounts to checking that each wff used in a test, such as TAKES(s,c) ^ -TAKES(s,c') ^ OFFERED(c') is indeed closed and that the lefthand and righthand sides of the assignments have the same types, besides checking that all relational program variables in OPL are indeed declared in SCL.

Thus, the predicate succeeds and we generate a sequence of terminal symbols, which becomes our programming language specification upon application of the definitions of the constructs *if ... then*, *insert*, etc.

Therefore, we have ascertained the syntactical correctness of our specification.

We now outline how we can verify that the representation level specification T3 (section 5.2) is a correct refinement of the functions level specification T2=(L2,A2) (section 4.2) under the interpretation K defined below:

K(*offered*) = OFFERED(c)
K(*takes*) = TAKES(s,c)
K(u) = U, where u is an update function and U is the homonym procedure

Let L3 be the underlying language of T3.

Intuitively, given a universe U for T3, the interpretation K induces a finitely generated structure A for L2. At this point, it suffices to clarify that each element p of the domain of the sort *state* of A will be in fact a structure in U. From now on, we will refer to such elements simply as States and use p, q, r,..., with subscripts if necessary, to denote them (the reader must therefore bear in mind that states are structures of L3).

Moreover, the domain of sort *state* of A is finitely generated by construction. That is, each element p of the domain of sort *state*

of A is the value of a term of L2, which is schematically of the form:

un(un-1(...u1(u0)...))

where u0 is the update function symbol initiate of L2 and ui, with i = 1,...,n, are also update function symbols of L2. Intuitively, since the data base application is encapsulated by the query and update functions, the current data base state can be represented by such terms, indicating the operations used thus far.

To prove that T3 is a correct refinement of T2 amounts to proving that each of the conditional equations in A2 is (universally) valid in A. Now, since A is finitely generated and in view of the previous discussion, we can in fact do an induction on the length of the terms corresponding to each element of the domain States of sort state of A. That is, for each P in A2, we will prove by induction on n that P is valid in A when the variable ó receives as value some state which is in turn the value of a term un(un-1(...(initiate)...)) of L2.

The basis is trivial. So assume that each P in A2 is valid in A when the state variable ó receives as value some state p and p is the value of a term un-1(un-2(...(initiate)...)) of L2. We will show that this result holds when we consider terms of length n.

Now, let q be an element of States and assume that q is the value of a term un(un-1(...(initiate)...)) of L2.

As an example, consider equation 6, namely

(1) offered(c,cancel(c,ó)) = true ≡ ∃s (takes(s,c,ó) = True)

This equation is (universally) valid in A when ó is valuated as p iff the following condition holds (from now on, C will denote the domain of sort course of A, T will denote the domain of sort student of A and S will denote the domain of sort state of A), where q is the state that the data base will reach upon application of the procedure to cancel course c at state p:

I[OFFERED(c)]q = I[∃s TAKES(s,c)]p,
    where (p,q) ∈ k[cancel(c')](c)

Let us consider the procedure for the update cancel. By means of the semantic equations together with the definition of the constructs if ... then and delete, we obtain, for each s in the domain of S,

k[proc cancel(c) = if ¬∃s TAKES(s,c) then
                    delete OFFERED(c)](c) =

m[if ¬∃s TAKES(s,c) then delete OFFERED(c)] =

{(p,q) ∈ (UxU)/I[∃s TAKES(s,c)]p = False and

$$q = p[\{x \in C / I[\![OFFERED(x)]\!]p = True \text{ and } I[\![x]\!]p \neq I[\![c]\!]p\}/OFFERED]) \} \cup \{(p,q) \in (U \times U) / I[\![\exists s\ TAKES(s,c)]\!]p = True \text{ and } q = p\}$$

In view of the form of the expression above giving the meaning of procedure cancel, it is natural to divide this verification into two cases, according to the value of $I[\![\exists s\ TAKES(s,c)]\!]p$.

Case 1: $I[\![\exists s\ TAKES(s,c)]\!]p = False$

Then $(p,q) \in m[\![$ if $\neg \exists s\ TAKES(s,c)$ then delete $OFFERED(c)]\!]$
iff $q = p[\{x \in C / I[\![OFFERED(x)]\!]p = True$ and
$I[\![x]\!]p \neq I[\![c]\!]p\}/OFFERED]$

Thus $I[\![OFFERED(c)]\!]q = False = I[\![\exists s\ TAKES(s,c)]\!]p$.

Case 2: $I[\![\exists s\ TAKES(s,c)]\!]p = True$

Then $(p,q) \in m[\![$ if $\neg \exists s\ TAKES(s,c)$ then delete $OFFERED(c)]\!]$
iff $q = p$

Hence $I[\![OFFERED(c)]\!]q = I[\![OFFERED(c)]\!]p$

Let ri be the state denoted by the term ui(ui-1 (...(initiate)...)), for i = 0,...,n (hence rn-1 = p and rn = q). By the induction hypothesis, each equation P in A2 is valid in A when ó is valuated as ri and c is valuated as b, for i = 0,...,n-1. (We use A != P[b/c,ri/ó] to indicate this condition.)

Let us proceed in a backward direction to examine the various possibilities for each ui, for any b', b" ∈ C and t, t' ∈ T:

(1) if ui is initiate then, by equation 2,

A != (takes(s,c,initiate) = False)[t/s,b/c]

(2) if ui is offer then, by equation 5,

A != (takes(s,c,offer(c',ó)) = takes(s,c,ó))[t/s,b'/c',b/c,ri-1/ó]

(3) if ui is cancel then, by equation 8,

A != (takes(s,c,cancel(c',ó)) = takes(s,c,ó))[t/s,b'/c',b/c,ri-1/ó]

(4) if ui is enroll then, by equation 10

A != (takes(s,c,enroll(s,c,ó)) = offered(c,ó))[t/s,b/c,ri-1/ó]

and by equation (11), if t ≠ t' and b ≠ b':

A != (takes(s,c,enroll(s',c',ó)) =
takes(s,c,ó))[t/s,t'/s',b/c,b'/c',ri-1/ó]

(5) if ui is transfer then, by equations 13, 14, 15

A != ((takes(s,c,transfer(s',c',c",ó)) = True ->
offered(c,ó) = True V takes(s,c,ó)) = True)
[t/s,T'/s',b/c,b'/c',b"/c",ri-1/ó]

The backward process uses (1)-(5) repeatedly. In many cases we are simply led to examine a previous state, since the expression used says that c is offered after the application of an ui if it was offered at ri-1. However this process cannot reach initiate, where c would not be offered, contrarily to the condition of case 2. We can verify that the only way to fulfill this condition, rewritten as

A != (∃s takes(s,c,ó))[b/c,rn-1/ó]

is either by enrolling s in c or by transferring s to c, in any case in a state ri-1 where c is offered. Moreover, by equations 9 and 12, c will still be offered after any of the two operations is applied. Hence, we conclude that

(6) there exists j < n such that

A != (∃s (takes(s,c,ó) = True)[b/c,rj/ó] iff
A != (∃s (takes(s,c,ó) = True)[b/c,rn-1/ó]
and A != (offered(c,ó) = True)[b/c,rj/ó]

Let k be the maximum such j. So, we have that

A != (∃s takes(s,c,ó) = True)[b/c,rk/ó] and
A != (offered(c,ó) = True)[b/c,rk/ó].

Now, we can proceed in a forward direction to show that indeed

A != (offered(c,ó) = True)[b/c,rn-1/ó].

since the only way to reach from rj a state where c is no longer offered is by cancelling c, which fails as long as there is a student taking c (here we are using, among others, equation 6, the very equation that we are about to prove; this is legitimate because we are assuming by hypothesis its validity up to state rn-1).

Hence

I[OFFERED(c)]q = I[OFFERED(c)]p

This ends the inductive step for equation 6.

Proceeding similarly we can then verify that all the equations of the operations level are satisfied by our specification of the representation level.

6. Conclusions

In spite of marked differences in notation, the five formalisms discussed in this paper have a characteristic in common: they are all related to logic.

The one to one correspondence between db-predicate symbols (first level), query functions (second level) and relations names (third level) provided a certain uniformity that facilitated going through the different notations. This coincidence, although not a mandatory design decision, proved to be convenient. We note in passing that it argues for the "naturalness" of the relational model.

One of the more significant differences across the three levels of specification is the treatment of states. States are implicitly described by their properties at the information level. They are explicit parameters at the functions level. At the representation level they are defined in terms of the value of the entire collection of data base relations; each statement mentions only the relations that it affects. Intermediate states may be considered as an operational (machine-like) aspect of the representation level, resulting from the execution of single statements.

The justification for the third level of specification is to lead to implementation on the current machines, mostly conforming to a von Neumann architecture. Yet there have been attempts to achieve executable programs written in logical languages (e.g. PROLOG) or algebraic languages (e.g. OBJ). Also programming languages intended originally for specification may be given a compiler or interpreter (e.g. TAXIS). In many cases the intention is only to provide running specifications for testing purposes (prototyping), but future developments may lead to systems that perform efficiently in a production environment.

We believe that the discussion and the example substantiate the claim that each formalism plays indeed some relevant role in the formal specification of data bases, especially when used for the objective that originally motivated its proposal.

Bibliography

Logical formalism

- Fundamentals

[En] H. B. Enderton - "A mathematical introduction to logic" - Academic Press (1972).

[Ha] D. Harel - "First-order dynamic logic" - Lecture Notes In Computer Science, vol. 68 - Springer-Verlag (1979).

[MP] Z. Manna and A. Pnueli - "Temporal verification of concurrent programs" - in "The correctness problem in computer science" - R. S. Boyer and J. S. Moore (eds.) - Academic Press (1981) 215-273.

[RU] N. Rescher and A. Urquhart - "Temporal logic" - Springer (1971).

- Data base research

[BADW] A. Bolour, T. L. Anderson, L. J. Dekeyser and N. K. T. Wong - "The role of time in information processing - a survey" - ACM SIGMOD RECORD, 12, 3 (1982) 27-50.

[CCF] J. M. V. de Castilho, M. A. Casanova and A. L. Furtado - "A temporal framework for data base specification" - Proc. of the 8th International Conference on Very Large Data Bases (1982) 280-291.

[CF] M. A. Casanova and A. L. Furtado - "On the description of database transition constraints using temporal languages" - Advances in Database Theory, vol. II - H. Gallaire, J. Minker and J. M. Nicolas (eds.) - Plenum (to appear).

[GM] H. Gallaire and J. Minker (eds.) - "Logic and data bases" - Plenum Press (1978).

[GMN1] H. Gallaire, J. Minker and J. M. Nicolas (eds.) - "Advances in data base theory" - Plenum Press (1980).

[GMN2] H. Gallaire, J. Minker and J. M. Nicolas - "Logic and databases: an overview and survey" - Joint Report CGE/CERT/University of Maryland (1982).

[Ja] B. E. Jacobs - "On database logic" Journal of the ACM, 29, 2 (1982) 310-332.

[Li] W. Lipski - "On databases with incomplete information" - Journal of the ACM, 28, 1 (1981) 41-70.

[Ma] D. Maier - "The theory of relational databases" - Computer Science Press (1983).

[Se] A. Sernadas - "Temporal aspects of logical procedure definition" - Information Systems, 5 (1950) 167-187.

Algebraic formalism

- Fundamentals

[GTW] J. A. Goguen, J. W. Thatcher and E. G. Wagner - "An initial algebra approach to the specification, correctness and implementation of abstract data types" - in "Current trends in programming methodology - R. T. Yeh (ed.), vol. IV, Prentice-Hall (1978) 80-169.

[Gu] J. Guttag - "Abstract data types and the development of data structures" - Communicattions of the ACM, 21, 12 (1978).

[Pa] C. Pair - "Sur les modeles des types abstraits algebriques" - Seminaire d'informatique theoretique - Universite de Paris VI et VII (1980).

- Data base research

[BZ] M. L. Brodie and S. N. Zilles (eds.) - Proc. of the Workshop on Data Abstraction, Databases and Conceptual Modelling - SIGMOD Record, 11, 2 (1981).

[DMW] W. Dosch, G. Mascari and M. Wirsing - "On the algebraic specification of databases" - Proc. 8th International Conference on Very Large Data Bases (1982) 370-385.

[EKW] H. Ehrig, H. J. Kreowski and H. Weber - "Algebraic specification schemes for data base systems" - Proc. 4th International Conference on Very Large Data Bases (1978) 427-440.

[LMWW] P. C. Lockemann, H. C. Mayr, W. H. Weil and W. H. Wohlleber - "Data abstractions for data base systems" - ACM Transactions on Database Systems, 4, 1 (1979) 60-75.

[VF] P. A. S. Veloso and A. L. Furtado - "Stepwise construction of algebraic specifications" - Advances in Database Theory, vol. II - H. Gallaire, J. Minker and J. M. Nicolas (eds.) - Plenum (to appear).

Programming language formalism

- Fundamentals

[Ea] J. Earley - "Toward an understanding of data structures" - Communications of the ACM, 14, 10 (1971) 617-627.

[Sch] J. T. Schwartz - "Principles of specification language design with some observations concerning the utility of

s=ecification languages" - in "Algorithm specification" - R. Rustin (ed.) - Prentice-Hall (1972).

[Wei] P. Wegner - "The Vienna Definition Language" - ACM Computing Surveys, 4, 1 (1972) 5-63.

- Data base research

[CB] M. A. Casanova and P. A. Bernstein - "A formal system for reasoning about programs accessing a relational database" - ACM Transactions on Programming Languages and Systems, 2, 3 (1980) 386-414.

[MBW] J. Mylopoulos, P. A. Bernstein and H. K. T. Wong - "A ganguage facility for designing database-intensive applications" - ACM Transactions on Database Systems, 5, 2 (1980) 185-217.

[Ro] N. Roussopoulos - "CSDL: a conceptual schema definition language for the design of data base applications" - IEEE Transactions on Software Engineering, 5, 5 (1979) 481-496.

[SW] A. M. Schettini and J. Winkowski - "Towards a programming language for manipulating relational data bases" - in "Formal description of programming conccepts II" - D. Bjorner (ed.) - North-Holland (1983) 265-280.

Generative formalism

- Fundamentals

[CER] V. Claus, H. Ehrig and G. Rozenberg (eds.) - "Graph grammars and their application to computer science and biology" - Springer Verlag (1979).

[ENR] H. Ehrig, M. Nagl and G. Rozenberg (eds.) - "Graph-grammars and their application to computer science" - Springer Verlag (1983).

[He] W. Hesse - "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).

[Pe] J. E. L. Peck - "Two-level grammars in action" - Proc. 6th IFIP World Computer Congress (1974) 317-321.

- Data base research

[EK] H. Ehrig and H. J. Kreowski - "Applications of graph grammar theory to consistency, synchronization and scheduling in database systems" - Information Systems, 5 (1980) 225-238.

[FVC] A. L. Furtado, P. A. S. Veloso and M. A. Casanova - "A

grammatical approach to data bases" - Proc. 9th IFIP World Computer Congress (1983) 705-710.

[Ki] C. M. R. Kintala - "Attributed grammars for query language translations" - Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (1983) 137-148.

[LMP] H. Laine, O. Maanavilja and E. Peltola - "Grammatical data base model" - Information Systems, 4, 4 (1979) 257-267.

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

Denotational formalism

- Fundamentals

[BJ] D. Bjorner and C. B. Jones - "Formal specification and software development" - Prentice-Hall (1982).

[Sco] D. Scott - "Logic and programming languages" - Communications of the ACM, 20, 9 (1977) 634-641.

[Te] R. D. Tennent - "The denotational semantics of programming languages" - Communications of the ACM, 19, 8 (1976) 437-453.

- Data base research

[BL] D. Bjorner and H. H. Lovengreen - "Formalization of database systems and a formal definition of IMS" - Proc. 8th International Conference on Very Large Data Bases (1982) 334-347.

[LP] G. Louis and A. Pirotte - "A denotational definition of the semantics of DRC, a Domain Relational Calculus" - Proc. 8th International Conference on Very Large Data Bases (1982) 348-356.

[LS] W. Lammersdorf and J. W. Schmidt - "Semantic definition of Pascal R" - reports 73-t4, Univ. of Hamburg (1980).

[NO] E. J. Neuhold and T. Olnhoff - "The Vienna Development Method (VDM) and its use for the specification of a relational data base system" - Proc. 8th IFIP World Computer Congress (1980) 3-16.

Complementarity

- Fundamentals

[Do] J. E. Donahue - "Complementary definitions of programming language semantics" - Springer Verlag (1976).

[HL] C. A. R. Hoare and P. E. Lauer - "Consistent and complementary formal theories of the semantics of programming languages" - Acta Informatica, 3 (1974) 135-153.

[Pa] F. G. Pagan - "Formal specification of programming languages" - Prentice-Hall (1981).

[We2] P. Wegner - "Programming language semantics" - in "Formal semantics of programming llanguages" - R. Rustin(ed.) - Prentice-Hall (1972) 149-248.

- Data base research

[VCF] P. A. S. Veloso, J. M. V. de Castilho and A. L. Furtado - "Systematic derivation of complementary specifications" - Proc. 7th International Conference on Very Large Data Bases (1981).