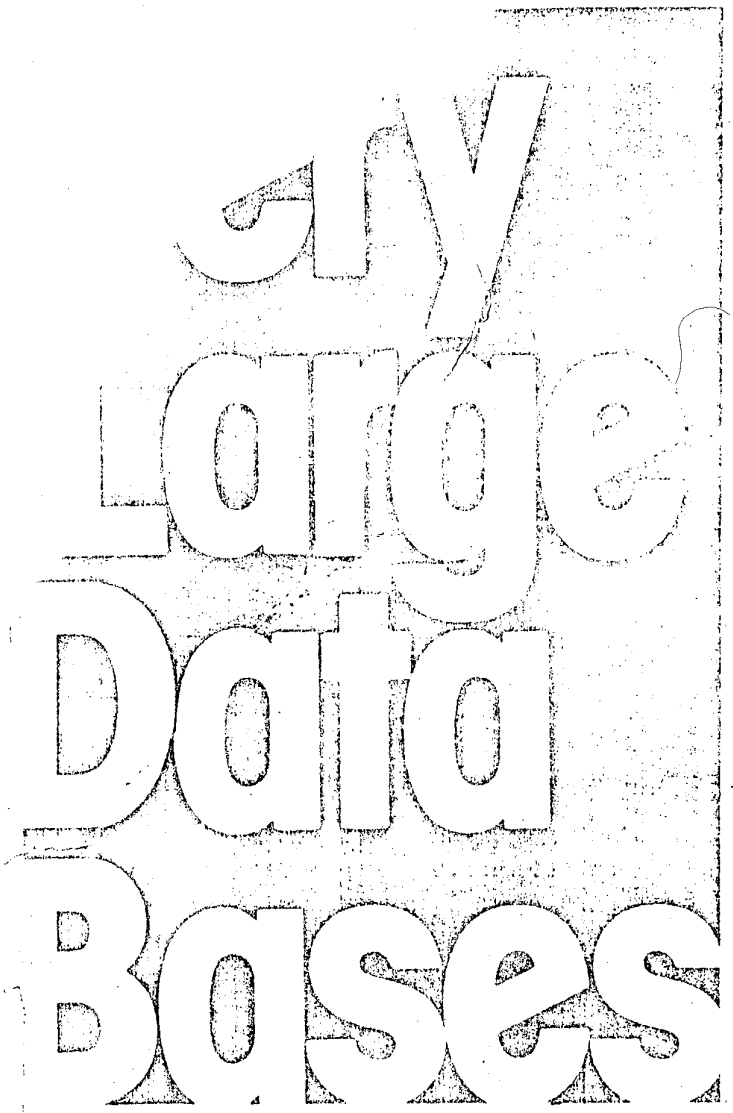


SEVENTH INTERNATIONAL CONFERENCE ON Very Large Data Bases CANNES FRANCE

SEPTEMBER 9-11, 1981

Sponsored by:

INRIA — Institut National de Recherche en Informatique et en Automatique

ACM — Association for Computing Machinery
SIGMOD — Management of Data
SIGBDP — Business Data Processing
SIGIR — Information Retrieval

IEEE — Institute of Electrical and Electronics Engineers
IEEE Computer Society

Support for the Conference has been provided by:

ADI	— Agence de l'Informatique
AFCET	— Association Française pour la Cybernétique Économique et Technique
AICA	— Associazione Italiana per il Calcolo Automatico
BCS	— British Computer Society
CII HB	— Compagnie CII Honeywell Bull
CNET	— Centre National d'Études des Télécommunications
DARPA	— Defense Advanced Research Projects Agency
EDF	— Electricité de France
USARDSG—UK	— European Research Office of the U.S. Army
	— French Ministry of Foreign Affairs
GI	— Gesellschaft für Informatik
IBM	— France
SIEMENS	— West Germany
SIF/VFI	— Société d'Informatique Fondamentale Vereniging voor Fundamenteel Informatika (Belgium)

ACM Order No. 471810
IEEE Catalog No. 81CH1701-2
Computer Society Order No. 371
Library of Congress No. 81-82788

IEEE COMPUTER SOCIETY PRESS

SYSTEMATIC DERIVATION OF COMPLEMENTARY SPECIFICATIONS

P.A.S. Veloso+, J.M.V. de Castilho*+ and A.L. Furtado+

+Pontifícia Universidade Católica do Rio de Janeiro, Brasil
* Universidade Federal do Rio Grande do Sul, Brasil

ABSTRACT

A methodology is proposed for the systematic derivation of a series of complementary specifications of a data base application. The starting point for this series is chosen so as to be obtainable without undue difficulty from an informal specification. Thereafter each formal specification is systematically derived from the preceding one. This multiplicity of specifications is further justified by their complementary nature. Having distinct aims, they jointly provide a multi-purpose, comprehensive characterization of the data base application. The formalisms employed can be categorized according to several criteria, including the usual definitional, denotational, operational characterizations of semantics. A simplified example is used throughout to illustrate the development.

1. INTRODUCTION

There are several possible ways of formally specifying a data base application, each one having its own advantages and disadvantages. So, it is of interest to be able to derive a particular formal specification from another one, which may have been constructed with another aim in mind. Also, the crucial and error-prone step of going from an informal specification to a formal one may be facilitated by an adequate choice of formalism for the formal specification. In this paper we start with an informal, natural language specification of a data base application and present a series of "complementary" formal specifications, in order to show how one can systematically derive one formal specification from another one.

Now, formal specifications are not praised for legibility or ease of understanding. One of the advantages of complementary specifications is exactly in aiding our intuitive understanding, another one being suitability for diverse purposes. These points have been stressed in connection with the definition of the semantics of programming languages[7,16] and in connection with abstract data type specifications[20,27,28]. Also, the ability to transform one formal specification into another by systematic means has been advocated as a useful approach to the process of program construction[1,3].

Let us assume that each state of a particular data base can be fully characterized by its observable properties and there are available a number of functions which map states into states.

A first question of practical interest presents itself:

- Question 1: For each state i, what other states are reachable from i through composition of functions?

Notice that Si, the set consisting of such states, is in general infinite and, therefore, we cannot exhibit it explicitly. A reasonable approach begins by characterizing each function by its preconditions and effects, i.e., respectively, by the values that certain predicates must have prior to the application of a function, and the alterations in the values of predicates resulting from the application. We can put this in the form of a finite set of axioms, and by their possibly repeated use we can determine what state j results from the application of some sequence (composition) f* of functions to the state i.

Given two states i and j, there may be more than one sequence f* mapping i into j. This leads to a second question of practical import, which somehow stands as a dual to the first one:

- Question 2: For each sequence f* of functions, what other sequences of functions will, when applied to the same (input) states, map them into the same (output) states?

Again, the set Sf* consisting of such sequences of functions is in general infinite. The axioms used in connection with question 1 are often not convenient for use directly in treating this second question. Let us begin here by considering some sequence g* such that, for any two states i and j, g* maps i into j if and only if f* does. This property clearly induces an equivalence relation R. Now, a reasonable approach to determine that R holds between two given sequences of functions would be to use a finite set of equations, capable of transforming one sequence into the other, possibly after repeated use, if and only if they are equivalent under R.

Besides providing convenient ways to handle our questions, axioms and equations lead to two complementary ways for the full specification of data bases. The former does so by defining the data base, where states are given by predicates. Since predicates will be regarded as our query operations (see the PLANNER-like features in section 4.2), we call this the query oriented approach. The latter corresponds to the algebraic specification and deals with sequences of functions. Since functions will serve as our update operations, this will be called the update oriented approach.

Each of the two forms of specification will be

CH1701-2/81/0000/0409$00.75 © 1981 IEEE

expanded so as to have either

1- the properties which implicity define the set of data base states (axioms in one case and equations in the other); or,

2- explicit representations for the states and either

(2.a) operations defined functionally; or

(2.b) operations defined by means of procedures on abstract machines.

This triple expansion follows the classification of semantics into (1) definitional, (2a) denotational and (2b) operational methods[2,7,16].

The discussion in the paper will be centred around a simplified example, presented in the next section. Section 3 will deal with query-oriented specifications illustrated by means of the example. In section 4 we continue our example by deriving three update-oriented specifications for it. Finally we conclude by comparing the various specifications and with comments on specification formalisms in general and their interrelationships.

2. INFORMAL SPECIFICATION

As an example of a (simplified) data base application we shall use the data base of an employment agency. Here, persons apply for positions, companies subscribe by offering positions, and companies hire candidates or fire employees. We impose the following constraints: a person may apply only once, thus becoming a candidate, losing this status when hired by a company but regaining it if fired; a company may subscribe several times, the positive number of offerings being added up; finally, only persons that are currently candidates may be hired, and only by companies having vacant positions. We shall have queries for checking whether a person is a candidate, whether a person works for a company, and whether a company still has some number of vacant positions. We assume the data base to be initialized to an empty state.

A first step towards giving a more precise specification might be simply to organize the above verbal information by operations. This increases readability and facilitates referencing, as shown in figure 1.

In doing so, we are conceiving updates as operations (with parameters) mapping states into states and queries as predicates (with parameters) indicating whether or not a state has the corresponding property. But the specification of each update or query is still couched in natural language. In other words, we are clarifying the syntax, but the semantics remains informal.

Updates:

- initag : initializes the data base to an "empty state";
- apply : a person x becomes a candidate (provided that x is not a candidate and x is not employed);
- subscribe : a company y offers n positions (increases the total number of vacant positions in y by n, if n > 0);
- hire : a company y hires a person x (provided that x is candidate and y has vacant positions; if so, then the number of vacant positions in y decreases by 1 and x ceases to be a candidate);
- fire : a company y fires a person x (provided that x works for y; if so then x becomes a candidate again and the number of vacant positions in y is increased by 1).

Queries:

- iscand : tests whether a person x is a candidate (holds if x has applied and is not employed);
- haspos : tests whether a company y has n vacant positions (holds if y has exactly n vacant positions, and is false if y has not yet subscribed);
- worksfor : tests whether a person x works for a company y (holds if x has been hired and not fired by y).

Figure 1: Informal specification

3. QUERY-ORIENTED SPECIFICATIONS

The so-called state-space approach to problem solving is frequently used in artificial intelligence[18,26]. In the data base terminology its basic idea may be described as follows: the data base, starting from the initial empty state, evolves through several states obtained by the application of updates, and one can identify the current state of the data base by means of queries. The latter assumption - that one can identify the current state simply by interrogating it with no need to change it - is of fundamental importance. It is generally satisfied in data base applications, but not by an arbitrary data type which may be poor in queries, such as, for example, a pushdown stack.

3.1. Axiomatic specification

The above ideas suggest describing each update by means of its effect on queries. This is somewhat similar to the Floyd-Hoare axiomatic description of the semantics of programming language constructs by means of assertions[13,22]. We can describe the effect of each update o by its pre- and post-conditions, in the form

$$\{pre(s)\}\ t:=o(s)\ \{post(s,t)\}$$

meaning that if in state s pre(s) holds then in state t = o(s) post(s,t) will hold. (This method is critically evaluated elsewhere[11]).

Thus, we may describe an update such as hire as follows

if in state s:
. x is a candidate
. company y has positions(a positive number)

then in state t = hire(x,y,s):
- x is no longer a candidate
- the number of vacant positions in y is decreased by 1;
- x works for y.

This is basically a more structured presentation of still informal descriptions. However we have made explicit the assertions built from queries. These assertions can now be translated into the first-order predicate calculus notation. As a result we obtain:

if in state s:
- iscand (x,s)
- ∃n(haspos(y,n) ∧ n > o)

then in state t = hire(x,y,s):
- ¬ iscand (x,t)
- haspos(y,n,s) → haspos(y,n-1,t)∧¬haspos(y,n,t)
- worksfor(x,y,t)

(Notice that haspos(y,n,s) is expected to hold for at most one n. This integrity constraint will be a consequence due to underlying assumptions to be explicited later on). We can translate this information into the predicate calculus formula:

∀s∀x∀y∀n ([iscand (x,s) ∧ haspos(y,n) ∧ n > o] →
→[¬iscand(x, hire(x,y,s)) ∧
∧haspos(y, n-1, hire(x,y,s)) ∧
∧¬haspos(y,n, hire(x,y,s)) ∧
∧worksfor (x,y,hire(x,y,s))])

which is the origin of axioms 8-11 in Figure 2. Similarly the first three axioms correspond to the initial empty state. Axioms 16 and 17 introduce nopos and empco as Skolem functions[32].

Thus, we obtain the axiomatic specification for our example given in Figure 2. Clearly, the axioms do not have to be regarded as collected in groups pertaining to certain updates. Notice that states are referred to only by means of the variable s.

1 ¬iscand(x,initag)

2 ¬haspos(y,n,initag)

3 ¬worksfor(x,y,initag)

4 ¬iscand(x,s) ∧ ¬worksfor(x,empco(x,s),s) →
→ iscand(x,apply(x,s))

5 v>0→[haspos(y,n,s) → haspos(y,n+v,subscr(y,v,s))]

6 v>0→[haspos(y,n,s) → ¬haspos(y,n,subscr(y,v,s))]

7 v>0→[¬haspos(y,nopos(y,s),s) →
→ haspos(y,v,subscr(y,v,s))]

8 iscand(x,s) ∧ haspos(y,n,s) ∧ n>0 →
→ ¬iscand(x, hire(x,y,s))

9 iscand(x,s) ∧ haspos(y,n,s) ∧ n>0 →
→ haspos(y, n-1, hire(x,y,s))

10 iscand(x,s) ∧ haspos(y,n,s)∧ n>0 →
→ ¬haspos(y,n,hire(x,y,s))

11 iscand(x,s) ∧ haspos(y,n,s) ∧ n>0 →
→ worksfor(x,y,hire(x,y,s))

12 worksfor(x,y,s) → iscand(x,fire(x,y,s))

13 worksfor(x,y,s) → [haspos(y,n,s) →
→ haspos(y,n+1,fire(x,y,s))]

14 worksfor(x,y,s) → [haspos(y,n,s) →
→ ¬haspos(y,n, fire(x.y.s))]

15 worksfor(x,y,s) → ¬worksfor(x,y,fire(x,y,s))

16 haspos(y,n,s) → haspos(y,nopos(y,s),s)

17 worksfor(x,y,s) → worksfor (x,empco(x,s),s)

Figure 2: Axioms

This specification is supposed to describe completely the (effect of each update on the) data base. However, it does not appear to settle explicity the following questions:

(i) Can x somehow become a candidate other than by applying (or by being fired)?

(ii) Does x's applying influence in any way, say, the number of vacant positions in a company?

(iii) What happens if x applies despite already being a candidate?

In fact, certain assumptions underlie the whole process of state-space specification (with (d) below corresponding to identification by queries as mentioned previously and each one of (a), (b), (c) settling questions (i), (ii), (iii) above, respectively):

a. Only-if assumption
The axioms express the only conditions under which an assertion holds.

b. Frame assumption
If a predicate is not explicity indicated in the axioms as being affected by an update, then it is not affected.

c. Non-applicability assumption
If the antecedent of an axiom fails, then the property in the consequent is not affected by the execution of the corresponding update.

d. Observability assumption
Two states s and t are equal if (and only if) for every predicate (the other possible arguments regarded as parameters) p(s) holds iff p(t) holds.

Two interesting consequences of the above assumptions are:

- negation as failure[6], from (a);
- o(s) = s whenever the precondition for o fails to hold at s, from (c) and (d).

It is worth remarking that the axioms are expressible as Horn clauses, extended to accomodate negation as failure. This ensures the existence of minimal models[34], a fact to be exploited in the sequel. Also, the frame assumption could be formalized into frame axioms[19].

From the axiomatic specification for our example, together with the underlying assumptions, we

can see that certain invariants (static integrity constraints, in data base terminology) are preserved at every state s (reachable from initag):

(a) worksfor(x,y,s) → ¬iscand(x,s)
(no person is both a candidate and an employee)

(b) haspos(y,v,s) ∧ haspos(y,w,s) → v=w
(the number of vacant positions in company y depends only on y)

(c) worksfor(x,y,s) ∧ worksfor(x,z,s) → y=z
(each person works for at most one company)

(d) worksfor(x,y,s) → ∃v(haspos(y,v,s))
(any company with employees is registered)

3.2. Predicative normal form

The state-space approach relies on the observability assumption: states are identifiable by queries. One can bring this explicity to the foreground by identifying each state with the queries that hold true of it.

In our employment agency example, consider, for instance, the state attained by the data base when persons e1, e2, e3 and e4 have applied, companies c1 and c2 have offered, respectively, 3 and 2 positions, e1 and e2 have been hired by c1, e3 has been hired by c2 and subsequently c1 fired e2. This state can be described by the fact that in it the following assertions hold

iscand(e2,s) , iscand(e4,s) ,
haspos(c1,2,s) , haspos(c2,1,s) ,
worksfor(e1,c1,s) , worksfor(e3,c2,s) (1)

This set of assertions is to be interpreted as a set of clauses in the sense of automated theorem proving[5,29]. Then our only-if assumption becomes the minimality assumption: a predicate holds at a state if and only if you can deduce the corresponding assertion[23]. An equivalent way of stating this is as follows. Consider the set of clauses (note the change of language: the state variable is absent)

Iscand(e2), Iscand(e4), Haspos(c1,2) (2)

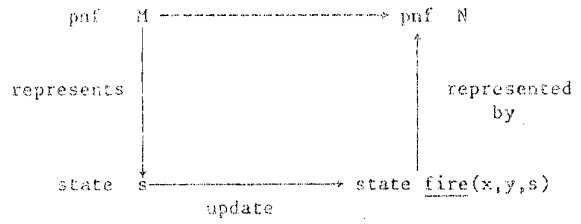
Haspos(c2,1), Worksfor(e1,c1), Worksfor(e3,c2)

State s referred to above satisfies (is a model of) the above clauses. It is completely characterized by these, if we understand (only-if assumption) that no other clauses hold.

In general, each state s is characterized by what we call its predicative normal form, which is the set of all positive ground literals holding at s. Notice the contrast with the axiomatic specification. There we use state variables to refer to elements of a sort (the state space), about at the same level of abstraction as, say, the sort of companies. Here, each state is thought of as a (relational) structure which is to be a model of certain clauses. This shift of viewpoint - from, e.g.,"iscand(e2,s) holds" to "s satisfies Iscand(e2)"is somewhat similar to the one from Hoare-logic to dynamic logic[14] in programming language semantics. In this approach (already used in connection with data bases[4]), an update is still an operation mapping states to states: only now it is viewed formally as transforming a structure (which is what a state is now) into another[24].

Now, a state is viewed as a structure defined by its predicative normal form (its positive diagram, in logical terminology[32]). The definition of queries is immediate: the result of a query q(s) is true iff s satisfies Q. The effect of an update consists of additions to and/or deletions from the predicative normal form.

The process of obtaining the pnf (predicative normal form) specification from the axiomatic one can be illustrated with the case of the update fire. We want



Axioms 12-15 suggest taking as pre-condition for this update: Worksfor(x,y) is satisfied at s. Then we can deduce that the resulting state should satisfy

. Iscand(x);

. Haspos(y,n+1) ∧¬Haspos(y,n) , whenever
Haspos(y,n) ∈ M;

. ¬Worksfor(x,y)

We have above an input-output specification for the update fire, which can be easily translated into the one in Figure 3 (which uses the primitive predicate in (for belonging) of the type set-of). This figure presents the pnf specification obtained for our example by proceeding similarly with the other updates and queries.

States:

sort pnf = set-of (ground-clause)
where ground-clause =
Iscand(x) ; x: person
Haspos(y,n) ; y: company, n: natural
Worksfor(x,y) ; x: person, y: company

Updates:

N := initag
pre: none (initial)
post: Iscand(x) ∉ N
Haspos(y,n) ∉ N
Worksfor(x,y) ∉ N

N := apply (x,M)
pre: Iscand(x) ∈ M
Worksfor(x,y) ∉ M
post: Iscand(x) in N

N := subscr(y,v,M)
pre: v > 0
post: (∀n ¬Haspos(y,n) in M) → Haspos(y,v) in N
∀n (Haspos(y,n) in M → Haspos(y,n+v) in N ∧
∧ Haspos(y,n) ∉ N)

N := hire(x,y,M)
pre: Iscand(x) in M
∃n (n > 0 ∧ Haspos(y,n) in M)

post: Iscand(x) $\not\in$ N
$\forall$n (Haspos(y,n) in M $\rightarrow$ Haspos(y,n-1) in N $\wedge$
$\wedge$ Haspos(y,n) $\not\in$ N)
Worksfor(x,y) in N

N : = fire(x,y,M)
pre: Worksfor(x,y) in M
post: Iscand(x) in N
$\forall$n (Haspos(y,n) in M $\rightarrow$ Haspos(y,n+1) in N $\wedge$
$\wedge$ Haspos(y,n) $\not\in$ N)
Worksfor(x,y) $\not\in$ N

Queries:

iscand(x,M) = T $\leftrightarrow$ Iscand(x) in M
haspos(y,n,M) = T $\leftrightarrow$ Haspos(y,n) in M
worksfor(x,y,M) = T $\leftrightarrow$ Worksfor(x,y) in M

Figure 3: pnf specification

The pnf specification has an explicit representation for the states, so the observability assumption is already built in. We still need the other 3 assumptions, which imply that only the states satisfying the integrity constraints of section 3.1 can be generated by updates.

3.3 Characteristic-set model

In the preceding section, a state was viewed as a model of a set of clauses. We can explicity describe such a model as a relational structure, with relations consisting of the tuples formed by the parameters of each query which has the value T at this state.

In the example of section 3.2, the state s defined by the pnf (2) can be described as consisting of the following three relations, called its characteristic sets

$$I = \{e2, e4\}$$
$$H = \{(c1, 2), (c2, 1)\} \qquad (3)$$
$$W = \{(e1, c1), (e3, c2)\}$$

The cs (characteristic set) specification of our example agency appears in Figure 4, where besides the notations Dom and Im for domain and image of a relation, respectively, we also employ H(y/m) to denote the (partial) function obtained from H by redefining its value at y to be m.

Domain

state = {S = (I,H,W)/I $\subseteq$ person,
H $\subseteq$ company $\times$ natural,
W $\subseteq$ person $\times$ company : such that
(a) I $\cap$ Dom W = $\emptyset$
(b) H is a partial map
(c) W is a partial map
(d) Im W $\subseteq$ Dom H }

Updates

initag = ($\emptyset$, $\emptyset$, $\emptyset$)
apply(x,S) =
= (I$\cup${x}, H, W) if x$\notin$I and x$\notin$DomW
= S otherwise
subscr(y, v, S) =
= (I, H(y/H(y)+1),W) if v > 0 and y$\in$DomH
= (I, H$\cup${(y,v)},W) if v > 0 and y$\in$DomH
= S otherwise
hire (x,y,S) =
= (I-{x},H(y/H(y)-1), W$\cup${(x,y)}) if
x$\in$I, y$\in$DomH, H(y)>0
= S otherwise
fire(x,y,S) =
= (I$\cup${x},H(y/H(y)+1), W-{(x,y)}) if (x,y) $\in$ W
= S otherwise

Queries

$$\text{iscand}(x,S) = \begin{cases} T & \text{if } x \in I \\ F & \text{otherwise} \end{cases}$$

$$\text{haspos}(y,n,S) = \begin{cases} T & \text{if } (y,n) \in H \\ F & \text{otherwise} \end{cases}$$

$$\text{worksfor}(x,y,S) = \begin{cases} T & \text{if } (x,y) \in W \\ F & \text{otherwise} \end{cases}$$

Figure 4: cs specification

This specification describes explicitly the state diagram by defining (a representation for)the states and defining completely the functions denoting updates and the characteristic functions of the predicates denoting queries. In particular, we can discard the underlying assumptions. (Some simple set-theoretical concepts still remain.)

Also important is the fact that this specification can be obtained fairly systematically from the one in section 3.2 by means of the underlying assumptions of the latter. First, notice that the domain of states consists exactly of the states satisfying the integrity constraints. We shall use the simple case of the update apply to illustrate the process of obtaining a cs specification from the corresponding pnf specification. First, the conditions x $\notin$ I and x $\notin$ DomW are immediate set-theoretical translations of the pre-conditions

Iscand(x) $\not\in$ M and Worksfor(x,y) $\not\in$ M

(Notice that y is not a parameter of apply). The non-applicability assumption then gives the "otherwise" case. In case the pre-conditions hold, Figure 3 gives the post-condition Iscand(x) in M, which translates into x $\in$I', but does not suffice to specify (I', H', W') = apply(x,S). The only-if assumption says that I' should be as small as possible with x $\in$I' and, by the frame assumption, including I. Hence I' = I $\cup$ {x}. The frame assumption forms the basis for the definitions of H' and W'.

3.4. Operational specifications

Both the pnf and the cs specifications are based on explicit representations for the states. The operations of the former were presented by means of input-output specifications, in the latter we defined them explicitly. (We could have chosen to do otherwise.)

In any case, states are represented as sets and the effects of the updates are to insert and/or remove certain elements. Such an operational view can be realized through an implementation on a set-theoretical abstract machine[15]. This would be easy to translate into executable procedures in a convenient programming language (cf. section 5).

4. UPDATE-ORIENTED SPECIFICATIONS

As already mentioned, the data base, starting at the initial empty state (initag in the example), attains other states by means of applications of updates. So far, in the query-oriented specifications, states have been described by their properties, each state being characterized by its possible answers to the queries. Now we shall switch to description by means of updates.

In order to visualize what these dual standpoints mean, it is useful to imagine the state-space as a graph whose nodes correspond to states and have their characteristic sets as labels, and whose edges are labelled with the single updates corresponding to one-step transitions between states. What we propose to do now is to denote a state s by some chosen composition of updates (sequence of edge lables) leading from the initial state to s.

4.1. Canonical term algebra

Within the state-space graph, choosing for each state s a sequence of updates leading initag to s amounts to determining a spanning tree, rooted at the initial state, over the whole graph. There may be several such spanning trees and a judicious choice can be very helpful in constructing (or deriving) such term-based specifications.

A term is a well-formed (i.e. syntactically correct) sequence of updates. However some syntactically distinct terms denote the same state; for instance, both terms

apply(e2,subscr(c1,3,apply(e1,initag)))
subscr(c1,3,apply(e2,apply(e1,initag)))

denote the same state, as do many others.

Another aspect is the fact that some updates require the previous application of others if there is to be any net effect, e.g. hire(x,y,s) = s unless the preconditions for hire are satisfied. This requirement establishes a partial order among updates which have at least one common argument, except for initag, which is always the first of all. The diagram showing this partial order is sketched in Figure 5.

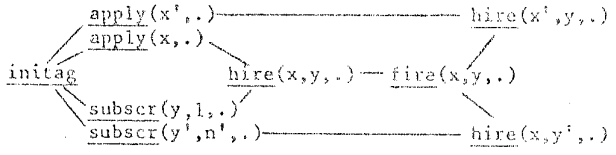


Figure 5: Partial order of updates

This partial order tells us which updates will generate new states if applied after a given sequence. We are still free to choose the ordering of unrelated updates, as for apply(x,.) and apply(x',.), with $x \neq x'$. We would like to have a unique term to represent each attainable state.

One way to obtain such representatives is as follows. The attainable states are exactly those represented in the domain of states of the characteristic set model (c.f. Figure 4). So, let us consider an attainable state represented by

$I = \{p_1, \ldots p_m\}$
$H = \{(c_1,v_1),\ldots,(c_k,v_k)\}$
$W = \{(q_1,c_{i1}),\ldots,(q_n,c_{in})\}$

and notice that, having in mind the characteristic-set specification, one way to transform initag into s is the following

- first all the p_i's and the q_j's apply;
- then each c_i subscr (v_i+w_i) positions, where w_i = the number of q_j's with $(q_j,c_i) \in W$;
- finally each q_j is hired by the corresponding c_{ij};

the relative order within each step being irrelevant.

We have already made some choices, like all apply's before all subscr's. We may complete this to a total order by ordering various occurrences of the same update lexicographically according to their arguments. Thus we arrive at a unique representative for state s. For instance, the canonical term for state s described by (3) in section 3.3, would be

hire(e1,c1,hire(e3,c2,subscr(c1,3,
subscr(c2,2,apply(e1,apply(e2,apply(e3,
apply(e4,initag)))))))))

Notice that all subterms of this term, e.g. subscr (c2,2,...,initag))))), apply(e4,initag), are themselves canonical terms, which represent intermediate states on the way from initag to the state s. So, it is natural to define the result of updating, say

apply(e1,apply(e3,apply(e4,initag)))

by c1 subscribing 3 positions to be

subscr(c1,3,apply(e1,apply(e3,apply(e4,initag))))

Thus, we have a one-to-one correspondence between states and canonical terms, the latter having the property that whenever a composite term is canonical then so are all of its arguments. We can take advantage of this fact to define the updates so that whenever o(t1,...,tn) is a canonical term then the result of applying o to t1,...,tn is o(t1,...,tn). If we do so we obtain a canonical term algebra[10], which is a useful tool for constructing abstract data type specifications[27].

A complete description of the canonical form chosen for our example agency appears in the specification of the domain state in Figure 6, where we write the terms with capital letters to aid in distinguishing them from the operations proper. It is important to notice that with this information we can derive the definitions of the operations on the canonical term algebra from some other specification, say characteristic-sets. In order to find the value of operation op on a canonical term can-arg, we use the given specification to discover

(1) the state st-arg represented by can-arg;
(2) the state st-res = st-op(st-arg);
(3) the canonical term can-res representing st-res;

according to the following diagram

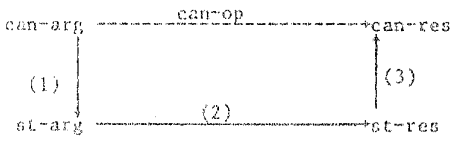


Figure 6 shows the definition of the updates and queries on the canonical term algebra. Notice that they involve symbolic manipulations of the canonical terms

Domain

state = {c=HIRE(z,w,...,SUBSCRIBE(y,n,...,APPLY(x,...,INITAG..)..)..)/

x,z ∈ person, y,w ∈ company, n ∈ natural;

such that

- each x occurs at most once as an argument of an APPLY;
- each z occurs at most once as an argument of a HIRE;
- each y occurs at most once as an argument of a SUBSCRIBE;
- each argument z of a HIRE is an argument x of some APPLY;
- each argument w of a HIRE is an argument y of some SUBSCRIBE;
- each argument n of SUBSCRIBE is positive;
- the number of HIRE's with argument w=y is at most the argument n in SUBSCRIBE(y,n,...)
- updates of the same type are lexicographically ordered, according to their first argument.}

Updates:/

initag = the term INITAG

apply(x,c) = the term c with APPLY(x,...) inserted in its proper place according to the order, if APPLY(x,...) does not occur in c;
= c, otherwise.

subscr(y,v,c) = the term c with SUBSCRIBE(y,v,...) inserted into its proper place, if c contains no occurrence of SUBSCRIBE (y,..,...) and v>0;
= the term c with SUBSCRIBE(y,n,...) replaced by SUBSCRIBE(y,n+v,...), if SUBSCRIBE(y,..,...) occurs in c and v > 0;
= c, otherwise.

hire(x,y,c) = the term c with HIRE(x,y,...) inserted in its proper place, if
(i) APPLY(x,...)occurs in c;
(ii) HIRE(x,y,...)does not occur in c;
(iii)SUBSCRIBE(y,n,...)occurs in c with n greater than the number of HIRE (..,y,...)'s already occurring in c;
= c , otherwise.

fire(x,y,c) = the term obtained by the removal of HIRE(x,y,...) from c, if it does occur;
= c, otherwise

Queries:

iscand(x,c) = T if APPLY(x,...)occurs in c and no HIRE(x,..,...)occurs in c;
= F, otherwise.

haspos(y,n,c) = T if c contains an occurrence of SUBSCRIBE(y,m,...) and (m-n) occurrences of HIRE(..,y,...);
= F otherwise.

worksfor(x,y,c) = T if HIRE(x,y,...)occurs in c
= F otherwise.

Figure 6: Canonical term algebra

4.2. Procedural specification

The canonical forms give a one-to-one correspondence between states and terms denoting them. Thus, if we want to describe the effect of an operation on states we may as well describe how it affects the corresponding canonical terms. This is the idea behind specifying the data base by means of a canonical term algebra, as in Figure 6.

A procedural specification consists of symbol-manipulating procedures, which implement the operations of the canonical term algebra[8]. Here we write these procedures using a pattern-matching construct and the PLANNER-like features "?", which stands for an arbitrary valid value of an argument and "?<variable>", which assigns the value found to <variable>. The structure of the procedure bodies we use consists of an optional pre-condition testing followed by a case-like pattern-matching command, which recursively scans the term.

A procedural specification for our example agency appears in Figure 7 where we use ">" to denote the lexicographic order of the parameters. As an example, consider the execution of

apply(e2,HIRE[e1|c1|SUBSCRIBE[c1|3|APPLY[e1|INITAG]]]).

The first step consists of testing the pre-conditions

iscand(e2,HIRE[e1|c1|SUBSCRIBE[c1|3|APPLY[e1|INITAG]]])
⇒iscand(e2,SUBSCRIBE[c1|3|APPLY[e1|INITAG]])(as e2≠e1)
⇒ ... ⇒ iscand(e2,INITAG) ⇒ false ; and

worksfor(e2,?,HIRE[e1|c1|SUBSCRIBE[c1|3|APPLY[e1|INITAG]]])
⇒ worksfor(e2,?,SUBSCRIBE[c1|3|APPLY[e1|INITAG]])
(as e2 ≠ e1)
⇒ ... ⇒ worksfor(e2,INITAG) ⇒ false

Having obtained ¬(¬false ∧ ¬false) the execution proceeds with match as follows

apply(e2,HIRE[e1|c1|SUBSCRIBE[c1|3|APPLY[e1|INITAG]]])
⇒HIRE[e1|c1|apply(e2,SUBSCRIBE[c1|3|APPLY[e1|INITAG]])]
⇒HIRE[e1|c1|SUBSCRIBE[c1|3|apply(e2,APPLY[e1|INITAG])]]
⇒HIRE[e1|c1|SUBSCRIBE[c1|3|APPLY[e1|apply(e2,INITAG)]]]
(as e2 > e1)
⇒HIRE[e1|c1|SUBSCRIBE[c1|3|APPLY[e1|APPLY[e2|INITAG]]]]

Two important points are worth mentioning in connection with such procedural specifications. First, they consist of - generally straightforward - translations of the descriptions of operations of the canonical term algebra. Second, the result of any such procedure (which always terminates) is a canonical term. As a consequence we have a method to convert any term into its canonical form, namely by inter-

preting the operation symbols in the term as procedure calls, working inside out, since the argument s must always be a canonical term. Notice, in particular, that this will leave canonical terms fixed and thus we may view the set of canonical terms as the least fixed-point of the transformation associated with this set of mutually recursive procedures.

In brief, a procedural specification is a representation of the cta on the type terms. The underlying assumption is the semantics of the PLANNER-like constructs. Also, notice that a procedural specification can be viewed as a deterministic implementation of a system of term-rewriting rules. This system has the properties of finite termination and confluence and its irreducible terms are exactly the canonical terms[17].

States:

```
sort agdb = constr-term
  where constr-term =
    INITAG|
    APPLY[x|constr-term]; x:person|
    SUBSCRIBE[y|n|constr-term]; y:company,n:natural|
    HIRE[x|y|constr-term];x:person,y:company.
```

Updates:

```
op initag():agdb
   ⇒INITAG
endop

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

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

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

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

Queries:

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

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

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

Figure 7: Procedural specification

4.3. Algebraic specification

Algebraic specifications have been used as abstract, representation-independent descriptions of data types[10,12,25].

Algebraic specifications contrast with the other specifications where the effect of each operation can be defined independently of the other operations. Basically, an algebraic specification tells us explicitly which sequences of operations have the same effect. Despite this marked difference, algebraic specifications may be regarded as somewhat similar to procedural ones in that an equational specification can be viewed as a term-rewriting system, where each equation gives rise to two rewriting rules, one for each "direction" in the equation[17]. Besides the directional difference, there is another important point concerning procedural versus algebraic specifications. In a procedural specification we have ordered statements to be executed and we assume the parameters of each procedure to be already in canonical form. On the other hand an algebraic specification consists of an unordered set of (conditional) equations which are to hold for any assignment of values to the variables.

Keeping in mind these differences it is possible to translate a procedural specification into a corresponding algebraic one. The equations obtained should be sufficient to ascertain whether any two terms are representable by the same canonical term[27].

We shall illustrate this process by showing the steps involved in translating the procedures of Figure 7 into corresponding equations.

The language we will use to write the set of equations is slightly different from the language used by[13]. We have logical and natural with their usual operations. Instead of the if-then-else construct, we will use the logical conditional → (cf Thatcher et al[33]) and to shorten the description of each equation, we will define as new predicates the pre-conditions for each update operation, which introduce negations and existential quantifiers (cf Figure 8).

The extraction of the "equations" from the procedures will be illustrated with the simple case of apply. First, the body of apply(x,s) (cf Figure 7) can be rewritten as follows, replacing < by ≠ (see below):

```
apply(x,s) =
= if ¬pcapply(x,s) then s
   else case
      s = INITAG ⇒ APPLY[x|s]
      s = APPLY[z|t] ⇒ if x≠z
            then APPLY[z|apply(x,t)]
            else APPLY[x|s]
      s = SUBSCRIBE[w|m|t] ⇒ SUBSCRIBE[w|m|apply(x,t)]
      s = HIRE[z|w|t] ⇒ HIRE[z|w|apply(x,t)]
```

This may be translated, by replacing the if-then-else and the case by →'s, as

```
¬pcapply(x,s) → apply(x,s) = s
pcapply(x,s) → (s=INITAG → apply(x,s)= APPLY[x|s]) ∧
                          ⋮
      ∧ (s=HIRE[z|w|t] → apply(x,s) = ... )
```

Now, the capital-letter terms are canonical ones and we may replace them by the corresponding small-letter terms only if their associated pre-conditions hold. Thus, we can dismember the second formula above into the four below

```
pcapply(x,s) → (s=initag → apply(x,s) = apply(x,s))

pcapply(x,s) → (s=apply(z,t) ∧ pcapply(z,t) →
        → [x≠z → (pcapply(z,apply(x,t)) →
            → apply(x,s)=apply(z,apply(x,t)))] ∧
        ∧ [x=z → apply(x,s) = apply(x,s)]

pcapply(x,s) → (s=subscr(w,m,t) ∧ pcsubscr(w,m,t) ∧
        ∧ pcsubscr(w,m,apply(x,t)) →
            → apply(x,s)=subscr(w,m,apply(x,t))

pcapply(x,s) → (s=hire(z,w,t) ∧ pchire(z,w,t) ∧
        ∧ pchire(z,w,apply(x,t)) →
            → apply(x,s)=hire(z,w,apply(x,t))
```

Now, if we further dismember these formulas we see that some, like the first one, have as consequent apply(x,s)=apply(x,s). Disregarding these and actually performing the substitutions allowed by the equality we arrive at the formulas 2, 3 and 4 of Figure 8.

The general method for extracting equations from procedures can be outlined as follows:

- for each procedure, and
  for each output condition
  build an axiom of the form
  condition → leftside = rightside
  where:
  condition is a description of the output condition for the procedure, presented as an expression whose result is a logical value, and
  leftside is quite similar to the actual procedure call with the proper value for argument s, in small letters;
  rightside is the output value corresponding to this actual procedure call, written in small letters.

When extracting equations from procedures, one must take some care. Identities will eventually be constructed. Atomic formulas corresponding to lexicographic ordering should be included in the convenient condition part with a ≠ instead of > or <, because in the equations this order is irrelevant.

Figure 8 shows the equations, where x, z are variables of sort person ; y, w are variables of sort company ; s, t are variables of sort agdb ; m, n are variables of sort natural .

The derivation of some equations (23-31 in Figure 8) does not follow completely the general method described above. The reason is that in the procedures we are always dealing with canonical terms, and sometimes we use this fact when constructing the procedure bodies.

The underlying assumptions are

- every state is denoted by a variable-free term;
- two variable-free terms t and t' denote the same state iff we can derive t = t' from the specification

Preconditions for update operations:

pcapply(x,s) ↔ ¬iscand(x,s) ∧ ¬∃y(worksfor(x,y,s))
pcsubscr(y,m,s) ↔ m > 0
pchire(x,y,s) ↔ iscand(x,s) ∧ ∃m(haspos(y,m,s) ∧ m > 0)
pcfire(x,y,s) ↔ worksfor(x,y,s)

Equations

1 ¬pcapply(x,s) → apply(x,s)=s

2 pcapply(x,apply(z,t)) ∧ pcapply(z,t) ∧ x≠z →
apply(x,apply(z,t)) = apply(z,apply(x,t))

3 pcapply(x,subscr(w,m,t)) ∧ pcsubscr(w,m,t) →
apply(x,subscr(w,m,t))=subscr(w,m,apply(x,t))

4 pcapply(x,hire(z,w,t)) ∧ pchire(z,w,t) →
apply(x,hire(z,w,t))=hire(z,w,apply(x,t))

5 ¬pcsubscr(y,m,s) → subscr(y,m,s) = s

6 pcsubscr(y,m,subscr(w,n,t)) ∧ pcsubscr(w,n,t) ∧ y=w →
subscr(y,m,subscr(w,n,t))=subscr(y,n+m,t)

7 pcsubscr(y,m,subscr(w,n,t)) ∧ pcsubscr(w,n,t) ∧ y≠w →
subscr(y,m,subscr(w,n,t)) =
subscr(w,n,subscr(y,m,t))

8 pcsubscr(y,m,hire(z,w,t)) ∧ pchire(z,w,t) →
subscr(y,m,hire(z,w,t))=hire(z,w,subscr(y,m,t))

9 ¬pchire(x,y,s) → hire(x,y,s) = s

10 pchire(x,y,hire(z,w,t)) ∧ pchire(z,w,t) ∧ y≠w →
hire(x,y,hire(z,w,t))=hire(z,w,hire(x,y,t))

11 ¬pcfire(x,y,s) → fire(x,y,s) = s

12 pcfire(x,y,hire(z,w,t)) ∧ pchire(z,w,t) ∧ x≠z →
fire(x,y,hire(z,w,t))=hire(z,w,fire(x,y,t))

13 pcfire(x,y,hire(z,w,t)) ∧ pchire(z,w,t) ∧ x=z →
fire(x,y,hire(z,w,t)) = t

14 pchire(z,w,t) ∧ x=z →
iscand(x,hire(z,w,t)) = false

15 pchire(z,w,t) ∧ x≠z →
iscand(x,hire(z,w,t)) = iscand(x,t)

16 pcsubscr(w,m,t) →
iscand(x,subscr(w,m,t)) = iscand(x,t)

17 pcapply(z,t) ∧ x=z →
iscand(x,apply(z,t)) = true

18 pcapply(z,t) ∧ x≠z →
iscand(x,apply(z,t)) = iscand(x,t)

19 iscand(x,initag) = false

20 pchire(z,w,t) ∧ w=y →
haspos(y,m,hire(z,w,t)) = haspos(y,m+1,t)

21 pchire(z,w,t) ∧ w≠y →
haspos(y,m,hire(z,w,t)) = haspos(y,m,t)

22 pcsubscr(w,n,t) ∧ w≠y →
haspos(y,m,subscr(w,n,t))=haspos(y,m,t)

23 pcsubscr(w,n,t) ∧ w=y ∧ haspos(y,m-n,t) →
haspos(y,m,subscr(w,n,t)) = true

24 pcsubscr(w,m,t) ∧ w=y ∧ m=n ∧ ∀v ¬haspos(y,v,t) →
→haspos(y,m,subscr(w,n,t)) = true

25 pcsubscr(w,m,t) ∧ w=y ∧ m=n ∧ ∃v(haspos(y,v,t) ∧ v≠m-n) →
→haspos(y,m,subscr(w,n,t)) = false

26 haspos(y,m,apply(z,t)) = haspos(y,m,t)

27 haspos(y,m,initag) = false

28 pchire(z,w,t) ∧ x=z ∧ y=w →
worksfor(x,y,hire(z,w,t)) = true

29 pchire(z,w,t) ∧ (x≠z ∨ y≠w) →
worksfor(x,y,hire(z,w,t)) = worksfor(x,y,t)

30 worksfor(x,y,subscr(w,n,t)) = worksfor(x,y,t)

31 worksfor(x,y,apply(z,t)) = worksfor(x,y,t)

32 worksfor(x,y,initag) = false

Figure 8: Equations

Here a remark about the form of our "equations" is important. Our "equations" have the general form

antecedent → t = t'

and the antecedent may involve quantifiers and negations, but these are "constructive" in that we only have in each case to check subterms of the state term. Thus, we have a minimal model[23,34] to satisfy the specification.

The reasoning above explains why we do not strive for "pure" equations. Of course, our specification could be simplified a little with some (not always systematic) work. For instance, "equation" 2 states that the updates apply(x,.) and apply(z,.) commute under certain conditions. One can check that they always commute and replace 2 by the simpler equation

apply(x,apply(z,t)) = apply(z,apply(x,t))

5. CONCLUSIONS

5.1. Derivation process

One criticism levelled against algebraic specifications is their non-constructiveness[21]. Indeed, we feel it is more natural to start with axioms expressing pre-conditions and effects of operations than addressing, via algebraic equations, the problem of determining the equivalence of sequences of operations. The arrows in Figure 9 depict the path that we chose to follow.

Notice that within the query-oriented specifications we followed a path of diminishing underlying assumptions. The connection between the triangles is via specifications with minimal underlying assumptions in that they concern explicit models. Finally, within the update-oriented triangle, we went from a denotational specification via an operational one to a definitional specification by equations.

The whole process can be simply described as a change of underlying assumptions. The reason why we consider it natural is twofold:

- macroscopically, starting from an informal specification it seems easier to pass to a query-oriented one - where objects are described by their properties - than to an update-oriented one - where objects are actually constructed;
- microscopically, since the underlying assumptions vary from one method to another it is not easy to compare, let alone derive, specifications in different formalisms; that is why the "bridge" is between the "minimal" points.

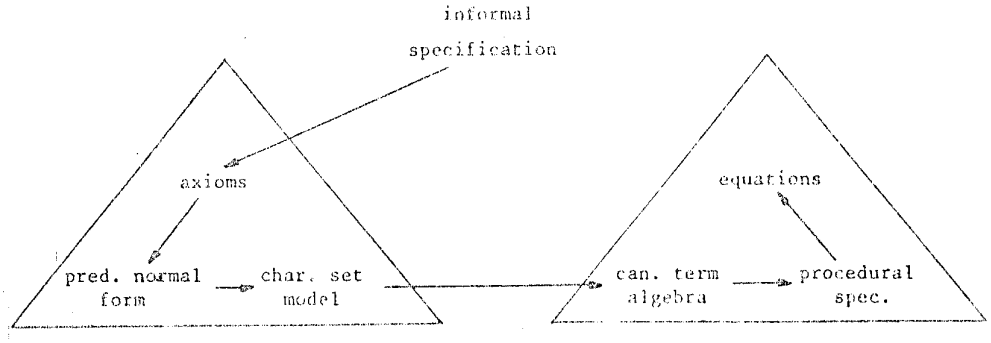


Figure 9: Structure of the derivation process

5.2. Complementarity

The matter of choosing between the two styles of specifications is of course related to their convenience for different usages. Specifications based on logic seem to be preferred by researchers in the area of artificial intelligence, where they are used for designing inferential data bases[9]. Specifications based on algebra have an appeal to researchers in abstract data types, and are used, for instance, for verifying the correctness of conventional implementations[10]. A combined approach is taken elsewhere[30], where theorem-proving is used for synthesizing update transactions (terms) able to take current states into target states (both described by predicates).

As far as suggesting the structuring of data bases, one may note that our query-oriented approach shows the items of information grouped together as they are likely to be retrieved, whereas in the update-oriented approach the grouping mirrors how information is created and changed. In this connection, we might remark that the update- oriented specifications tend to be longer and more intricate than the corresponding query-oriented ones. The reason for this appears to be the fact that the queries are less tightly coupled, in constrast with the strongly coupled term structure underlying the update-oriented approach.

Concerning the finer subdivision, generally a denotational specification will furnish a model, useful for intuitive global understanding. An operational one can be a useful guide in obtaining a lower-level implementation; in particular, operational specifications can be directly translated into some symbol-manipulation language (as we did with SNOBOL[8]) for providing experimental confirmation[31] that the original intentions have been duly captured. A definitional specification is often more adequate for proving properties of the objects.

5.3. Specification formalisms

We regard a specification formalism as consisting of two parts

- syntax: a language for writing expressions;
- semantics: a map assigning to each correct expression an object (a model of the expression).

As a simple example, let us take the case of formal language theory, when we want to specify sets of strings of symbols by means of grammars. We may regard a grammar as a syntactic expression. The semantical map assigns to each such grammar the set of strings of terminal symbols it generates. Notice that underlying the definition of the semantical map are the notions of applying a rule, derivation, etc. Thus, simply giving a grammar, say $N = \{S\}$, $T = \{a,b\}$, $P = \{S \to aS,\ S \to b\}$, is enough to specify the semantical object if we have the semantical map which underlies the formalism.

Another specification formalism used in the same area is that of, say, finite automata. Here, we may view a state graph as a syntactical expression of the formalism. The semantical map explains how the

		orientation: query	orientation: update	
data	implicit	axiomatic (3.1)	algebraic (4.3)	defin.
	explicit	char. set model (3.3)	can. term alg. (4.1)	denot.
		pred. normal form (3.2)	procedural (4.2)	oper.
		abstr. set mach. (3.4)		

functions

Figure 10: Taxonomy of specification formalisms

automaton responds to a given input string and assigns to the state graph the set of strings recognized by it. Thus, the same language specified above by a grammar would be specified in the finite-automaton formalism by a different device.

A third specification formalism used in connection with the above is that of regular expressions. Here, a syntactical expression would be, say, a*b. The semantical map assigns to each expression a set of strings of symbols, in this case |a*b| = {a^n b/ n = 0,1,...}

These examples of specification formalisms for regular sets are intended to illustrate the following points

- within a fixed formalism, a syntactical expression suffices to specify an object because the semantical map is understood;
- the same object can be specified in different formalisms and this can be quite advantageous (complementarity: finding the intersection by means of regular expressions is difficult, but it is easy with finite automata);
- frequently one can derive a specification for an object from another one for the same object in a different formalism, e.g. algorithms for transforming finite automata into regular expressions.

Furthermore, some analogies can be drawn between specification formalisms for regular sets and those we used for data base applications. One such analogy correlates automaton-theoretic transducers, extensional descriptions of regular sets and generative grammars, respectively, with query-oriented processors (as in sections 3.2, 3.3, 3.4), valid sets of terms such as canonical forms of section 4.1, and rewriting rules underlying procedural specifications as in section 4.2.

5.4. Concluding remarks

In the preceding sections we have seen some specifications (syntactical expressions) for the same data base (object) in diverse formalisms, each one with distinct underlying assumptions (semantical components). Moreover, we have proceeded fairly systematically in deriving these formal specifications. Notice that part of the underlying assumptions of a given formalism can be formalized. As mentioned, the process of translating from a formalism to another one amounts to shifting underlying assumptions.

Since the objects specified consist of structured data there are various useful criteria to classify the specification formalisms employed[28]. An important criterion is whether the formalism relies on an explicit representation for the data or whether the nature of the data can be left implicit, the only references to them being via variables as it is typical of definitional methods. Another important criterion is - once we have explicit data - how to specify updates and queries: one can specify directly the mathematical operations and predicates they denote or one can use procedures to implement them. These criteria together with the distinction between observing objects via their properties versus constructing them suggests a taxonomy. Figure 10 (above) shows the classification of our specification formalisms.

ACKNOWLEDGMENTS

The authors are grateful to M.A. Casanova and T.S.E. Maibaum for reading the manuscript and for helpful discussions, and to IBM do Brasil for financial help.

REFERENCES

1 J. Arsac - Nouvelles Leçons de Programmation - Dunod, Paris (1977).

2 E.A. Ashcroft, W.W. Wadge - R for semantics - University of Waterloo, Canada, Research Report CS-79-37 (Dec. 1979).

3 F.L. Bauer, H.Partsch, P. Pepper, H. Wössner - Techniques for program development - Infotech State of the Art Report 34, Berkshire (1977)

4 M.A. Casanova, P.A. Bernstein - A formal system for reasoning about programs accessing a relational database - ACM TOPLAS, vol. 2, nº 3, 384-414 (Jul. 1980).

5 C.L. Chang, R.C.T. Lee - Symbolic Logic and Mechanical Theorem Proving - Academic Press (1973).

6 K.L. Clark - Negation as failure - In: "Logic and Data Bases" (Gallaire, Minker eds.) - Plenum Press (1978).

7 J.E. Donahue - Complementary definitions of programming language semantics - Springer (1976).

8 A.L. Furtado, P.A.S. Veloso - Procedural specifications and implementations for abstract data types - ACM/SIGPLAN Notices, vol. 16, nº 3, p. 53-62 (Mar 1981).

9 H.Gallaire, J. Minker (Eds.) - Logic and Data Bases - Plenum Press (1978).

10 J.A. Goguen, J.W.Thatcher, E.G. Wagner - An initial algebra approach to the specification, correctness and implementation of abstract data types - Current Trends in Programming Methodology, R.T. Yeh (ed.), vol. IV, Prentice Hall, 80-149 (1978).

11 I. Greif, A. Meyer - Specifying programming language semantics - 6th ACM Symp. on Principles of Programming Languages, San Antonio, Tx, 1979.

12 J. Guttag - Abstract data types and the development of data structures - CACM, vol. 20 (Jun. 1977).

13 J.V. Guttag, E. Horowitz, D.R. Musser - Abstract data types and software validation - CACM, vol. 21 nº 12 (Dec. 1978).

14 D. Harel - First-Order Dynamic Logic - Springer (1979).

15 C.A.R. Hoare - Proof of correctness of data representations - Acta Informatica, vol.1(1972).

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

17 G. Huet, D.C. Oppen - Equations and rewrite rules, a survey - Computer Science Department, Stanford University, Report nº STAN-CS-80-785 (1980).

18 P.C. Jackson - Introduction to Artificial Intelligence - Petrocelli (1974).

19 R. Kowalski - Logic for Problem Solving - North-Holland (1980).

20 M.R. Levy - Some remarks on abstract data types- SIGPLAN Notices, vol. 12, nº 7, pp 126-128 (Jul. 1977)

21 B. Liskov, S. Zilles - An introduction to formal specifications of data abstractions - Current Trends in Programming Methodology, vol I, R.T. Yeh (ed.), Prentice Hall, pp. 1-32(1977).

22 Z. Manna - Mathematical theory of computation - McGraw Hill (1974).

23 J. McCarthy - Circumscription: a form of non-monotonic reasoning - Artificial Intelligence, vol. 13, pp 27-39 (Apr. 1980).

24 T.S. Maibaum - Mathematical semantics and a model for data bases - Information Processing 77, B. Gilchrist (ed.) North-Holland, p. 133-138 (1977).

25 M. Melkanoff, M. Zamfir - The axiomatization of database conceptual models by abstract data types - UCLA-ENG (Jan. 1980).

26 N.J. Nilsson - Problem-solving methods in Artificial Intelligence - McGraw-Hill (1971).

27 T.H.C. Pequeno, P.A.S. Veloso - Don't write more axioms than you have to - Proceedings International Computing Symposium, vol. 1, Academia Sinica, Taipei, China, pp 487-498 (Dec. 1978).

28 A.A.Pereda B. - Description methods for data types and data structures (in Portuguese),D.Sc. Thesis, PUC Rio de Janeiro, Brasil (1979).

29 A. Pirotte - Automatic theorem proving on resolution 4 in "Annual Review in Automatic Programming", vol. 7, part 4, Pergamon Press, Oxford.

30 C.S. dos Santos, T.S.E. Maibaum, A.L. Furtado - Conceptual modelling of data base operations - International Journal of Computer and Information Sciences (to appear).

31 M. Shaw - Remark in the discussion during the tutorial on programming language research (led by L.A. Rowe) - Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling - pp. 44 (1980).

32 J. Shoenfield - Mathematical Logic - Addison-Wesley (1967).

33 J.W. Thatcher, E.G. Wagner, J.W. Wright - Data type specifications: parametrizations and the power of specification techniques - 10th SIGACT Conference (1978).

34 M.H.van Emden, R.A. Kowalski - The semantics of predicate logic as a programming language - JACM, vol. 23 (Oct. 1976).