ADVANCES IN DATA BASE THEORY

Volume 2

Edited by

Hervé Gallaire
CGE—Laboratoire de Marcoussis
Marcoussis, France

Jack Minker
University of Maryland
College Park, Maryland

and

Jean Marie Nicolas
Centre d'Etudes et de Recherches de Toulouse
Toulouse, France

PLENUM PRESS • NEW YORK AND LONDON

STEPWISE CONSTRUCTION OF ALGEBRAIC SPECIFICATIONS

P. A. S. Veloso and A. L. Furtado

Pontífica Universidade Católica do Rio de Janeiro

Rio de Janeiro, Brasil

ABSTRACT

A multistep methodology for the formal specification of data base applications, entirely within the algebraic approach to abstract data types, is presented. Towards this end the concept of traces is extended to several levels, which gives a useful tool to obtain formal specifications in a systematic and constructive way. In addition, the concept of traces has familiar simple analogues. The presentation is based on a simplified example of a database, which is successively specified in three formats: procedural notation, systems of term-rewriting rules and conditional equations.

INTRODUCTION

A multistep methodology, entirely within the algebraic approach, for obtaining formal specifications of abstract data types and, in particular, database applications is proposed. The main tool proposed to aid in the systematic construction of these algebraic specifications is the concept of traces extended to several levels.

The algebraic approach to abstract data types has been advocated widely as a useful tool for the formal specification of data structures and, in particular, databases (Ehrig et al. [1978], Paolini [1981]). However, formal specifications in general are not noted for their legibility or ease of construction. In fact, due to the difficulties in obtaining an algebraic specification directly from a model (given formally or, worse, informally) some methodologies have been suggested. Canonical term algebras, used to verify the correctness of specifications (Goguen et al. [1978]), have been used also as an aid in constructing an algebraic specification (Pequeno and Veloso [1978]), and are more helpful if used in conjunction with rewrite rules (Veloso [1982]). Helpful as they are, these methodologies still leave room for improvement. On the other hand, some multistep methods have been proposed (Ehrig and Fey [1981], Veloso et al. [1981]). These methods usually involve intermediate formalisms other than the algebraic one, which is one of their main disadvantages. For instance, one such method (Veloso et al. [1981]) involves specifications by means of logical axioms, abstract models, pre- and post-conditions, in addition to the algebraic specification.

We propose here a multistep methodology where, in contrast, each step can be carried out entirely within a single formalism, in this case, the algebraic formalism itself. The main advantage stems from being both multistep and within a single formalism. The usage of a single formalism avoids the problem of having to translate from one formalism to another. Rather, and that is where the multistep aspect comes in, what has to be done is to refine specifications within a single formalism. This feature of stepwise refinement is, we believe, a major advantage.

The algebraic approach views each object of a data type as (represented by) a variable-free term, so that the abstract data type is (the isomorphism class of) a homomorphic image of the free algebra of terms, the specification pinpointing which homomorphic image. In particular, a database state can be represented by a sequence of update operations capable of generating the state. One may regard such a term as a trace (Bartussek and Parnas [1977]) of how the database has actually evolved on its way to the present state. As such, these terms may contain some extra information, which may be redundant if one is interested in the state per se rather than in a particular way of generating it.

The proposed methodology starts at a level where all ground terms are taken as representatives of states and gradually proceeds via a series of intermediate levels until reaching the desired level, e.g., one with a unique representative for each state. On the intermediate levels the specifications progress towards smaller sets of representatives by considering fewer sequences of updates. As a consequence, the specifications gradually shift their main orientation from queries to updates. Nevertheless, each level specification is consistent and sufficiently complete, in the sense defined in the next section. Furthermore, at each level, three kinds of algebraic formalisms can be employed to express the specifications: (conditional) equations, systems of term-rewriting rules, and procedural notation, the latter leading to executable specifications.

The structure of the chapter is as follows. The next section, PRELIMINARIES, reviews mathematical terminology and notation used in the chapter. In AN ILLUSTRATIVE EXAMPLE, an illustrative example to be used throughout the paper is introduced. This simple example is employed in THE IDEA OF TRACE LEVELS to convey the basic ideas of trace levels. In TRACE LEVELS AND ALGEBRAS, trace levels are formalized in terms of canonical term algebras and some of their properties are presented. These ideas are illustrated in TRACE LEVEL SPECIFICATIONS by presenting semi-formal specifications for various trace levels of our running example, which are translated into the procedural formalism in PROCEDURAL SPECIFICATIONS. In REWRITE RULES we consider and illustrate the methodology as couched in the context of systems of term-rewriting rules. These ideas are further developed in EQUATIONAL SPECIFICATIONS, where we outline the process of deriving an equational specification, again using our running example. Finally, CONCLUSION presents some general conclusions.

This structure is modular in that one can skip PRELIMINARIES and TRACE LEVELS AND ALGEBRAS on a first reading to obtain the main intuitive ideas.

PRELIMINARIES

The usual notation and terminology for abstract data types is employed (Goguen et al. [1978], Guttag and Horning [1978]). For a clear presentation we refer to Pair [1980], whose main concepts of interest here are outlined below.

A *signature* L consists of:

° a nonempty set S of *sorts*
° a set Σ of *operation symbols*
° a *profile declaration* Π assigning to each operation symbol $\sigma \in \Sigma$ its functionality; $\Pi(\sigma) = (s_1 \ldots s_n, s)$, denoted by $\sigma: s_1 \ldots s_n \to s$.

Terms are defined as usual and collected according to their target sorts. In general, we call such families of sets indexed by S an S-*set*.

An L-*algebra* $\mathcal{A}$ consists of an assignment of nonempty *domains* to the sorts of S and of *operations* to operation symbols respecting their profiles. In other words, $\mathcal{A} = (A, \Sigma^{\mathcal{A}})$ where A is an S-set of domains $A_s \neq \emptyset$ for $s \in S$ and for $\sigma \in \Sigma$ with profile $\Pi(\sigma) = (s_1 \ldots s_n, s)$, $\sigma^{\mathcal{A}}: A_{s_1} \times \ldots \times A_{s_n} \to A_s$.

The S-set of all (ground) terms T of L can be given a natural structure of L-algebra, called the term-algebra $\mathcal{T}$ of L, which is initial in the category of L-algebras (with L-homomorphisms) (Goguen et al. [1978]). In fact, given an algebra A the mapping assigning to each $t \in T$ its *denotation* (or value) t^A in A is the unique homomorphism h^A of $\mathcal{T}$ into A. We call A *finitely generated* (Wirsing and Broy [1980]) iff this denotation map h^A is onto A. Then, by the isomorphism theorem of Grätzer [1968], the assignment $A \mapsto \equiv[A]$ (where $t_1 \equiv t_2[A]$ iff $t_1^A = t_2^A$) gives a one-to-one correspondence between the finitely generated algebras and the complete lattice of congruences on $\mathcal{T}$.

The algebra $\mathcal{T}$ of terms has the advantage of having syntactical domains. Other such algebras will also be of interest here.

A *canonical form* F is an S-set of terms (called *canonical terms*) that is closed under the formation of subterms, i.e. whenever $\Pi(\sigma) = (s_1 \ldots s_n, s)$ and $\sigma t_1 \ldots t_n \in F_s$ then $t_1 \in F_{s_1}, \ldots, t_n \in F_{s_n}$. A *canonical term algebra* (*cta*, for short) is an algebra C whose domains constitute a canonical form and their operations consist of syntactical manipulations on canonical terms in the sense that whenever $\Pi(\sigma) = (s_1 \ldots s_n, s)$ and $\sigma t_1 \ldots t_n \in C_s$ then $\sigma^C[t_1, \ldots, t_n] = \sigma t_1 \ldots t_n$.

Canonical forms have their origin in the normal forms of term rewriting systems (Huet and Oppen [1980]). A cta adds to the advantage of having syntactical domains that of ease in carrying out inductive arguments, since being "closed under the formation of subterms" generally is what one needs in the inductive step. Notice that a cta is *not* a subalgebra of the algebra of terms $\mathcal{T}$, but we have the following useful property proved by Goguen, Thatcher and Wagner [1978].

Lemma 1 (Goguen et al. [1978]). If C is a cta and h^C is the denotation homomorphism from $\mathcal{T}$ to C, then h^C is onto and for each $t \in C$ $h^C(t) = t$.

As noted above, a finitely generated algebra is characterized by a congruence on $\mathcal{T}$. In order to describe such congruences we resort to some specification formalism, which consists of a finite number of schemas involving terms with variables and generating the desired congruence.

In many cases it is convenient to single out certain sorts as of special interest (cf. the "type of interest", TOI, of Guttag and Horning [1978]), viewing the others as, say, parameters. In particular for databases, a domain of special interest is that consisting of database states or instances. Operations with values in this domain are updates, the others — with values in other sorts — are queries (Veloso et al. [1981], Dosch et al. [1982]).

This leads to the notion of *hierarchical signature,* which is a signature, as before, with a specified subsignature called *basic subsignature,* consisting of *basic sorts* and *basic operations* whose profiles involve only basic sorts. The terms generated by the basic operations are called *primitive terms.* Notice that the set P_s, of primitive terms of a basic sort s is a subset of T_s. Call an operation symbol whose target sort is basic an *external operation.*

A *hierarchical data type* consists of a hierarchical signature together with an algebra, $\mathcal{B}$, called *basic algebra,* for the basic subsignature. This basic algebra induces a congruence $\equiv_P$, called *primitive congruence,* on P.

A *hierarchical algebra* is an algebra of the signature, whose reduct obtained by restricting the sorts and operations to the subsignature coincides with the basic algebra. The congruences corresponding to hierarchical algebras with a given basic algebra $\mathcal{B}$ form a complete sublattice of the lattice of all congruences on $\mathcal{T}$ (Pair [1980]).

Two important properties of a congruence θ on $\mathcal{T}$ are the following:

- θ is *consistent* iff any class of θ contains at most one class of $\equiv_P$, i.e. whenever $(p,p') \in \theta$, for $p,p' \in P$, also $p \equiv_P p'$.

- θ is *sufficiently complete* iff any class of θ contains at least one class of $\equiv_P$, i.e. for any term t of a basic sort there exists a primitive term $p \in P$, such that $(t,p) \in \theta$.

The above terminology is generally employed in connection with specifications.

We deal with three specification formalisms: equational, (Goguen et al. [1978]), rewriting systems (Huet and Oppen [1980]) and procedural notation (Furtado and Veloso [1981]). A set E of equations (pairs of terms with variables) describes the least congruence $\equiv[E]$ on $\mathcal{T}$ containing all ground instances of the equations (Goguen et al. [1978]). Similarly, a rewriting system R consisting of rewrite rules (ordered pairs of terms with variables) describes the relation $\equiv[R]$ on T (relating two terms that can be converted into a common one), which is a congruence on $\mathcal{T}$ if R is confluent, i.e. has the Church-Rosser property (Huet and Oppen [1980]). A procedural specification is a CLU-like cluster (Liskov et al. [1977]), consisting of a symbol-manipulating procedure for each operation symbol. It can be regarded as a deterministic implementation of a rewriting system obtained by superimposing some order of application for the rewrite rules (Furtado and Veloso [1981]).

A specification Γ in one of the above formalisms generates a *syntactical congruence* ≡ [Γ] on *T* as follows: t ≡ t'[Γ] iff the equality t = t' can be derived from Γ.

Given an algebra *A*, by means of its congruence ≡[*A*], a specification Γ is (see Figure 1)

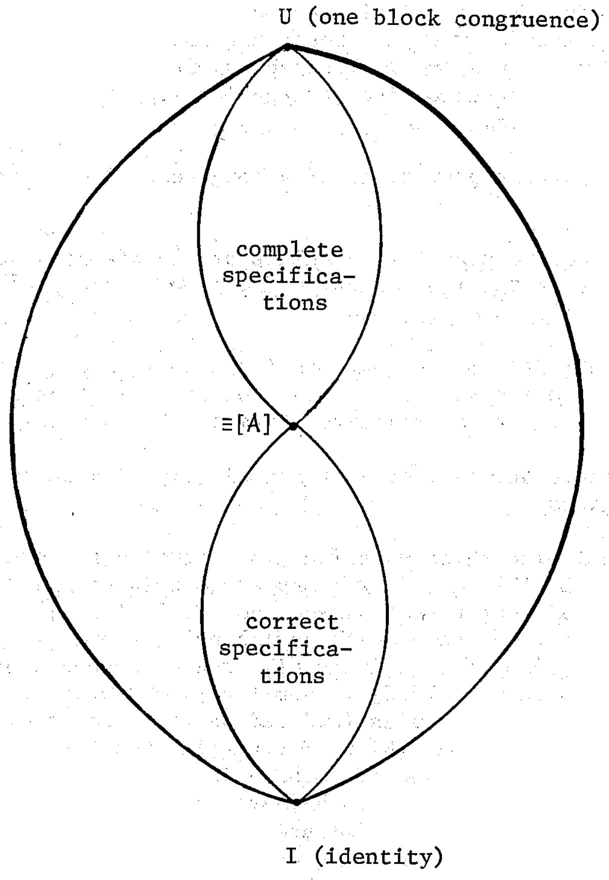


Figure 1. Correct and Complete Specifications with Respect to *A*.

- *correct* for A iff $\equiv[\Gamma] \subseteq \equiv[A]$, i.e., any equality $t = t'$ derivable from Γ is true in A.
- *complete* for A iff $\equiv[A] \subseteq \equiv[\Gamma]$, i.e., any equality $t = t'$ holding in A is derivable from Γ.

The *specification problem* (in a given formalism, as above) for a given algebra A consists of finding a specification Γ (in the given formalism) that is both correct and complete with respect to A.

In the following sections we shall show how the notion of traces, in particular, several levels of traces, can be used in a methodology for the stepwise construction of specifications for abstract data types.

AN ILLUSTRATIVE EXAMPLE

As a simple example to illustrate the discussion we use the database of a company, called Acme, marketing a machine. Acme can either *lease* or *sell* a machine to a customer. In both cases the customer will *use* the machine, but only in the latter case will he *own* it. If a machine has been leased to a customer he may later decide to buy it or else he may choose to *return* it to Acme. On the other hand if a machine has been sold to a customer he cannot return it. For simplicity we assume that there is only one kind of machine and at any time a customer will have at most one machine.

The operation symbols here correspond to the words italicized in the preceding paragraph together with the usual update *phi* which initializes the database to an "empty" state. More specifically. we have three sorts *state, customer* and *Bool*(ean), the TOI is *state* and the other two are basic sorts, in the sense of the previous section.

Operation symbol $\underline{\text{phi}}$ has a profile (λ,*state*); it is a constant for initialization. On the other hand, $\underline{\text{lease}}$, $\underline{\text{sell}}$ and $\underline{\text{return}}$ all have profile (*customer state,state*), and thus are updates. Also, $\underline{\text{uses}}$ and $\underline{\text{owns}}$ have profile (*customer,state,Bool*) and are queries. In addition the basic sorts have some basic operation symbols. *Bool* has two constant symbols $\underline{\text{True}}$ and $\underline{\text{False}}$ and *customer* is assumed to be supplied with operation symbols that allow the generation of variable-free terms as names of customers. Finally, we also have operation symbols = and $\neq$ of profile (*customer customer,Bool*) to compare these customers.

The above exposition is a description of our hierarchical signature L. So far, we have a formal description of the syntax of our example. The meaning of the operations has been given only informally in natural language.

It is assumed that each state can be identified by the results it yields to queries. Also in the initial state all queries yield *False* and the only way to reach a state where a query yields *True* is by applying some sequence of updates.

So, the effect of an update is to cause changes in the answer to queries. Conversely certain answers to queries may be preconditions for an update; if the preconditions fail the application of the update will not change the state. For instance, if in state s customer c owns a machine then return(c,s) = s.

A specification for the updates of our example in terms of preconditions and effects is as follows:

t := phi
  preconditions : none
  effects : $\forall c \neg \underline{\text{uses}}(c,t) \wedge \neg \underline{\text{owns}}(c,t)$

t := lease(c,s)
  preconditions : $\neg \underline{\text{uses}}(c,s)$
  effects : $\underline{\text{uses}}(c,t)$

t := sell(c,s)
  preconditions : $\neg \underline{\text{owns}}(c,s)$
  effects : $\underline{\text{uses}}(c,t) \wedge \underline{\text{owns}}(c,t)$

t := return(c,s)
  preconditions : $\underline{\text{uses}}(c,s) \wedge \neg \underline{\text{owns}}(c,s)$
  effects : $\neg \underline{\text{uses}}(c,t)$

This specification, together with the explicit statement of the underlying assumptions (Veloso et al. [1981]), such as the well-known frame assumption: properties not explicitly mentioned are preserved under the application of an update, constitute a formal description of the semantics of our example. However, it is not an algebraic specification.

THE IDEA OF TRACE LEVELS

In the algebraic approach each database state is denoted by a ground term representing a sequence of update operations capable of generating it. One may regard such a term as a log or trace (Bartussek and Parnas [1977]) of how the database has been manipulated to attain this particular state. As such, these terms may contain extra information that can be discarded if one is interested in the state per se rather than in a particular way of generating it.

In order to clarify these ideas, assume that our example database is initially in the empty state and the following sequence of transactions is performed:

1. a machine is leased to customer D;
2. a machine is leased to customer B;
3. D returns a machine;
4. customer C buys a machine;
5. customer B buys a machine;
6. C returns a machine;
7. a machine is leased to customer A;
8. a machine is leased to customer B.

The list of (the symbols of) the above nine updates (beginning with phi and ending with the last lease would be a log of this sequence of transactions. It is often more convenient, however, to represent this trace in an "applicative" format as

(1) lease(B,lease(A,return(C,sell(B,sell(C,return(D,lease(B,lease(D,phi)))))))) .

The execution of these updates will cause the database to evolve from the initial empty state, through a series of intermediate states, to a state P where customers A, B and C use machines and B and C own machines. This state can be represented by the two sets $U_P = \{A,B,C\}$ and $O_P = \{B,C\}$. Similarly, each intermediate state s can be described by its set U_s of customers using machines and its set O_s of customers owning machines (notice that $O_s \subseteq U_s$ is a static integrity constraint of our example). The next table shows this evolution.

s	U_s	O_s
0	$\emptyset$	$\emptyset$
1	{D}	$\emptyset$
2	{B,D}	$\emptyset$
3	{B}	$\emptyset$
4	{B,C}	{C}
5	{B,C}	{B,C}
6	{B,C}	{B,C}
7	{A,B,C}	{B,C}
8	{A,B,C}	{B,C}

Some points are worth remarking upon in connection with this trace and corresponding evolution.

(a) States 5 and 6 have identical sets O and U, and are the same. Likewise for states 7 and 8. If we delete from trace (1) the updates <u>return</u>(C,•) and <u>lease</u>(B,•) causing these repetitions we obtain the evolution:

update	U	O
<u>phi</u>	∅	∅
<u>lease</u>(D,•)	{D}	∅
<u>lease</u>(B,•)	{B,D}	∅
<u>return</u>(D,•)	{B}	∅
<u>sell</u>(C,•)	{B,C}	{C}
<u>sell</u>(B,•)	{B,C}	{B,C}
<u>lease</u>(A,•)	{A,B,C}	{B,C}

corresponding to the repetition-free trace

(2) <u>lease</u>(A,<u>sell</u>(B,<u>sell</u>(C,<u>return</u>(D,<u>lease</u>(B,<u>lease</u>(D,<u>phi</u>))))))

denoting the same final state P as (1).

(b) In going from state 2 to 3, the set U decreases from {B,D} to {B}. This is due to the fact that the update <u>return</u>(D,•) cancelled the effects of a previous <u>lease</u>(D,•). If we eliminate these two updates from trace (1) we obtain the following evolution:

update	U	O
<u>phi</u>	∅	∅
<u>lease</u>(B,•)	{B}	∅
<u>sell</u>(C,•)	{B,C}	{C}
<u>sell</u>(B,•)	{B,C}	{B,C}
<u>return</u>(C,•)	{B,C}	{B,C}
<u>lease</u>(A,•)	{A,B,C}	{B,C}
<u>lease</u>(B,•)	{A,B,C}	{B,C}

corresponding to the following nondecreasing (albeit not repetition-free) trace denoting the same state P:

(3) lease(B,lease(A,return(C,sell(B,sell(C,lease(B,phi))))))

(c) In state 2 we have B ∈ O because of the update lease(B,·) and subsequently we have in state 5 both B ∈ U and B ∈ O because of update sell(B,·). So the effects of sell(B,·) subsume those of the previous lease(B,·). Now, replace in trace (1) the first lease(B,·) by sell(B,·) to obtain the trace:

(4) lease(B,lease(A,return(C,sell(B,sell(C,return(D,sell(B,lease(D,phi))))))))

which is subsumption-free (but neither repetition-free nor nondecreasing). The evolution corresponding to (4) is the same as (1) except that states 2 and 3 now have O = {B} and state 4 has O= {B,C}.

(d) We can view the aim of executing the original sequence of updates as attaining a state P where U_P = {A,B,C} and O_P = {B,C}. From this viewpoint traces (1) through (4) achieved this aim in a somewhat roundabout way. If we want to increase (U,O) from (∅,∅) to ({A,B,C},{B,C}) via as few intermediate states as possible we may consider the following evolution:

update	U	O
phi	∅	∅
sell(B,·)	{B}	{B}
sell(C,·)	{B,C}	{B,C}
lease(A,·)	{A,B,C}	{B,C}

corresponding to the trace:

(5) lease(A,sell(C,sell(B,phi)))

If we eliminate from trace (5) any one of the updates we fail to achieve the same final state P. So we call (5) a reduced trace.

(e) The effect of lease(B,·) at a state is the insertion of B into the corresponding set U. Likewise lease(C,·) causes the insertion of C into U. So, lease(B,·) and lease(C,·) commute, the net effect of their execution in any order being the insertion of both B and C into U. This gives us the freedom to reorder occurrences of the same operation symbol according to their customer parameters. We can choose an order among customer names (say

A < B < C < D) and reorder adjacent occurrences of the same operation symbol so that A is more external than B, and so forth. For instance, trace (2) is already ordered according to this criterion. Ordered traces corresponding to the previous traces shown above are:

• corresponding to (1):

(6) lease(A,lease(B,return(C,sell(B,sell(C,return(D,lease(B,lease(D, phi))))))))

• corresponding to (3):

(7) lease(A,lease(B,return(C,sell(B,sell(C,lease(B,phi))))))

• corresponding to (4):

(8) lease(A,lease(B,return(C,sell(B,sell(C,return(D,sell(B,lease(D, phi))))))))

• corresponding to (5):

(9) lease(A,sell(B,sell(C,phi)))

(f) Other possible trace levels are combinations of the preceding ones. For instance, an increasing trace (i.e. repetition-free and nondecreasing) is:

(10) lease(A,sell(B,sell(C,lease(B,phi))))

causing the evolution:

O	U
∅	∅
{B}	∅
{B,C}	{C}
{B,C}	{B,C}
{A,B,C}	{B,C}

where each transition causes a strict increment in O or U.

(g) Of special interest are trace levels where each state has a unique representative, for then we have a one-to-one correspondence between states and traces denoting them. A systematic way to achieve this uniqueness consists of proceeding via a series of traces, each one "closer" to uniqueness than the preceding one. For instance:

- starting at the actual trace level, containing traces such as (1),
- pass to the repetition-free level, with traces such as (2),
- then move on to the increasing level, with traces such as (10),
- and proceed to "more definite" levels by adding successively the conditions of being subsumption-free, reduced, etc. and finally ordered.

TRACE LEVELS AND ALGEBRAS

A subterm of a trace corresponds to a past portion of a log. For instance, trace (5) is a reduced trace and its subterm sell(B,phi) is a reduced trace, as well. Moreover, sell(B,phi) is a reduced trace for an intermediate state in the evolution corresponding to trace (5).

So it is natural to define a *trace level* as a canonical form and a *trace algebra* as a canonical term algebra (cta). Thus, a trace level is a set of representatives for the congruence classes of the corresponding trace algebra.

Trace levels can be ordered naturally by inclusion. Also, it is easy to see that the property of being a canonical form is closed under arbitrary unions and intersections. Thus we have the following proposition illustrated in Figure 2.

Proposition 1. The set of all trace levels of a signature L, under inclusion, forms a complete lattice. Its least upper bound or supremum is the actual trace level and its greatest lower bound or infimum is the empty canonical form.

Notice that an empty canonical form is permitted but it corresponds to no cta. The least trace levels corresponding to ctas are those consisting of a single term for each sort. Moreover, a given congruence can have, in general, many sets of representatives. In fact the proof of the following lemma due to Goguen, Thatcher and Wagner [1978] indicates some ways of choosing canonical representatives.

Lemma 2 (Goguen et al. [1978]). Any finitely generated algebra is isomorphic to a canonical term algebra (cta).

In addition this lemma shows that one can always work with trace algebras without losing any finitely generated algebra.

In using trace levels as a tool for methodical specification we are interested only in canonical forms that can be refined so as to represent a given algebra A. In other words, we are interested in ctas C with $\equiv[A] \supseteq \equiv[C]$. Now the corresponding canonical forms are no longer closed under intersection. But we still have the following theorem illustrated in Figure 3.

Theorem 1. The set of trace levels of ctas with congruences included in a given congruence θ on T is a complete upper sub-semilattice of the lattice of all trace levels. Its lub is the actual trace level and its minimal elements are the trace levels of the ctas isomorphic to the quotients T / θ.

Proof: The set of all congruences on T included in θ forms a complete sublattice of the lattice of all congruences on T (Grätzer [1968]). Thus the result follows from Proposition 1 and Lemma 2 in view of the preceding remarks.

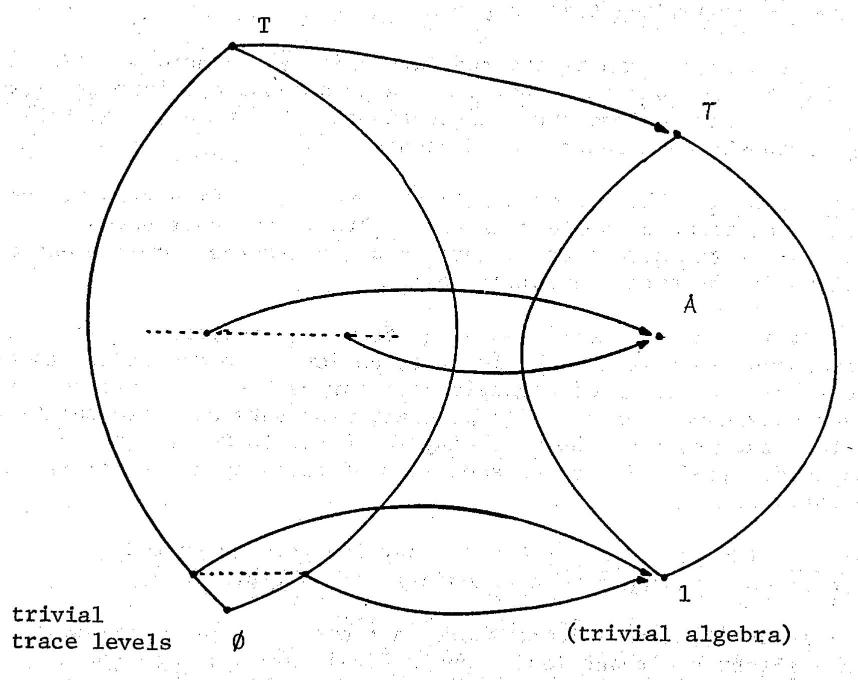


Figure 2. Correspondence between the Lattices of Trace Levels and of Finitely Generated Algebras.

In THE IDEA OF TRACE LEVELS we introduced the concept of trace levels by means of an illustrative example. Some of the trace levels presented therein have wide applicability, therefore deserving general definition and comments.

First the *actual trace level* is the canonical form consisting of all (syntactically correct) (ground) terms without any further restriction. It corresponds to the term algebra T.

The repetition-free level deserves further clarification, in view of a distinction not clearly stressed in THE IDEA OF TRACE LEVELS. A trace level P is *free from adjacent repetitions* iff whenever $\sigma t_1 \ldots t_i \ldots t_n \in P_s$ and $t_i \in P_s$ then their values in the given algebra A are distinct: $t_i^A \neq \sigma^A[t_1^A, \ldots, t_n^A]$. A trace level Q is *free from (arbitrary) repetitions* iff it satisfies the stronger requirement that whenever a term $t \in Q_s$ has one of its proper subterms $\tilde{t}$ also in Q_s then $t^A \neq \tilde{t}^A$.

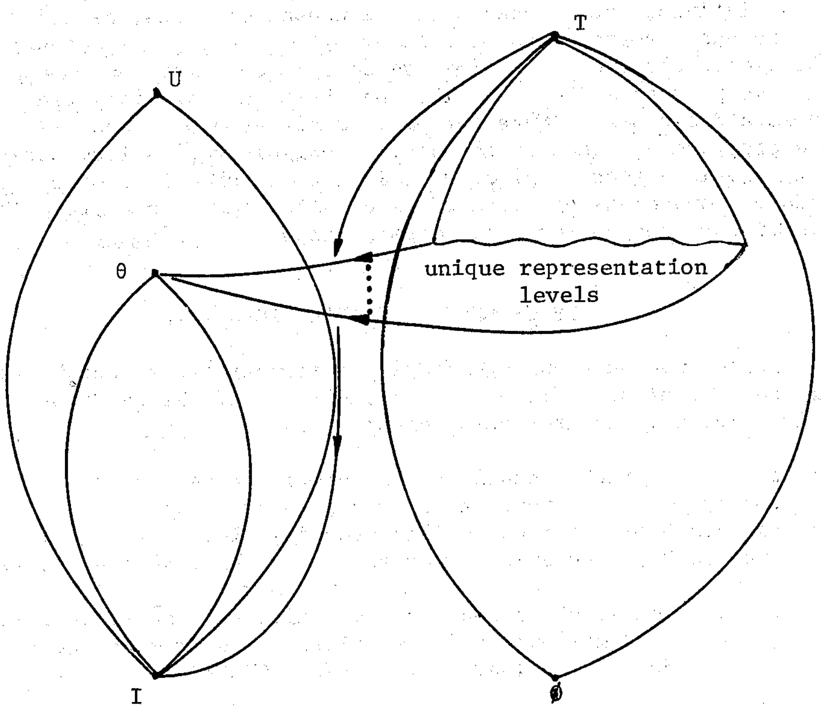


Figure 3. Sublattice of Congruences Included in θ and Corresponding Upper Sub-Semilattice of Trace Levels.

Now a trace level R is said to be *reduced* iff whenever a term t is in R_s and one of its subterms t' has the form $\sigma t_1 \ldots t_i \ldots t_n$ with $t_i \in R_s$ then $t^{A} \neq t_i^{A}$.

Finally, a trace level U is said to be on the *unique-representative level* iff for each $a \in A$ there exists exactly one trace $t \in U$ denoting a, i.e. with $t^{A} = a$.

The following implications are easily seen to hold:

unique-representative $\Rightarrow$ reduced $\Rightarrow$ free from adjacent repetition.

Moreover, they cannot be reversed, in general. Also, notice that while the property of being a canonical form is purely syntactic, this is no longer true for trace levels. A trace level is free from adjacent repetitions, reduced, etc. only with respect to a given algebra A.

The preceding discussion of trace levels and trace algebras concerns finitely generated algebras in general. In the case of hierarchical data types we are given a basic algebra B. We can then relativize our concepts to the nonbasic sorts, as follows. For the basic sorts we choose one primitive term to represent each equivalence class of the primitive congruence. For the nonbasic sorts we proceed as before. Thus we obtain hierarchical trace levels and algebras. More formally, a *hierarchical trace algebra* is a cta, whose reduct to the basic subsignature is isomorphic to the basic algebra. Also, we call a hierarchical trace level actual, nondecreasing, reduced, etc. iff viewed as a trace level it satisfies the corresponding restriction.

TRACE LEVEL SPECIFICATIONS

We now use our running example to illustrate various hierarchical trace level specifications and their usage. We describe, in a semi-formal way, the following ctas:

- the cta T of the actual (hierarchical) trace levels;
- the cta P of the repetition-free (hierarchical) level;
- the cta D of the reduced (hierarchical) level;
- the cta U of the unique-representative (hierarchical) level.

It will become clear that what we call, for short, repetion-free level should more properly be called "free from adjacent repetitions", as defined in TRACE LEVELS AND ALGEBRAS.

Actual (Hierarchical) Trace Level

On this level, the traces of sort *state* are all (syntactically correct) terms, that represent the actual sequences of all updates actually invoked in their chronological order. This is suggestive of the so-called audit trails. Indeed, for auditing or for statistical purposes these actual traces may be useful. As an example consider:

(11) $\underline{\text{lease}}(C,\underline{\text{return}}(A,\underline{\text{return}}(B,\underline{\text{lease}}(C,\underline{\text{sell}}(B,\underline{\text{lease}}(A,\underline{\text{lease}}(B,\underline{\text{phi}})))))))$.

A semi-formal specification for this level consists of:

....

$\underline{\text{lease}}^{T}[c,t] = \underline{\text{lease}}(c,t)$

$\underline{\text{sell}}^{T}[c,t] = \underline{\text{sell}}(c, t)$

....

$$\underline{\text{owns}}^{T}[c, t] = \begin{cases} \underline{\text{True}} & \text{if t contains } \underline{\text{sell}}(c,\ldots) \\ \underline{\text{False}} & \text{otherwise} \end{cases}$$

Of course, this specifies the cta T of all terms of sort *state*. (*Bool*(ean) is assumed to have the constants True and False as traces and the sort *customer* is assumed to have constants as representatives for customers.) Notice that the specification is sufficiently complete (one can determine the results of queries; for instance, with t denoting trace (11), as it contains $\underline{\text{sell}}(B,\ldots)$, we have $\underline{\text{owns}}^{T}[B,t] = \underline{\text{True}}$), and correct with respect to the intended model but is not complete. Indeed, the value of trace (11), as any other trace, is itself on this level, in the sense

$$\underline{\text{lease}}^{T}[C,\underline{\text{return}}^{T}[A,\ldots,\underline{\text{lease}}^{T}[B,\underline{\text{phi}}]\ldots]\ ]$$
$$= \underline{\text{lease}}(C,\underline{\text{return}}(A,\ldots,\underline{\text{lease}}(B,\underline{\text{phi}})\ \ldots\ )\ )$$

Repetition-Free (Hierarchical) Trace Level

Here, a trace consists only of the operation symbols that actually caused state changes. For instance, a repetition-free trace corresponding to (11) is

(12) $\underline{\text{return}}(A,\underline{\text{lease}}(C,\underline{\text{sell}}(B,\underline{\text{lease}}(A,\underline{\text{lease}}(B,\underline{\text{phi}})))))$

Again, this level is suggestive of logs kept, in this case, for recovery purposes.

A semi-formal specification for this level is as follows:

....

$$\underline{\text{lease}}^{P}[c,t] = \begin{cases} \underline{\text{lease}}(c,t) & \text{if } \underline{\text{uses}}^{P}[c,t] = \underline{\text{False}} \\ t & \text{otherwise} \end{cases}$$

$$\underline{\text{return}}^{P}[c,t] = \begin{cases} \underline{\text{return}}(c,t) & \text{if } \underline{\text{uses}}^{P}[c,t] = \underline{\text{True}} \\ & \text{and } \underline{\text{owns}}^{P}[c,t] = \underline{\text{False}} \\ t & \text{otherwise} \end{cases}$$

....

$$\underline{\text{uses}}^{P}[c,t] = \begin{cases} \underline{\text{True}} & \text{if } t \text{ contains } \underline{\text{sell}}(c,\ldots) \\ & \text{or } t = u_1(c_1,u_2(c_2,\ldots,u_n(c_n, \\ & \underline{\text{lease}}(c,t'))\ldots)) \\ & \text{and whenever } u_i = \underline{\text{return}}, \text{ then } c_i \neq c \\ \underline{\text{False}} & \text{otherwise} \end{cases}$$

$$\underline{\text{owns}}^{P}[c,t] = \begin{cases} \underline{\text{True}} & \text{if } t \text{ contains } \underline{\text{sell}}(c,\ldots) \\ \underline{\text{False}} & \text{otherwise.} \end{cases}$$

Again, this is a sufficiently complete specification, correct with respect to the intended model and it is a refinement of the preceding one.

On this level, the value of the trace (12) is itself, in the sense

$$\underline{\text{return}}^{P}[A,\underline{\text{lease}}^{P}[C,\ldots,\underline{\text{lease}}^{P}[B,\underline{\text{phi}}^{P}]\ \ldots]\ ]$$
$$= \underline{\text{return}}(A,\underline{\text{lease}}(C,\ldots,\underline{\text{lease}}(B,\underline{\text{phi}})\ \ldots\ )\ )\ ,$$

as can be seen from the above specification. However, the value of trace (11) can be obtained as follows. First, the specification gives

$$\underline{\text{lease}}^{P}[C,\underline{\text{sell}}^{P}[B,\underline{\text{lease}}^{P}[A,\underline{\text{lease}}^{P}[B,\underline{\text{phi}}^{P}]]]]$$
$$= \underline{\text{lease}}(C,\underline{\text{sell}}(B,\underline{\text{lease}}(A,\underline{\text{lease}}(B,\underline{\text{phi}}))))$$

Call this term t'. Then, as t' contains $\underline{\text{sell}}(B,\ldots)$, we have $\underline{\text{owns}}^{P}[B,t'] = \underline{\text{True}}$, whence $\underline{\text{return}}^{P}[B,t'] = t'$. Now, as t' contains $\underline{\text{lease}}(A,\ldots)$ with no later $\underline{\text{return}}(A,\ldots)$, we have $\underline{\text{uses}}^{P}[A,t'] = \underline{\text{True}}$. Also $\underline{\text{owns}}^{P}[A,t'] = \underline{\text{False}}$. So $\underline{\text{return}}^{P}[A,t'] = \underline{\text{return}}(A,t')$. Now $\underline{\text{uses}}$ $[C,\underline{\text{return}}(A,t')] = \underline{\text{True}}$, which implies $\underline{\text{lease}}^{P}[C,\underline{\text{return}}(A,t')] = \underline{\text{return}}(A,t')$. Hence the value of trace (11) on this level is trace (12). As this equality could not be derived on the actual trace level, we now have a proper refinement, i.e. $\equiv[P] \subsetneq \equiv[T]$.

Ev

Reduced (Hierarchical) Trace Level

On the preceding levels the length of the traces increased with time. Now we keep only those operation symbols whose effects were not later cancelled or subsumed by others. (In this example this is enough to guarantee that the trace is reduced.) A reduced trace corresponding to (12) is

(13) $\underline{\text{lease}}(C,\underline{\text{sell}}(B,\underline{\text{phi}}))$

A semi-formal specification for this level is

....

$$\underline{\text{sell}}^{\mathcal{D}}[c,t] = \begin{cases} \underline{\text{sell}}(c,t'), \text{ where } t' \text{ is the result of removing any occurrence of } \underline{\text{lease}}(C,\ldots) \text{ from } t & \text{if } \underline{\text{owns}}^{\mathcal{D}}[c,t] = \underline{\text{False}} \\ t & \text{otherwise} \end{cases}$$

$$\underline{\text{return}}^{\mathcal{D}}[c,t] = \begin{cases} t & \text{if } \underline{\text{owns}}^{\mathcal{D}}[c,t] = \underline{\text{True}} \text{ or } \underline{\text{uses}}^{\mathcal{D}}[c,t] = \underline{\text{False}} \\ \text{the result of removing } \underline{\text{lease}}(c,\ldots) \text{ from } t & \text{otherwise} \end{cases}$$

....

$$\underline{\text{uses}}^{\mathcal{D}}[c,t] = \begin{cases} \underline{\text{True}} & \text{if } t \text{ contains } \underline{\text{lease}}(c,\ldots) \text{ or } \underline{\text{sell}}(c,\ldots) \\ \underline{\text{False}} & \text{otherwise} \end{cases}$$

$$\underline{\text{owns}}^{\mathcal{D}}[c,t] = \begin{cases} \underline{\text{True}} & \text{if } t \text{ contains } \underline{\text{sell}}(c,\ldots) \\ \underline{\text{False}} & \text{otherwise.} \end{cases}$$

This specification is sufficiently complete, a proper refinement of the preceding one and still consistent with the intended model.

Unique-Representative (Hierarchical) Trace Level

Even on the reduced level a state can be represented by more than one trace. If, however, we decide to order the customers and the corresponding updates as illustrated in THE IDEA OF TRACE LEVELS, we obtain the desired uniqueness. But since we have on the reduced trace level only two updates with customers as parameters, we prefer to reorder the trace so that A appears before B, and so forth, independent of the associated updates. So, a trace corresponding to (13) on this level is:

$\underline{\text{sell}}(B,\underline{\text{lease}}(C,\underline{\text{phi}}))$

A semi-formal specification for this level is:

....

$$\underline{\text{sell}}^{U}[c,t] = \begin{cases} \text{the result of removing any occurrence of } \underline{\text{lease}}(c,\ldots) \text{ and inserting } \underline{\text{sell}}(c,\ldots) \text{ in the appropriate position} & \text{if } \underline{\text{owns}}^{U}[c,t] = \underline{\text{False}} \\ t & \text{otherwise} \end{cases}$$

....

$$\underline{\text{uses}}^{U}[c,t] = \begin{cases} \underline{\text{True}} & \text{if } t \text{ contains } \underline{\text{sell}}(c,\ldots) \text{ or } \underline{\text{lease}}(c,\ldots) \\ \underline{\text{False}} & \text{otherwise.} \end{cases}$$

....

Now, as each state is represented by a unique trace, we have a correct and complete specification for the intended model, i.e. $U \cong A$.

This sequence of four hierarchical trace levels is illustrated in Figure 4, where the values of traces (11), (12), (13) and (14) are shown on each level.

PROCEDURAL SPECIFICATIONS

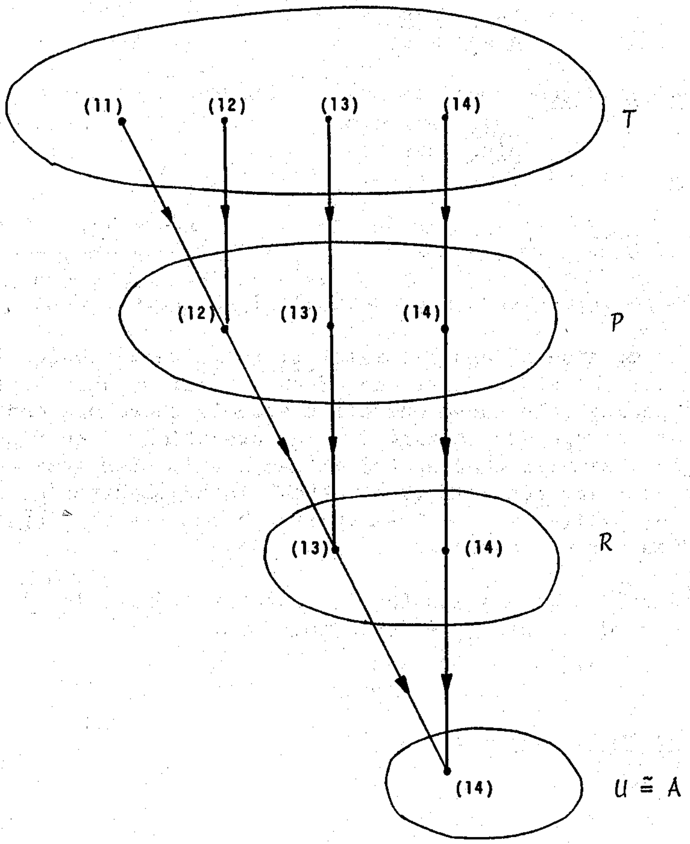
As noted previously, the execution of the operations of a cta involves inspecting and manipulating traces. A procedural specification regards a trace as a sequence of symbols and "implements" the operations by means of procedures for symbolic manipulations.

Here we describe such procedures by means of a procedural notation of Furtado and Veloso [1981], whose main features are as follows. Each procedure (op) has a heading and a body. Within the latter, statements are sequentially executed and the value returned is that of the first expression on the right of a '⇒' whose left-hand side has the value True. The match statement is a case-like construct for pattern-matching, its value is that of the right-hand side of the expression whose left-hand side mathces the trace.

Actual (Hierarchical) Trace Level

For the actual trace level, the symbolic execution of an update is trivial; it suffices to add to the trace the symbol of the new operation. On the other hand, in order to know whether B uses a machine at the state denoted by (11) it is neither enough to find <u>return</u>(B,...) (and answer <u>False</u>) nor to answer <u>True</u> simply because we found <u>lease</u>(B,...). It is necessary to check whether the corresponding updates caused state changes and were not later cancelled.

A procedural specification for this level consists of procedures such as the following:



<u>Figure 4</u>. Sort *state* of Four Hierarchical Trace Algebras

```
....................
op sell(c:customer, t:state) : state
   ⇒ sell(c,t)
endop
....................
op uses(c:customer, t:state) : Bool
    var x:customer, s:state
    match t
      phi ⇒ False
      lease(x,s) ⇒ if c = x
                      then True
                      else uses(c,s)

      sell(x,s)  ⇒ if c = x
                      then True
                      else uses(c,s)

      return(x,s)⇒ if c = x
                      then owns(c,s)
                      else uses(c,s)
    endmatch
endop
....................
```

Repetition-Free (Hierarchical) Trace Level

Now the execution of updates is still relatively simple, for any addition will be at the left end of the trace; we only have to take care in adding only those operation symbols whose preconditions for state change are satisfied. The execution of queries becomes slightly simpler than on the actual level. For instance, if the trace contains return(A,...) without later occurrences of lease(A,...) or sell(A,...), we can guarantee that at this state customer A does not use a machine.

A procedural specification for the repetition-free level consists of procedures with the following form:

```
....................
op sell(c:customer, t:state) : state
    owns(c,t) ⇒ t;
    ⇒ sell(c,t)
endop
....................
```

```
op uses(c:customer, t:state) : Bool
    var x:customer, s:state
    match t
      phi ⇒ False
      lease(x,s)    ⇒  if c = x
                          then True
                          else uses(c,s)

      sell(x,s)     ⇒  if c = x
                          then True
                          else uses(c,s)

      return(x,s)   ⇒ if c = x
                          then False
                          else uses(c,s)
    endmatch
endop
....................
```

Notice that the body of the above procedure sell has a precondition.

Reduced (Hierarchical) Trace Level

Now the execution of updates becomes somewhat more complex, as some manipulation within the trace may be necessary. On the other hand, the execution of queries becomes simpler as traces are more "compact" and contain no "negative" updates.

Procedures for the reduced trace level have the following form:

```
....................

op sell(c:customer, t:state) : state
    var x:customer, s:state
    owns(c,t) ⇒ t;
    ¬ uses(c,t) ⇒ sell(c,t);
    match t
      lease(x,s)   ⇒  if c = x
                         then sell(c,s)
                         else lease(x,sell(c,s))
      sell(x,s)    ⇒     sell(x,sell(c,s))
    endmatch
endop
....................
```

```
op uses(c:customer, t:state) : Bool
    var x:customer, s:state
    match t
       phi ⇒ False
       lease(x,s)    ⇒ if c = x
                          then True
                          else uses(c,s)
       sell(x,s)     ⇒ if c = x
                          then True
                          else uses(c,s)
    endmatch
endop
....................
```

Unique-Representative (Hierarchical) Trace Level

On this level the execution of updates involves more internal manipulations than on the preceding one, since the addition of an operation symbol is to be performed at an appropriate position. The execution of queries, however, is just as simple; in fact they can become more efficient if one takes advantage of the ordering.

A specification for this trace level in our procedural notation, employing the order '<' among customer names, looks like:

```
....................

op sell(c:customer, t:state) : state
    var x:customer, s:state
    owns(c,t) ⇒ t;
    match t
      phi ⇒ sell(c,t)
      lease(x,s)    ⇒ if x = c
                         then sell(x,s)
                         else if c    x
                                then sell(x,t)
                                else lease(x,sell(c,s))
      sell(x,s)     ⇒ if c < x
                         then sell(c,t)
                         else sell(x,sell(c,s))
    endmatch
endop
....................
```

```
...................
op uses(c:customer, t:state) : Bool
    var x:customer, s:state
    match t
      phi ⇒ False
      lease(x,s)   ⇒ if  c = x
                         then True
                         else if c < x
                               then False
                               else uses(c,s)
      sell(x,s)    ⇒ if  c = x
                         then True
                         else if c < x
                               then False
                               else uses(c,s)
    endmatch
endop.
```

A procedural specification actually converts a sequence of operations into the corresponding trace. For instance, consider

(15) lease(C,return(A,return(B,lease(C,sell(B,lease(A,lease(B,phi)
))))))

If we execute expression (15) with the procedures of the repetition-free level we obtain as result, trace (12) whereas the same expression (15) executed on the reduced trace level yields trace (13). On the other hand the result of executing

    return(A,lease(C,sell(B,lease(A,lease(B,phi)))))

on the unique-representative level is

    sell(B,lease(C,phi))

Some remarks concerning these specifications are in order. First, notice that, as we progress from the actual trace level to that of unique representatives, the syntactical complexity of the procedural specifications shifts from queries towards updates. Also the cluster-like module of a given level will generate exactly the traces of this level.

It is worthwhile mentioning that a procedural specification is not only formal but also executable. This allows the designer to experiment with the specification to determine whether the original intentions were actually captured before being committed to the costly and arduous task of machine implementation. Besides, it is relatively straightforward to translate a procedural specification into actual programs written in some symbol-manipulation language such as SNOBOL, as in Furtado and Veloso [1981].

REWRITE RULES

As mentioned above, a procedural specification can be regarded as a device for transforming sequences of operation symbols into the corresponding traces. Such transformations can also be described in another formalism, namely that of term rewriting rules (Huet and Oppen [1980]).

We illustrate the derivation of rewrite rules by means of a simple example. Consider again trace (11). It denotes a state where customer B uses a machine, as can be seen from any of the preceding specifications: the informal, the semiformal or the procedural. We consider the problem of transforming the term $\underline{\text{uses}}(B, (11))$ into $\underline{\text{True}}$.

We may start by trying to convert $\underline{\text{uses}}(B, (11))$ into simpler terms by moving $\underline{\text{uses}}$ inwards until the transformation is immediate. We are led to the following rules:

(16) $\underline{\text{uses}}(x,\underline{\text{lease}}(y,s)) \to \underline{\text{uses}}(x,s)$, whenever $x \neq y$

(17) $\underline{\text{uses}}(x,\underline{\text{sell}}(x,s)) \to \underline{\text{True}}$.

These rules correspond to two possible paths in the execution of the procedure, uses, on the actual trace level, the first one corresponding to the path where t matches $\underline{\text{lease}}(y,s)$ and y is different from the customer parameter.

Notice that the rule (16) has a precondition, which may be incorporated into the rule if we assume the Boolean sort equipped with an if-then-else, together with its natural specification. Then, (16) would be merged with its companion rule into:

$$\underline{\text{uses}}(x,\underline{\text{lease}}(y,s)) \to \underline{\text{if}}\ x = y\ \underline{\text{then}}\ \underline{\text{True}}\ \underline{\text{else}}\ \underline{\text{uses}}(x,s) \cdot$$

We shall not pursue the example here, as it is similar to the equational one, treated in the next section. Some general remarks, however, are in order.

Consider a system R of rewrite rules and sets of terms $V,W \subseteq T$. We call V R-$\underline{\text{controllable}}$ to W, noted $V \overset{R}{\dashrightarrow} W$, iff for every $v \in V$ there exists a $w \in W$ such that $v \xrightarrow{R} w$. Here we employ the usual notation $v \xrightarrow{R} w$ to denote that v can be rewritten as w according to the rules of R. Given an algebra A, call R *sound* on V with respect to A iff whenever $v \xrightarrow{R} t$ with $v \in V$ then $v^A = t^A$.

In order to specify a cta C we need a system R that is (α) complete, in the sense $T \xrightarrow{R} C$, and

(β) correct, in the sense that R is sound on T with respect to $\mathcal{C}$.

Such a system R must have the Church-Rosser property, but not necessarily that of finite termination, the role of the latter being played here by controllability.

Notice that each of the procedural specifications presented for our running example is actually a deterministic implementation of a rewriting system with the above properties. In particular, on each level, T is controllable to the corresponding canonical form. However, from the viewpoint of stepwise specification, there is a simpler alternative: we may relativize requirements (α) and (β) to the preceding level. For instance, for the rewriting system of the reduced level, it suffices that

- the repetition-free canonical form be controllable to the reduced canonical form;
- the rules are sound on the repetition-free canonical form with respect to $\mathcal{D}$.

In general, our stepwise methodology for specifying a given algebra $\mathcal{A}$ will consist of obtaining a sequence of trace algebras $\mathcal{T} = \mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_n \cong \mathcal{A}$ with corresponding trace levels $C_0, C_1, \dots, C_n$ and a sequence of rewriting systems $R_1, \dots, R_n$ such that for each $k = 1, \dots, n$

- C_{k-1} is R_k-controllable to C_k;
- R_k is sound on C_{k-1} with respect to $\mathcal{C}_k$.

Theorem 2. Under the above conditions the rewriting system $R = R_1 \cup \dots \cup R_n$ is a correct and complete specification for $\mathcal{C}_n$, and $\mathcal{T}/\equiv[R] \cong \mathcal{A}$, i.e. the quotient of $\mathcal{T}$ by the congruence $\equiv[R]$ is isomorphic to the given algebra $\mathcal{A}$.

Furthermore, this approach leads naturally to a better documentation for R. Namely, $R_1 \{C_1\}\ R_2 \dots R_{n-1} \{C_{n-1}\}\ R_n$, where the comment $\{C_k\}$ gives a description of the intermediate trace level. It has the advantage of suggesting to a prospective user of the specification a good and safe way to use it; namely, first use the rules of R_1 to rewrite into C_1, then apply the rules of R_2, etc.

The preceding general remarks refer to an arbitrary finitely generated algebra. In this case we have $\mathcal{C}_0 = \mathcal{T}$, where C_0 is the actual trace. For the case of hierarchical data types, we have given a basic algebra $\mathcal{B}$. We assume that the corresponding primitive congruence $\equiv_P$ on the basic sorts is given by means of a system R_0

of rewrite rules that is a correct and complete specification for the basic algebra $\mathcal{B}$.

The relativization of our general methodology for the specification of the given hierarhical algebra $\mathcal{A}$ then is as follows

(1) Start with a rewrite system R_0, such that $\equiv[R_0]$ is consistent and sufficiently complete (with respect to the given primitive congruence).

(2) Obtain a sequence of hierarchical trace algebras $\mathcal{C}_0,\mathcal{C}_1,\ldots,\mathcal{C}_n$ with corresponding hierarchical trace levels $C_0,C_1,\ldots,C_n$, so that C_0 is the actual hierarchical trace level and C_n is a unique representative hierarchical trace level.

(3) Obtain a sequence of rewrite systems $R_1,\ldots,R_n$ for the terms of nonbasic sorts, such that for $k = 1,\ldots,n$

- R_k is sound on C_{k-1} with respect to $\mathcal{C}_k$.
- C_{k-1} is R_k-controllable to C_k.

An important aspect of this stepwise methodology is its modularity with respect to sorts as well. In dealing with hierarchical data types we may assume the above R_0 as given, which amounts to assuming the basic algebra $\mathcal{B}$ already specified. But, we can also back up and use the same general methodology to construct a specification for the basic algebra $\mathcal{B}$ itself in a stepwise manner. This was illustrated in the beginning of this section when we obtained rewrite rules to convert uses(B,(11)) into True.

EQUATIONAL SPECIFICATIONS

In the previous section we tried to obtain rules to transform a term into the corresponding canonical form. These rules can be translated easily into conditional equations. Alternatively, we may ask ourselves what axioms would enable us to derive the equalities between terms and corresponding canonical forms. We illustrate this process within our stepwise approach, level by level.

Actual (Hierarchical) Trace Level

Here, the equalities between terms of sort state consist only of syntactic identities, no special equations being needed for them. All we need is a set of conditional equations allowing us to derive the correct answers for all queries.

The following twelve conditional equations obtained by the reasoning outlined in the preceding section, are:

(18) $\underline{\text{uses}}(x,\underline{\text{phi}}) = \underline{\text{False}}$
(19) $\underline{\text{owns}}(x,\underline{\text{phi}}) = \underline{\text{False}}$
(20) $\underline{\text{uses}}(x,\underline{\text{lease}}(x,s)) = \underline{\text{True}}$
(21) $x \neq y \rightarrow \underline{\text{uses}}(x,\underline{\text{lease}}(y,s)) = \underline{\text{uses}}(x,s)$
(22) $\underline{\text{owns}}(x,\underline{\text{lease}}(y,s)) = \underline{\text{owns}}(x,s)$
(23) $\underline{\text{uses}}(x,\underline{\text{sell}}(x,s)) = \underline{\text{True}}$
(24) $x \neq y \rightarrow \underline{\text{uses}}(x,\underline{\text{sell}}(y,s)) = \underline{\text{uses}}(x,s)$
(25) $\underline{\text{owns}}(x,\underline{\text{sell}}(x,s)) = \underline{\text{True}}$
(26) $x \neq y \rightarrow \underline{\text{owns}}(x,\underline{\text{sell}}(y,s)) = \underline{\text{owns}}(x,s)$
(27) $\underline{\text{uses}}(x,\underline{\text{return}}(x,s)) = \underline{\text{owns}}(x,s)$
(28) $x \neq y \rightarrow \underline{\text{uses}}(x,\underline{\text{return}}(y,s)) = \underline{\text{uses}}(x,s)$
(29) $x \neq y \rightarrow \underline{\text{owns}}(x,\underline{\text{return}}(y,s)) = \underline{\text{owns}}(x,s)$

Notice that these equations are arranged according to the leading operation symbol in the trace: $\underline{\text{phi}}$, $\underline{\text{lease}}$, $\underline{\text{sell}}$, $\underline{\text{return}}$ and then according to the query: $\underline{\text{uses}}$, $\underline{\text{owns}}$.

Repetition-Free (Hierarchical) Trace Level

In TRACE LEVEL SPECIFICATIONS we saw that (12) is a repetition-free trace corresponding to the actual trace (11). One way to derive the equality (11) = (12) is by means of axioms enabling the elimination of the symbols of updates causing no net state change. This can be done by axioms (30) to (33) below, which should be added to the preceding ones to give an equational specification for the repetition-free trace level.

(30) $\underline{\text{uses}}(x,s) = \underline{\text{True}} \rightarrow \underline{\text{lease}}(x,s) = s$
(31) $\underline{\text{owns}}(x,s) = \underline{\text{True}} \rightarrow \underline{\text{sell}}(x,s) = x$
(32) $\underline{\text{uses}}(x,s) = \underline{\text{False}} \rightarrow \underline{\text{return}}(x,s) = s$
(33) $\underline{\text{owns}}(x,s) = \underline{\text{True}} \rightarrow \underline{\text{return}}(x,s) = s$

Notice that the last axiom above concerns violation of requirements, whereas the other three refer to redundant updates.

Reduced (Hierarchical) Trace Level

Referring again to TRACE LEVEL SPECIFICATIONS, we see that (13) is a reduced trace corresponding to trace (12). In order to derive the equality (12) = (13) we need an equation like (34) below, which states that a $\underline{\text{return}}$ cancels an immediately preceding $\underline{\text{lease}}$. In order to treat nonadjacent operation symbols we further introduce commutative axioms — both conditional ones like (40) and unconditional ones like (36), (37), (38) and (39).

(34) $\underline{\text{return}}(x,\underline{\text{lease}}(x,s)) = \underline{\text{return}}(x,s)$
(35) $\underline{\text{sell}}(x,\underline{\text{lease}}(x,s)) = \underline{\text{sell}}(x,s)$
(36) $\underline{\text{lease}}(x,\underline{\text{lease}}(y,s)) = \underline{\text{lease}}(y,\underline{\text{lease}}(x,s))$
(37) $\underline{\text{sell}}(x,\underline{\text{sell}}(y,s)) = \underline{\text{sell}}(y,\underline{\text{sell}}(x,s))$

(38) $\underline{\text{lease}}(x,\underline{\text{sell}}(y,s)) = \underline{\text{sell}}(y,\underline{\text{lease}}(x,s))$

(39) $\underline{\text{return}}(x,\underline{\text{sell}}(y,s)) = \underline{\text{sell}}(y,\underline{\text{return}}(x,s))$

(40) $x \neq y \rightarrow \underline{\text{return}}(x,\underline{\text{lease}}(y,s)) = \underline{\text{lease}}(y,\underline{\text{return}}(x,s))$

Axioms (18) to (40) constitute an equational specification for this level.

Unique-Representative (Hierarchical) Trace Level

In our example, we can already derive from the previous level specification equalities like (13) = (14). In general, we may need some extra axioms, typically of commutativity, enabling the reordering of some terms.

Thus, conditional equations (18) to (40) constitute a correct and complete equational specification of our running example. We just remark that axioms (27), (28) and (29) are no longer necessary and may be discarded. Actually, these three axioms were no longer needed for the reduced trace level for the same reason: return no longer occurs in the traces.

CONCLUSION

The proposed methodology provides a multistep strategy for the difficult task of obtaining an algebraic specification, noting that every step is within the algebraic formalism itself.

The methodology starts at a level where all ground terms are taken as representatives for states and gradually proceeds via a series of intermediate levels until reaching the desired level (say, that with a unique representative for each state). On the intermediate levels the specifications progress towards smaller sets of representatives by considering fewer sequences of updates as representatives. Typical (but not exhaustive) examples of criteria for this purpose are:

- not adding an update producing no net effect in a state,
- making a "negative" update, cancel the corresponding "positive" update;
- making an update whose effects subsume those of another, replace the latter;
- reordering some updates that commute.

Some general properties of these level specifications are worth mentioning:

- each level corresponds to a canonical term algebra;
- the set of trace levels is characterized in terms of lattices;

- the correctness criterion for each level is given by the observability relation '~': we have $t \sim t'$ iff, for all queries q, $q(t,t_1,\ldots,t_n) = q(t',t_1,\ldots,t_n)$; and $t \sim t'$ must imply that t and t' denote the same state;
- each level specification is sufficiently complete;
- at each level any one of the three following kinds of algebraic formalisms can be employed to express the specifications: (conditional) equations, rewriting rules and procedural notation.

From an application point of view, the intuitive meaning of traces as carrying a "history" of the database deserves attention; the extra information available at the different trace levels may be of interest during the early experimental phase, made possible by the usage of executable specifications.

Finally we should stress that this methodology, theoretically proven correct, has been found quite useful in practice in the specification of a number of examples of database applications.

REFERENCES

1. Bartussek, W. and Parnas, D. [1977] "Using Traces to Write Abstract Specifications for Software Modules", Technical Report, 77-012, University of North Carolina (1977).

2. Ehrig, H., Kreowski, H. J., and Weber, H. [1978] "Algebraic Specification Schemes for Data Base Systems", *Proceedings Fourth International Conference on Very Large Data Bases,* Berlin, Germany (1978) 427-440.

3. Ehrig, H. and Fey, W. [1981] "Methodology for the Specification of Software Systems: from Formal Requirements to Algebraic Design Specifications", *Proceedings GI-11 Jahrestagung* (W. Brauer, Ed.), Springer (1981) 255-269.

4. Dosch, W., Mascari, G., and Wirsing, M. [1982] "On the Algebraic Specification of Databases", *Proceedings Eighth International Conference on Very Large Data Bases,* Mexico City, Mexico (1982).

5. Furtado, A. L., and Veloso, P.A.S. [1981] "Procedural Specifications and Implementations for Abstract Data Types", *ACM/SIGPLAN Notices 16*(3) (1981) 53-62.

6. Furtado, A. L., Veloso, P.A.S., and Castilho, J.M.V. [1981] "Verification and Testing of S-ER Representations", In: *Entity Relationship Approach to Information Modeling and Analysis* (P.P. Chen, Ed.) E-R Institute (1981) 125-149.

7. Guttag, J. V. and Horning, J. H. [1978] "The Algebraic Specification of Abstract Data Types", *Acta Informatica 10*(1) (1978) 27-52.

8. Grätzer, G. [1968] *Universal Algebra*, D. van Nostrand (1968).

9. Goguen, J. A., Thatcher, J. W. and Wagner, E. G. [1978] "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) 80-149.

10. Huet, G. and Oppen, D. C. [1980] "Equations and Rewrite Rules: a Survey", Technical Report STAN-CS-80-785, Stanford University (1980).

11. Liskov, B. H. et al. [1977] "Abstraction Mechanisms in CLU", *Communications of the ACM 20*(8) (1977) 564-576.

12. Paolini, P. [1981] "Abstract Data Types and Data Bases", *ACM/SIGMOD Record 11*(2) (1981) 171-173.

13. Pair, C. [1980] "Sur les Modèles des Types Abstraits Algebriques", Séminaire d'Informatique Théoretique, Université de Paris VI et VII (1980).

14. Pequeno, T.H.C. and Veloso, P.A.S. [1978] "Don't Write More Axioms than You Have to", *Proceedings International Computing Symposium 1*, Academia Sinica (1978) 487-498.

15. Veloso, P.A.S., Castilho, J.M.V., and Furtado, A. L. [1981] "Systematic Derivation of Complementary Specifications", *Proceedings Seventh International Conference on Very Large Data Bases*, Cannes, France (1981) 409-421.

16. Veloso, P.A.S. [1982] "Methodical Specification of Abstract Data Types via Rewriting Systems", *International Journal of Computer and Information Sciences 11*(5) (1982) 295-323.

17. Wirsing, M. and Broy, M. [1980] "Abstract Data Types as Lattices of Finitely Generated Models", Institut für Informatik, Tech. Univ. München (1980).