An Informal Approach to Formal (Algebraic) Specifications

A. L. FURTADO*
Departamento de Informatica, Pontificia Universidade Catolica do Rio de Janeiro, Rua Marques de Sao Vicente 225, CEP-22453, Rio de Janeiro, Brazil

T. S. E. MAIBAUM†
Department of Computing, Imperial College, 180 Queen's Gate, London SW 7 2BZ

Formal techniques exist for the crucial specification phase in the design of systems, including database applications. We briefly indicate the potential benefits of the so-called abstract data type discipline and show how it might be made more palatable to the non-mathematician. This is done through the mechanism of traces. This tool is used both as a mechanism for modelling (in an executable manner) the application and as a basis for a methodology which can be used in the development of a formal algebraic specification.

1. INTRODUCTION

When an enterprise decides to adopt the database approach for some application area, its first concern should not be to determine how the data will be structured, but to specify what kinds of data will be kept and the usage characteristics that can be anticipated.

Formal techniques exist for this crucial specification phase.[1] They have been discussed and evaluated recently.[2] We shall briefly indicate some of the potential benefits of the so-called abstract data-type discipline.

Using this discipline, we may concentrate initially on the abstract classes of data we need and the usage of such data, by introducing an adequate set of operations and predicates. The descriptions of the operations and predicates are such that it is possible to convert the descriptions into executable procedures.[3] So we can experiment with the specification, letting the users find out if it really behaves as they imagined, before the enterprise commits itself to a costly and time-consuming process of structuring the data and designing the production implementation to conform to the given specification.

In addition to being executable, this formal specification lends itself well to the verification of the fact that the ensuing implementation is correct (i.e. will exhibit the behaviour specified), remains as good documentation for the benefit of those users who do not want to know about the details of implementation, and can be re-used in the future for experimenting with possible changes.

However, the general style of the abstract data-type literature is not directed to the practitioner, thus diminishing the opportunity of application of these techniques in the business data-processing area,[4] where even the more immediately practice-oriented new techniques are not meeting ample acceptance.[5] One strategy – to be pursued in this paper – to bridge the gap is to show that several notions that have been presented in logical or mathematical terminology are translatable into notions quite familiar to the data-processing specialist.

Paramount among these is the notion of trace (of operations executed). Traces constitute a widely used tool for testing programs, being generated as a by-product of their execution. Here we shall show that traces can play yet another role, serving as a 'universal' data structure upon which to base the previously mentioned executable specifications. (We would point out that we intend to use traces as a tool in building conceptual schema and do not intend in any way to suggest that it should also be an implementation tool. The internal schema and its relationship to the conceptual schema is not dealt with at all here. See ref. 6.) Using traces as a data structure means, for example, that in order to answer the query: 'Does Peter work for Acme?', we inspect the trace to see if an operation *hire*(Peter, Acme) has been invoked in the presence of the required pre-conditions (in turn created by other operations) and has not been superseded by some subsequent operation.

Our contention is not that all the logical and mathematical treatment is a mere embellishment to a few trivial practical concepts. To say that the Herbrand terms from the algebraic theory of abstract data types are 'just' sophisticated forms of traces is the same as saying that relations (from the relational data model) are 'just' flat files or that CODASYL sets (from the network data model) are 'just' a pointer structure. Our contention is that the deep theory underlying abstract data types is not an impediment to the practical application of its results and that the simple analogies proposed in this paper can pave the way towards an intuitive understanding. In a more formal setting the word 'traces', in the sense of algebraic terms, has already been used.[7]

Section 2 introduces a simplified example to be used throughout the discussion. Traces and trace levels are treated in section 3. Section 4 contains examples illustrating in enough detail how to write, at each trace level, the symbolic procedures corresponding to the updates and queries informally described in section 3. A few connections with the theoretical fundamentals are drawn in section 5. Section 6 presents the case for and against the use of the formal techniques described.

2. A SIMPLE DATABASE APPLICATION

As an example of a simplified database application we shall use the database of an employment agency. Here, persons apply for positions, companies subscribe by offering positions, and companies hire candidates or fire

* Partially supported by FINEP.
† To whom correspondence should be addressed.

employees. We impose the following integrity constraints: a person may apply once only, thus becoming a candidate, losing his status when hired by a company, but regaining it if fired: a company may subscribe several times, the number of offerings – which must be positive – 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 has a given number of vacant positions. We assume the database to be initialized to an empty state.

3. TRACE LEVELS

Tracing has been defined as providing 'a record of each processed instruction, by the recording of all instructions, operands and results for analysis of computer runs'.[8] Here, an instruction is the invocation of an update operation, and the recording will be produced on what we shall simply call a **sequence**.

For us, the idea of **state** or **database instance** is crucial in both the formalization of the concepts to be outlined and the understanding of the intuitive and less formal notions we use to develop the concept of trace. A database can be seen as a series of states where the transition from one state s to another s' is accomplished by means of updates applied to the state s. Thus updates have meaning only in the context of the state to which the update is applied. Similarly, queries are applied to particular states and have a meaning only in this context. Hence a given query applied to two different states may yield different results. What we shall in fact attempt to do is to represent various states of the database by traces of update operations.

Note that, in a rigorous sense, the state to which an update or query is applied is an argument of the applied operation. However, it is often left implicit, as is the environment in conventional programs when an assignment is performed.

The execution of an update operation, recalling that traces are assumed to be recorded somehow on 'sequences', consists of adding to the sequence the name of the operation with the actual parameters used. The execution of queries will consist of scanning (analysing) the sequence and determining information based on this scan.

If a trace in fact includes all update operations invoked, the resulting sequence will contain more information than just what is needed for answering a query related to the current state attained. An analysis of the sequence would disclose what operations were attempted but failed to alter the state (by not meeting some pre-condition), what intermediate states were traversed and by means of what operations ('historical data'), and the chronological order of the operations. This extra information may or may not be useful. If it is not, we gain the freedom of not keeping the record of such updates or, in general, of restructuring the sequence in convenient ways. These considerations lead us to identify different **levels** of trace.

The reader will notice that, for higher levels, the difficulty of the operations increases for updates and decreases for queries. Another interesting property is that the queries as 'programmed' for a level will work over traces of all the subsequent levels (although less efficiently than those pertaining to those levels). Table 1 shows the series of updates to be referred to in the discussion.

Table 1. Example series of updates

1	initag;
2	apply(Peter);
3	subscribe(Acme, 1);
4	apply(John);
5	hire(John, Acme);
6	hire(Peter, Acme);
7	fire(John, Acme);
8	hire(Peter, Acme);
9	subscribe(Acme, 2);

We note here again that formally each operation has an extra argument, namely the state to which the update is applied. Thus, we have apply(Peter, present-state) and hire(Peter, Acme, present-state). Assignments in programs do not explicitly mention environments but can be thought of as mapping a given environment into a new one (e.g. (x: = e) (present-environment) results in an environment in which if we queried the value of x, we would get back as the answer the value of e evaluated in present-environment). Analogously, we may think of updates as defining maps between states and so we obtain operations such as:
(apply(Peter))(present-state), (hire(Peter, Acme))(present-state).
Now a sequence of updates op1(arg1), op2(arg2), ..., opn(argn) applied to the state any_state takes the following form:
(opn(argn))((opn-1(argn-1))(...((op1(arg1))(any_state)) ...))
However, this reverses the sequentiality of operations in time and so we convert from the above prefix notation to a variant of the postfix notation to get:
(...(((any_state)(op1(arg1)))(op2(arg2)))...)(opn(argn))
Removing obtrusive bracketing and using the ';' symbol for 'concatenation' of operations we obtain the simpler form:
any_state; op1(arg1); op2(arg2);...; opn(argn)

3.1 Level 1: the intended trace

The intended trace includes all update operations, regardless of whether they succeed or fail to change the database. The trace must also start with the initialization update.

At this level the symbolic execution of updates is trivial. Each new operation is simply recorded at the end of the sequence that represents the trace. Processing the queries, on the other hand, is a somewhat more involved task. For example, if we want to find whether Peter works for Acme it is not sufficient to find a HIRE(Peter, Acme) in the recorded sequence; we have to check if the preconditions for the operation hold, and such tests may propagate backwards until the beginning of the sequence.

Fig. 1 gives an informal description of updates and queries as 'programmed' to work on sequences, and Table 2 contains the sequence obtained by the execution of the series of updates in our example (of Table 1). In Fig. 1, c denotes the present state, whose trace representation is the sequence recording the updates thus far executed.

Intended traces bring to mind the notion of audit trails, familiar from the database area. In fact, for auditing and for the study of usage patterns it may actually be useful to register such full traces.

Updates at the intended trace level

initag	c <– INITAG
c; apply(x)	c <– c * APPLY(x)
c; subscribe(y, v)	c <– c * SUBSCRIBE(y, v)
c; hire(x, y)	c <– c * HIRE(x, y)
c; fire(x, y)	c <– c * FIRE(x, y)

Queries at the intended trace level

iscandidate(x)	=	true, if APPLY(x) occurs in c and, for no y appearing in HIRE(x, y)'s, worksfor(x, y) is true in c
	=	false, otherwise
haspositions(y, n)	=	true, if $n = m - k$ and $m > 0$, where m is the sum of all positive mi in the SUBSCRIBE(y, mi)'s occurring in c, and k is the number of distinct x's appearing in HIRE(x, y)'s such that worksfor(x, y) is true in c
	=	false, otherwise
worksfor(x, y)	=	true, if c′*HIRE(x, y)*c″ occurs in c, where the indicated occurrence of HIRE(x, y) is the last one in c such that iscandidate(x) and haspositions(y, n) for $n > 0$, are true in c′, and if FIRE(x, y) does not occur in c″
	=	false, otherwise

Figure 1

Table 2. Intended trace

'INITAG *
APPLY(Peter) *
SUBSCRIBE(Acme, 1) *
APPLY(John) *
HIRE(John, Acme) *
HIRE(Peter, Acme) *
FIRE(John, Acme) *
HIRE(Peter, Acme) *
SUBSCRIBE(Acme, 2)'

3.2 Level 2: the effective trace

The effective trace includes only the updates that succeed in changing the database.

The execution of updates is still relatively easy, because only concatenations to the end of the sequence are performed, with the requirement, however, that the concatenation is not performed if the preconditions for the operation fail. A particular kind of precondition is that the desired effects should not already be present; for instance, one precondition for apply(x) is that x should not already be a candidate, for otherwise the (redundant) apply(x) would not succeed in changing the database. Queries become simpler than in the previous level; now for example, if we find HIRE(Peter, Acme) (and FIRE(Peter, Acme) does not appear after that) we can be sure that Peter works for Acme, without having to check the preconditions for the hire operation (i.e. without having to inspect the initial portion of the trace).

Fig. 2 defines the updates and queries, and Table 3 contains the sequence representing the trace for our example. As before, effective traces remind us of a familiar database concept: the logs that are kept for recovery purposes. More importantly one could think of databases where past information is not deleted but, instead, all information is time-stamped.[9]

Updates at the effective trace level

initag	c <– INITAG
c; apply(x)	c <– c * APPLY(x) if iscandidate(x) is false and for no y worksfor(x, y) is true in c
	<– c, otherwise
c; subscribe(y, v)	c <– c * SUBSCRIBE(y, v) if $v > 0$
	<– c, otherwise
c; hire(x, y)	c <– c * HIRE(x, y) if iscandidate(x) and haspositions(y, n) for $n > 0$ are true in c
	<– c, otherwise
c; fire(x, y)	c <– c * FIRE(x, y) if worksfor(x, y) is true in c
	<– c, otherwise

Queries at the effective trace level

iscandidate(x)	=	true, if APPLY(x) occurs in c and, for no y's appearing in HIRE(x, y)'s, worksfor(x, y) is true in c
	=	false, otherwise
haspositions(v, n)	=	true, if $n = m - k$ and $m > 0$, where m is the sum of all mi in the SUBSCRIBE(y, mi)'s occurring in c and k is the number of distinct x's appearing in HIRE(x, y)'s such that worksfor(x, y) is true in c
	=	false, otherwise
worksfor(x, y)	=	true, if c′*HIRE(x, y)*c″ occurs in c, where the indicated occurrence of HIRE(x, y) is the last one in c, and if FIRE(x, y) does not occur in c″
	=	false, otherwise

Figure 2

Table 3. Effective trace

'INITAG *
APPLY(Peter) *
SUBSCRIBE(Acme, 1) *
APPLY(John) *
HIRE(John, Acme) *
FIRE(John, Acme) *
HIRE(Peter, Acme) *
SUBSCRIBE(Acme, 2)'

3.3 Level 3: the current trace

At levels 1 and 2 the traces are ever-increasing. If we are not interested in keeping historical information but only those elements in a trace which are sufficient to characterize the current state of the database, we may want to consider ways to obtain compressed sequences. In fact, we may want to substitute a different but equivalent trace for the one if the second expresses the present state of the database in a more succinct manner.

The current trace contains only the updates whose effects still hold and, if fewer operations can produce the effects of a given series of operations, then the former is substituted for the latter. Two examples will illustrate such reductions.

(1) Since John has been fired from Acme, the effects of HIRE(John, Acme), no longer hold and can be removed from the sequence: notice that the combined effect of HIRE(John, Acme)*FIRE(John, Acme), for the purposes of characterizing the present state, is nil and so the second operation 'cancels' the first and none need be kept in the sequence.

(2) The net effect of Acme offering 1 position and then 2 more positions is the same as having offered 3 positions, and thus SUBSCRIBE(Acme, 1) * SUBSCRIBE (ACME, 2) can be replaced with SUBSCRIBE(Acme, 3).

Updates at the current trace level

initag	c < – INITAG
c; apply(x)	c < – c * APPLY(x) if iscandidate(x) is false and for no y worksfor(x, y) is true in c
	< – c, otherwise
c; subscribe(y, v)	c < – c * SUBSCRIBE(y, v) if v > 0 and for no w haspositions(y, w) is true in c
	< – c with SUBSCRIBE(y, n) replaced by SUBSCRIBE(y, n+v) if v > 0 and for some n haspositions(y, n) is true in c
	< – c, otherwise
c; hire(x, y)	c < – c * HIRE(x, y) if iscandidate(x) and haspositions(y, n) for n > 0 are true in c
	< – c, otherwise
c; fire(x, y)	c < – c without HIRE(x, y) if worksfor(x, y) is true in c
	< – c, otherwise

Queries at the current trace level

iscandidate(x)	=	true, if APPLY(x) occurs in c but not HIRE(x, y), for any y
	=	false, otherwise
haspositions(y, n)	=	true, if c contains an occurrence of SUBSCRIBE(y, m) and m – n occurrences of HIRE(x, y)'s
	=	false, otherwise
worksfor(x, y)	=	true, if HIRE(x, y) occurs in c
	=	false, otherwise

Figure 3

At this level the updates become considerably less simple, because an internal manipulation of the sequence is needed. For instance, the execution of subscribe(Acme, 1) is still simply a concatenation at the end of the sequence but subscribe(Acme, 2) causes the recorded SUBSCRIBE(Acme, 1) to be replaced (where it stands inside the sequence) by SUBSCRIBE(Acme, 3). Conversely, the execution of the queries is further simplified, since we do not have to search for 'negative' operations (such as fire) and because the information that remains is more concentrated (e.g. there is now a single SUBSCRIBE per registered company). Fig. 3 defines the updates and queries, and Table 4 the current trace for our example. Current traces can be related to efforts to optimize transactions,[10] that is, user interactions involving several operations within a single session.

Table 4. Current trace

'INITAG *
APPLY(Peter) *
SUBSCRIBE(Acme, 3) *
APPLY(John) *
HIRE(Peter, Acme)'

3.4 Level 4: the re-ordered trace

Even with the reductions at level 3, there may be more than one (reduced) series of operations leading from the itital state to the current state. This happens because, for some operations leading from the initial state to the current state, the end effects are the same regardless of the order of their execution. For instance, it does not matter who applies first, Peter or John, as also there is no fixed precedence between candidates applying and companies subscribing. On the other hand a hire(x, y) operation depends on the previous execution of apply(x) and subscribe(y, n), with n > 0.

Fig. 4 sketches the partial order characterizing the interdependences among operations, in the sense that operations that produce as effects the preconditions for other operations must be executed before the latter.

$$\begin{tikzcd}
 & \text{apply}(x') & & & \text{hire}(x', y) \\
\text{Initag} & \text{apply}(x) & \text{hire}(x, y) & \text{fire}(x, y) & \\
 & \text{subscribe}(y, 1) & & & \\
 & \text{subscribe}(y', n') & & & \text{hire}(x, y')
\end{tikzcd}$$

Figure 4

Given these considerations, we can say that the re-ordered trace includes the updates that would appear in a current trace, arranged in a way that is compatible with the partial ordering of the operations, and for operations whose order does not matter arbitrary ordering criteria are chosen.

Our arbitrary criteria are that all apply's are placed before all subscribe's and occurrences of the same operation are ordered lexicographically by the first argument. Thus, in the trace we shall have APPLY(John) before APPLY(Peter), assuming John < Peter.

At this level, the execution of updates involves, more than before, the manipulation of the sequence, since each additional update will be recorded in its proper place according to the chosen order. The execution of queries is as simple as at the previous level: the only difference is that one can take advantage of the order for speeding up the task of scanning the sequence. Suppose that we want to know if John currently works for Acme: we start scanning from the end of the sequence until one of the following cases occurs.

- HIRE(John, Acme) is found: worksfor(John, Acme) is true;
- HIRE(x, Acme) is found, where x < John: worksfor(John, Acme) is false;

– an operation is scanned that is not a HIRE (this is the case in our example): worksfor(John, Acme) is false.

Fig. 5 defines the updates and queries, and Table 5 contains the sequence representing the re-ordered trace.

Updates at the re-ordered trace level

initag	c < – INITAG
c; apply(x)	c < – c with APPLY(x) inserted in its proper place according to the order, if iscandidate(x) is false and for no y worksfor(x, y) is true in c
	< – c, otherwise
c; subscribe(y, v)	c < – c, with SUBSCRIBE(y, v) inserted in its proper place, if v > 0 and, for no w, haspositions (y, w) is true in c
	< – c, with SUBSCRIBE(y, n) replaced by SUBSCRIBE(y, n + v), if v > 0 and, for some n, haspositions(y, n) is true in c
	< – c, otherwise
c; hire(x, y)	c < – c, with HIRE(x, y) inserted in its proper place, if iscandidate(x) and haspositions(y, n), for n > 0, are true in c
	< – c, otherwise
c; fire(x, y)	c < – c, without HIRE(x, y) if worksfor(x, y) is true in c
	< – c, otherwise

Queries at the re-ordered trace level

Same as at the previous level, with the time-saving possibility to stop searching for the indicated occurrences as the scanning goes beyond their proper places according to the ordering.

Figure 5

Table 5. Re-ordered trace

'INITAG *
APPLY(John) *
APPLY(Peter) *
SUBSCRIBE(Acme, 3) *
HIRE(Peter, Acme)'

Re-ordered traces provide a unique way to represent equivalent series of operations. Again this has a parallel in data processing: if you want to compare two sets of items of information, perhaps stored on different tapes, you first perform a sort on both sets, using the same ordering criterion; then the comparison can be simply and efficiently performed in a merge-like fashion.

4. FROM THE INFORMAL DESCRIPTIONS TO SYMBOLIC PROCEDURES

Figs 6 and 7 contain procedural specifications of an update (subscribe) and a query (haspositions), respectively.[3] A procedural specification treats the trace as a sequence of symbols and 'implements' queries and updates as symbol manipulation procedures defined on the traces. The procedures (called **ops**) are fairly easy to understand for anyone with some experience of symbol manipulation, and can be promptly translated into any existing symbol manipulation language (e.g. SNOBOL, ICON, LISP, REDUCE, etc.). The statements in the body of a procedure are examined sequentially and the first one with a valid precondition (the condition to the left of the ' = > ' symbol) is executed. The procedure is then exited. By execution, we mean that the value of the expression specified on the right of the ' = > ' symbol is returned as the value of the procedure. If ' = > ' has no precondition, then it is executed whenever it is encountered. The **match** statement is executed by matching in order the input sequence of type agdb against the alternative patterns specified to the left of the ' = > ' symbols; the value of the expression specified to the right of the ' = > ' symbol whose precondition was the first successful match is the result of the entire statement.

Procedural specifications of subscribe

```
Level 1
  op subscribe (y:company, m:natural, s:agdb):agdb
    = > SUBSCRIBE[y|m|s]
  endop
Level 2
  op subscribe (y:company, m:natural, s:agdb):agdb
    m > 0 = > SUBSCRIBE[y|m|s]
    = > s
  endop
Level 3
  op subscribe (y:company, m:natural, s:agdb):agdb
    var z:person, t:agdb, w:company, n:natural
      (m > 0) = > s
    match s
      HIRE[z|w|t] ⇒ HIRE[x|w|subscribe (y, m, t)]
      SUBSCRIBE[w|n|t] = >
        if y = w
        then SUBSCRIBE[y|m + n|t]
        else SUBSCRIBE[y|n| subscribe(y, m, t)]
APPLY[z|t] = > APPLY[z|subscribe(y, m, t)]
      INITAG = > SUBSCRIBE[y|m|s]
    endmatch
  endop
Level 4
    op subscribe(y:company, m:natural, s:agdb):agdb
    var x:person, t:agdb, w:company, n:natural
      (m > 0) = > s
    match s
          HIRE[x|w|t] = > HIRE[x|w|subscribe(y, m, t)]
          SUBSCRIBE[w|n|t] = >
            if y = w
            then SUBSCRIBE[y|n + m|t]
            else if y > w
                then SUBSCRIBE[w|n|subscribe(y, m, t)]
                else SUBSCRIBE[y|m|s]
          APPLY[x|t] = > SUBSCRIBE[y|m|s]
          INITAG = > SUBSCRIBE[y|m|s]
        endmatch
      endop
```

Figure 6

Note the use of upper- and lower-case names for the queries and updates. The lower-case versions clearly describe procedures which implement the operation. The upper-case versions denote the record of the operation in the trace. Note also the use of the normal bracketing and commas for procedures as compared to square bracketing and vertical bars in the trace. Again, this arises from the need to distinguish an operation's implementation (and invocation) from the occurrence of operations

Procedural specifications of haspositions

```
Level 1
  op haspositions(y:company, m:natural, s:agdb):logical
    var x:person, t:agdb, w:company, n:natural
    match s
      INITAG = > false
      APPLY[x|t] = > haspositions(y, m, t)
      SUBSCRIBE[w|n|t] = >
        if w = y
        then haspstns(y, m − n, t)
        else haspositions(y, m, t)
      HIRE[x|w|t] = >
        if w = y and iscandidate(x, t)
          and haspositions(y, ?n, t) and n > 0
        then haspositions(y, m + 1, t)
        else haspositions(y, m, t)
    endmatch
  endop
Level 2
  op haspositions(y:company, m:natural, s:agdb):logical
    var n:natural, t:agdb, w:company, z:person
    match s
      INITAG = > false
      APPLY[z|t] = > haspositions(y, m, t)
      SUBSCRIBE[w|n|t] = >
        if y = w
        then haspstns(y, m − n, t)
        else haspositions(y, m, t)
      HIRE[z|w|t] = >
        if y = w
        then haspstns(y, m + 1, t)
        else haspositions(y, m, t)
      FIRE[z|w|t] = >
        if y = w
        then haspstns(y, m − 1, t)
        else haspositions(y, m, t)
    endmatch
  endop
Level 3
  op haspositions(y:company, m:natural, s:agdb):logical
    var z:person, w:company, t:agdb, n:natural
    match s
      INITAG = > false
      HIRE[z|w|t] = >
        if y = w
        then haspositions(y, m + 1, t)
        else haspositions(y, m, t)
      APPLY[z|t] = > haspositions(y, m, t)
      SUBSCRIBE[w|n|t] = >
        if w = y
        then if m − n = 0
          then true
          else false
        else haspositions(y, m, t)
    endmatch
  endop
Level 4
  op haspositions(y:company, m:natural, s:agdb):logical
    var z:person, w:company, t:agdb, n:natural
    match s
      INITAG = > false
      APPLY[z|t] = > false
      SUBSCRIBE[w|n|t] = >
        if w ≠ y
        then haspositions(y, m, t)
        else if m = n
          then true
          else false
      HIRE[z|w|t] = >
        if w = y
        then haspositions(y, m + 1, t)
        else haspositions(y, m, t)
    endmatch
  endop
```

Figure 7

and arguments as subsequences of the trace. Finally, notice that we have again formally included the trace (state) as a formal argument of the operations; in turn each update operation is described as a function returning a new state as its value. The resulting nested prefix notation is particularly suitable to recursive handling.

In the implementation of haspositions at the intended trace level, we search the trace and, depending on what subsequence we encounter next, return a new pattern (in this case just representing either true or false). Note that we need to use the hidden query (i.e. one that cannot be used directly by users of the database) haspstns. This query computes the number of positions represented by the trace by adding up subscribe's to the same company and subtracting hire's by the same company. We have also used the notation '?n' in the Level 1 definition of haspositions to indicate that the value of n is obtained from the matching process and is not an input to the procedure haspositions.

5. SOME BRIEF THEORETICAL REMARKS

Since the purpose of this paper is to provide a framework for a non-technical understanding of some concepts we believe will be useful to the practitioner, we have generally restricted our remarks on theoretical questions. We take this opportunity to provide for the interested reader a bridge from the earlier sections to relevant theoretical concepts. Reordered traces derive their importance from their connection with techniques for data type verification.

Thus the techniques we presented attempt to provide a methodology for developing axiomatisations which capture relevant intuitive details, simplify the specification problem by reducing the generality of the situations that have to be treated in the specification, and offer a criterion for determining whether a specification is in some sense complete by specifying relatively few standard sets of combinations of operations or queries on the re-ordered expressions. Each re-ordered trace denotes uniquely one valid data base state.

Techniques based on the concept of abstract data types also exist for defining implementations, proving the correctness of an implementation and studying other relevant properties of the specifications. Thus it is important that we establish some connection between the concept of traces as we have outlined it and the normal specification of data types. In fact, the transformations undergone by traces in moving from one level to the next can be formally expressed as equational axioms. Equational specifications are the basis of the current theories of abstract data types.

As in the notation used for the procedural specifications of the last section, we use the nested, prefix notation for equations and we again include the database state as a formal argument (parameter) of the updates and queries. Let us now examine the passage between the various levels to determine the kinds of equations needed in an axiomatization.

(*a*) *Level 1 to level 2.* If the preconditions for the update **op** to be applied to the state s fail, then the database state is unchanged. That is, **op**(..., a) = s. We used this kind of rule to eliminate from the trace operations which fail. Thus we could write the axiom

$n \leqslant 0 -> \text{subscribe}(y, n, s) = s$

where n is a natural number, y a company and s a database state. Thus, if the number of positions offered is non-positive, the effect of subscribe on the database state is nil.

(*b*) *Level 2 to level 3.* If, for two sequences of operations

op1(..., **op2**(... ..., **opn**(..., s)...)...)...)
and
op1′(..., **op2′**(......, **opm′** (..., s)...)...)

the precondition for initiating their execution is the same and their final effects are the same, we can then state that they are equivalent. This kind of equivalence is often used to reduce the length of the trace (and hence of the re-ordered trace eventually used) by substituting the shorter of two sequences for the longer one. Notice that the shorter sequence may be the empty sequence (i.e. just the database state with no update applied). For example, the final effect of hiring and firing the same person with respect to the same company is nil. A formal example using the subscribe operation is:

$m > 0 \text{ and } n > 0 ->$
$\quad \text{subscribe}(y, m, \text{subscribe}(y, n, s))$
$\quad = \text{subscribe}(y, m+n, s)$

where m, n are naturals, y a company, and s a database state. Thus the cumulative effect of various subscribes performed one after the other is reflected using one subscribe with the total of the positions being offered.

c) *Level 3 to level 4.* If, for two sequences of operations

op1(..., **op2**(... ..., **opn**(..., s)...)...)
and
op1′(..., **op2′**(... ..., **opm′**(..., s)...)...)

the precondition for initiating their execution is the same and their final effects are the same and, moreover, m = n (i.e. the sequences are the same length) and one is just a permutation of the other, then we can say that they are equivalent. This kind of equivalence is often used to achieve the total ordering of all possible update operations discussed in section 3.4. This is used to achieve the **canonical form** which is our objective in going through the various traces. As an example, we have:

$m > 0 \text{ and } n > 0 \text{ and } y \neq w ->$
$\quad \text{subscribe}(y, m, \text{subscribe}(w, n, s))$
$\quad = \text{subscribe}(w, n, \text{subscribe}(y, m, s))$

Thus two subscribes by two **different** companies can be re-ordered (they should be to achieve a re-ordered trace if the lexicographic order for the companies, which are the first argument, is incorrect).

We have had some experience with this methodology for defining canonical forms and then extracting an axiomatization from the procedural specification, and we can say that this experience has proved to be encouraging. The specification method based on canonical expressions has been put forward in a variety of places[11, 12] but it is not always clear how to choose canonical forms and how to choose what should go into the specification.

In this respect, we should note the connection between our assumptions and aims and those of reports where canonical forms and ideas of observability are assumed. For databases, it is quite reasonable to assume that database states differ only if queries can differentiate between them. We might call this concept observational completeness. Note that the traces at the very lowest level exhibit a very pure form of observational completeness. Any well-formed expression denoting a state can be examined by queries, and our specifications at this level must guarantee complete observability. This complete observability must be guaranteed at each subsequent level, but at each level the domain of expressions over which the queries are defined is reduced in scope. This is because we are defining equivalences among traces and providing ways of transforming traces which are not in the 'legal' set for that level to ones which are. These sequences are defined by stating equivalences between sequences of update operations, as in the examples presented above.

At the final level, expressions denoting states must be in canonical form before a query can be applied. The queries at this level guarantee observable completeness only for states denoted by canonical expressions. (This seems to be the meaning of Guttag's sufficient completeness criterion used in his work.[13] It is also what is meant by 'completeness criterion' in ref. 7.) However, we have defined enough rules of transformation via our procedural specifications and the equations obtained from them to guarantee the existence of an equivalent canonical expression denoting each state.

In summary, at the lowest level two expressions denoting a database state are equivalent if and only if there are no queries (predicates) which can differentiate between them. At the highest level, two expressions denoting a state are equivalent if and only if they have the same canonical form. Note that 'observational equivalence' is generally quite hard to prove, since one has to use some general form of reasoning about **all** queries. Testing whether two expressions have the same canonical form is generally much easier.

In future work, we hope to study the process outlined above for obtaining a specification in terms of canonical expressions. Each level of trace above can be seen as choosing a smaller and smaller class of representatives of each equivalence class of expressions, eventually ending up with the single canonical representative, and at the same time providing ways of transforming expressions not in the designated subset into equivalent ones within the subset. And, although we have used four levels of trace here, in the more general theory it is not necessary to fix the number of levels before applying the methodology. The nature of the application, the nature of the canonical form, and the transformations chosen will determine the number of levels needed. Moreover, we may not want to assume the observational completeness property if our application is not a database one.

Finally, we note that with the use of predicates in our axiomatization, we avoid the usual combinatorial explosion of selectors necessitated to query a relation. For instance, instead of needing 2**2 = 4 selectors to test the relation defined by haspositions (we do not count the state argument) for the various combinations of constants (given) and variables (to receive the selected values) which we can possibly use as arguments for a query, we use existentially quantified versions of the predicate in place of the selectors. For example, ∃n haspositions(Acme, n) should select the non-negative number of positions which Acme has available. ∃y ∃n haspositions(y, n) returns some company with the number of positions offered by that company.

6. AN 'ABSTRACT' DIALOG

AUTHORS. To summarize, we would say that it pays off to produce an executable specification and experiment with it, perhaps changing it several times as demanded by the future users, before committing oneself to a lengthy and costly implementation. Further, it is useful to have this specification cast in a style that favours rigorous verifications of correctness. Finally, the executable specification can be re-activated during the maintenance phase, in order to experiment with changes necessitated by shortcomings in the original design or by new needs of the user community.

PRACTITIONER. It is certainly nice to have an executable specification available with all those features. The idea of producing a first version for experimentation only has been in fact defended by software engineers[14] and has been used with good results reported.[15] However, I have certain misgivings as to the effort needed to produce it, particularly when formal methodologies are employed. First, let me point out the matter of scale. You were discussing a very small example. I fear that producing a specification for any realistic database application and making it reliably consistent would be a very difficult job indeed.

A. There are examples of non-trivial applications being specified using these techniques[16] and at least one of them (ref. 7) helped finding errors that several people had failed to detect in a previous informal specification, errors that would be tricky and expensive to correct in the programming phase.

P. Yes, but the specifications were done by university people. Also it is convenient to employ symbol-manipulation languages for handling the traces, whereas most professional programmers are not trained to use such languages. The size itself of the programming task in the case of realistic applications could be such that, for having the executable specification, we would perhaps double the time and cost of embarking on an implementation after a simpler requirements analysis.

A. Designing a database application is decidedly not an outright programming job, unless one confuses it with the mere superimposition of a database management system over a number of existing files. Highly trained people are needed, although, admittedly, not necessarily university people. Anyway, in order to simplify this task and also to take care of the bulk of the effort involved, certain software tools have begun to appear.[17, 18, 19]

P. How do I communicate my intentions to these systems? Can I just indicate the queries and updates with their preconditions and effects?

A. In general you are required to supply the equations (see section 5) which show what expressions are equivalent. But it may be possible to construct some interface enabling you to communicate through it as you said and having the system (perhaps with your help, interactively) derive the equations.

P. I look forward to seeing such software tools widely available and featuring user-friendly interfaces. It is fine to praise a formal 'non-procedural' methodology for its freedom from programming and other implementation details; but to dispense the users of the methodology from knowing about programming by demanding in exchange that they become logicians or mathematicians is totally unrealistic. Another objection that I have to the formal specifications that we have been discussing is that they leave out more than the structuring of data, which you claim to be convenient to postpone. Certain properties of systems such as time and space requirements, memory access patterns, reliability, synchronization and process independence are left out as well.[20]

A. This is true, yet an executable specification can help in preparing for the phase when those requirements will be considered, if you put it to a monitored experimental usage and collect some statistics. But in fact the main benefits of executable specifications, at the current state of the art, refer to testing the **behaviour** of a database application subjected to integrity constraints, thereby giving prospective users an opportunity to assess it and react to it.

P. I submit that we still have to wait for reports on real applications of all this by people working in the business environment, as has been done for structured programming and top-down design (in ref. 21 for example). Only from the analysis of such reports will one be able to settle the case of whether this line of research is relevant to practitioners, or if they remain exclusively as a contribution towards the understanding of data and data-handling functions.

REFERENCES

1. B. Liskov and S. Zilles, *An Introduction to Formal Specifications of Data Abstractions*, vol I, *Current Trends in Programming Methodology*, edited R. T. Yeh, N. J. Prentice-Hall, pp. 1–32 (1977).
2. M. L. Brodie and S. N. Zilles (eds), *Proceedings of a Workshop on Data Abstraction, Databases and Conceptual Modelling*. NBS and ACM (1981).
3. A. L. Furtado and P. A. S. Veloso, Procedural specifications and implementations for abstract data types. *ACM/Sigplan Notices* **16** (No. 3), 53–62.
4. E. H. Sibley, Database management systems: past and present. In ref. 2, p. 192.
5. C. C. Gotlieb. Some large questions about very large data bases. *Proceedings of the 6th Conference on Very Large Data Bases, ACM*, pp. 3–7 (1980).
6. B. Youmark, *The ANSI/X3/SPARC/SGDBMS Architecture*. Rand Corporation, Santa Monica, CA.
7. W. Bartussek and D. Parnas, *Using Traces to write Abstract Specifications for Software Modules*. UNC report 77–012, University of North Carolina at Chapel Hill (1977).

8. C. J. Sippl and C. P. Sippl, *Computer Dictionary and Handbook*. Howard W Sons & Co Inc. (1978).
9. J. A. Bubenko jr, On the role of 'understanding models' in conceptual schema design. *Proceedings of the 5th Conference on Very Large Data Bases, ACM*, pp. 129–139 (1979).
10. J. M. Smith and P. Y. T. Chang, Optimising the performance of a relational algebra database interface. *CACM* **18** (No. 10), 568–579 (1975).
11. J. A. Goguen, J. W. Thatcher and E. G. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types. In vol. IV, *Current Trends in Programming Methodology*, edited R. T. Yeh, N. J. Prentice-Hall, pp. 80–149 (1978).
12. T. H. C. Pequeno and P. A. S. Veloso, Do not write more axioms than you have to. *Proceedings, International Computer Symposium*, pp. 488–498 (1978).
13. J. Guttag, Notes on type abstraction (version 2). *IEEE Transactions on Software Engineering*, vol. 6 (No. 1) pp. 13–23 (1980).
14. F. P. Brooks, *The Mythical Man-month*. Addison Wesley (1979).
15. M. M. Astrahan *et al. A History and Evaluation of System R*. Research report RJ 2843(36129), IBM, San José (1980).
16. J. Guttag and J. J. Horning, Formal specification as a design tool. *Proceedings of the 7th Annual Symposium of Principles of Programming Languages*, pp. 251–261 (1980)
17. S. L. Gerhart *et al.* An overview of Affirm: a specification and verification system. *Proceedings IFIP*, pp. 343–348 (1980).
18. J. A. Goguen and J. J. Tardo, An introduction to OBJ: a language for writing and testing formal algebraic specifications. *Proceedings of a Conference on Specification of Reliable Software*, IEEE Comp. Soc. (1979).
19. R. W. Burstall and J. Goguen, Putting theories together to make specifications. *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pp. 1045–1058 (1977).
20. M. Shaw, *The Impact of Abstraction Concerns on Modern Programming Languages*. Technical report CMU-CS-80 = 116, Carnegie-Mellon University (1980).
21. J. B. Holton, Are the new programming techniques being used? *Datamation*, pp. 97–103 (July 1977).