PUC

Series: Monografias em Ciência da Computação

Nº 8/81

ON MULTI-LEVEL SPECIFICATIONS BASED ON TRACES

by

A.L. Furtado

P.A.S. Veloso

Departamento de Informática

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

Series: Monografias em Ciência da Computação

Nº 8/81

Series Editor: Marco A. Casanova July, 1981

ON MULTI-LEVEL SPECIFICATION BASED ON TRACES*+

A. L. Furtado

P.A.S. Veloso

* Research partly sponsored by FINEP and CNPq 402090 7 402097/80

+ Helpful discussion with T.S. Maibaum and J.M.V. de Castilho are gratefully acknowledged.

ABSTRACT

A methodology for the formal specification of data base applications is presented. Data base states are denoted by the traces of the update operations invoked. Using the knowledge of preconditions and effects of updates, procedures working on traces are designed to specify each update and query operation. By starting with the full traces, and then passing to modified traces, successive specifications are derived until a level corresponding to canonical terms is reached. At each level one has the symbolic execution capability, fundamental for the early usage and testing of specifications. The relationships between these specifications and those based on equations are examined.

Keywords: Data Bases, Formal Specifications, Abstract Data Types, Traces, specification Methodology.

RESUMO

Apresenta-se uma metodologia para a especificação formal de aplicações de bancos de dados, denotando-se cada estado do banco de dados por um traço, i.e. uma sequência de operações de atualização que o produz. Baseado no conhecimento das precondições e dos efeitos, escrevem-se procedimentos manipulando traços para especificar cada operação de atualização ou de consulta. Começando-se com traços completos obtém-se uma série de especificações, manipulando traços modificados, até se atingir o nível correspondente a termos canônicos. Cada um dos níveis oferece a possibilidade de execução simbólica, o que é fundamental para o uso e teste sem demora das especificações. Examinam-se também relações entre estas especificações e as algébricas, baseadas em equações.

Palavras chaves. Banco de dados , especificações formais, tipos abstratos de dados, traços, metodologia para especificação.

C.R. Categories: 4.33, 5.21, 5.24

1. Introduction

The methodology that we propose in this work to specify a data base application has as its starting point a description of the intended update and query operations expressed in the very same terminology of the application area.

The update operations are described by the preconditions for their application and by their effects; both preconditions and effects are observed through the query operations. In turn, the query operations are described by the way they are affected by the update operations.

From these descriptions, we pass to an executable specification, where the "programs" for updates and queries follow easily from the above-mentioned relationships among them. In this first specification , an update is executed by simply appending its name and actual parameters to a trace, where the previous updates have been likewise registered. A query is executed by inspecting the trace for the pertinent updates.

An advantage of this approach is that we avoid giving attention too early to concerns extraneous to the application area, related to the choice of a data model and how the data will be organized, while also avoiding the lack of precision of unlimitedly used natural language [Parnas]. Moreover we

seek to avoid the unrealistic requirement that the application area specialists become knowledgeable in the precise logical or mathematical formalisms, as the price for dispensing them from programming expertise.

The first specification can be refined through several levels. Passing to a next level corresponds to using more compact or differently ordered traces. This passage, which has always a simple intuitive meaning, provides a constructive way to derive presentations based on equations [Liskov and Zilles; Guttag; Goguen et al], the traces at the last level corresponding to canonical terms. This constitutes a bridge to the more complex techniques, which may be of interest whenever other appropriately trained persons are involved.

Although we restrict to the specified updates and queries the operations that will eventually be implemented, this does not imply that we shall end up with a closed data base, such as an airline reservation system. Exactly as in the general abstract data type approach, we assume that the operations will be available as procedures to be called from programs written in some general-purpose data base management system.

The discussion will be centered around a simple example, to be presented in the next section.

2. A simple example

As a simple example to illustrate the discussion, we shall use the data base of a company which markets a single kind of machine. The company can either:

A. lease a machine, or

B. sell a machine

to a customer. In both cases the customer uses the machine, but only in the second case he owns it. In case A, the customer can decide later to buy the machine. If a machine has been leased (but not bought), the customer can also choose to return it to the company. At any time a customer will have at most one machine.

The words underlined correspond to the update and query operations for our example data base, with the addition of the usual update initiate, which initializes the data base to an "empty" state. More specifically, the operations are:

Updates

u := initiate() - initializes the data base

u := lease(x,s) - a machine is leased to customer x

u := sell(x,s) - a machine is sold to customer x

u := return(x,s) - customer x returns the machine to the company

Queries

uses(x,s) - customer x uses a machine; true, if a

machine has been leased or sold to him,
and false otherwise

owns(x,s) - customer x owns a machine; true, if a
machine has been sold to him,
and false otherwise

In all operations above, except initiate, the last parameter s refers to the current data base state. For the update operations, u refers to the new state reached by applying the operation; so, updates are interpreted as functions, mapping states into states.

Usually queries can either be predicates or selectors, the latter being functions yielding certain components of a constructed data structure. It is interesting to note that the same operations can be taken as predicates or selectors. For instance:

owns(a,s)

where a is a constant (the name of a particular customer) is a case where owns is employed purely as a predicate. However:

$\exists v$ owns(v,s)

where v is a bound variable, confers to owns the double character of predicate and selector, if in addition to checking whether there exists some customer owning a machine we also assign to v the name of one such customer. This

corresponds to "computing" the Skolem function associated with the existentially quantified formula. In our work we have been using for this purpose the PLANNER-like notation [Hewitt]:

owns(?v,s)

We assume that a state s is fully characterized by what can be observed by applying the queries specified. More precisely, s can be denoted by the conjunction of all the positive ground instances of the predicates that hold in it. Another fundamental assumption is that at the initial state all predicates are false, and the only way to reach a state where a predicate yields a positive value is by executing one of the specified updates (or a sequence of them).

Thus the <u>effect</u> of updates is to change the logical value of certain predicates. Conversely, the logical value of certain predicates may constitute a <u>precondition</u> for the application of an update; if a precondition for an update fails no effects are produced, and so the state is not changed. The preconditions and effects of the update operations in our example are listed below.

<u>Preconditions and effects of updates</u>

u := initiate()

preconditions: none

effects: ∀x ¬uses(x,u) <u>and</u> ¬owns(x,u)

u := lease(x,s)

preconditions: ¬uses(x,s)

effects: uses(x,u)

u := sell(x,s)

preconditions: ¬owns(x,s)

effects: uses(x,u) <u>and</u> owns(x,u)

u := return(x,s)

preconditions: uses(x,s) <u>and</u> ¬owns(x,s)

effects: ¬uses(x,u)

In the graph in figure 1 the nodes are three particular data base states, labelled with the characterizing positive ground predicates. The edges represent transitions between states and are labelled with updates that, according to the above preconditions and effects, are the adequate means to perform the transitions. For convenience we omit the state parameter.

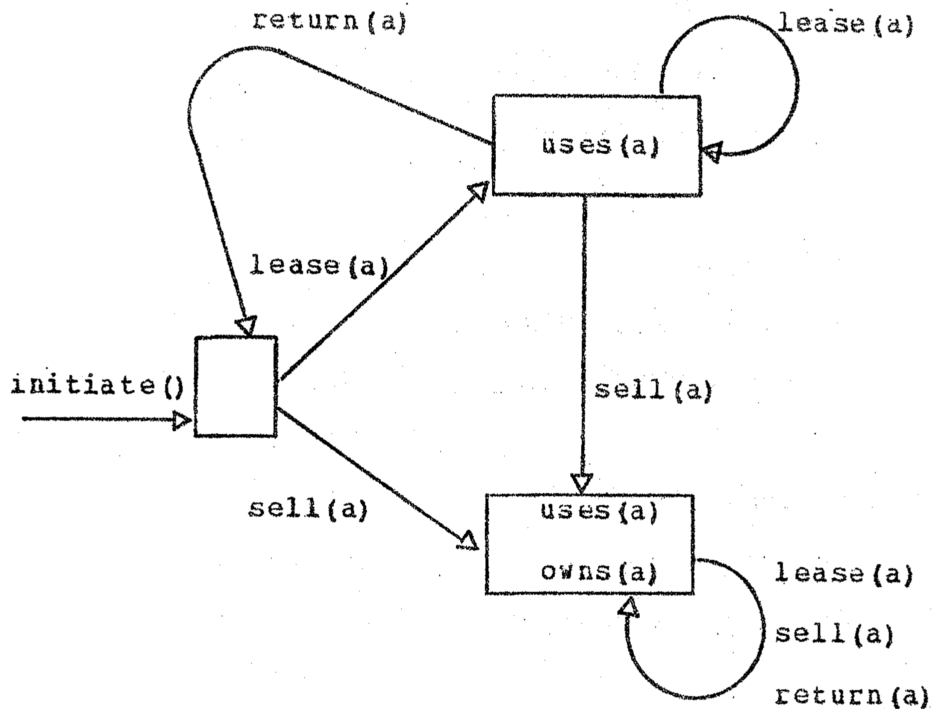

Figure 1: a state-space graph

By traversing the graph in various manners, but always using initiate as entry-point, and concatenating the edge-labels as they are encountered, we form sequences of operations (in fact, an infinite number of them), such as:

1 - initiate();lease(a);sell(a)

2 - initiate();sell(a)

3 - initiate();sell(a);return(a)

4 - initiate();lease(a);return(a)

The reader will notice, by inspecting figure 1, that sequences 1, 2 and 3 designate paths leading to the same state (where uses(a) and owns(a) hold). Similarly sequence 4 produces the same state as the "trivial" sequence initiate(). We are led to express this through equations, as:

```
initiate();lease(a);sell(a) = initiate();sell(a)
initiate();lease(a);return(a) = initiate()
```

or in a nested format, reintroducing the state parameter:

```
sell(a,lease(a,initiate())) = sell(a,initiate())
return(a,lease(a,initiate())) = initiate()
```

The equations express the fact that certainn sequences of operations are equivalent in that they lead to the same states. Thus, besides denoting a state by the positive ground predicates that hold in it (node labels) we have the alternative of denoting the state by the equivalence class of all the sequences of operations (composition of edge labels) leading to it.

Sequences of operations can be recorded as traces. The trace of sequence 3 is:

```
'INITIATE;SELL(a);RETURN(a)'
```

Since this trace is one of the names of the state also known as {uses(a),owns(a)} we should be able to ascertain the truth of both predicates by inspecting the trace. This is indeed possible; what we have to do is use our knowledge of preconditions and effects in "programming" each query. For instance, uses(a) can be processed as follows, by scanning the trace from left to right:

- INITIATE is encountered: ¬uses(a) and ¬owns(a) hold;
- SELL(a) is encountered: the precondition ¬owns(a) hold, and so the effects uses(a) and owns(a) are produced;
- RETURN(a) is encountered: the precondition uses(a) holds, but ¬owns(a) does not; hence no effect is produced and uses(a) is true at the end.

Similarly, the traces corresponding to sequences 1 and 2 could be employed with the same final result.

Our conclusion at this point is that traces can be used as (a universal kind of) data structures. Processing an update means then to manipulate a trace in some manner and processing a query means to inspect the trace. The "programs" for updates and queries should reflect the preconditions and effects of operations.

In the next section we shall examine the implications of

the existence, in general, of a multiplicity of traces naming each state.

3. Trace levels

Throughout this section we shall be referring to the sequence of operations given in figure 2, together with the positive ground predicates holding at the end of its execution.

```
initiate();lease(b);lease(a);sell(b);lease(c);
return(b);return(a);lease(c)


{uses(b),uses(c),owns(b)}
```

Figure 2: sequence of operations and state reached

Using the knowledge of preconditions and effects in order to derive the state from the sequence of operations in figure 2, the reader will notice that:

i. return(b) and the second activation of lease(c) will fail to alter the state, the former being an attempt to return a machine that has been sold and the latter being redundant.

ii. sell(b) will supersede (subsume) the effects of lease(b), and return(a) will cancel the effects of lease(a);

iii. the same final state would have been reached if lease(c) had been executed before sell(b), and so we are free to impose some arbitrary ordering criterion, such as the decreasing lexicographic order of customer names appearing in either lease or sell operations.

Let us now examine the traces given in figure 3, having the above considerations in mind. Trace 1.0 is an exact record of the example sequence of updates, whereas the others result from modifications suggested, respectively, by each of the three kinds of considerations above. Thus, in trace 2.0 the operations that cause no change of state are not included; in trace 3.0 the operations whose effects have been superseded or cancelled by other operations have been replaced or eliminated; finally, trace 4.0 contains the same operations as the previous one, but re-ordered according to the chosen criterion. Still, all four traces denote the same state, being therefore equally eligible as data structures to be generated along the execution of the example sequence of operations.

<u>trace 1.0</u>:	'INITIATE;LEASE(b);LEASE(a);SELL(b);LEASE(c); RETURN(b);RETURN(a);LEASE(c)'
<u>trace 2.0</u>:	'INITIATE;LEASE(b);LEASE(a);SELL(b);LEASE(c); RETURN(a)'
<u>trace 3.0</u>:	'INITIATE;SELL(b);LEASE(c)'
<u>trace 4.0</u>:	'INITIATE;LEASE(c);SELL(b)'

<u>Figure 3</u>: traces for the sequence of operations

The notion of <u>trace levels</u> comes as a consequence of this discussion. The point is that the structure of our symbolic trace-handling procedures for queries and updates will depend on the level of trace chosen. Here, we shall consider levels numbered like the corresponding example traces, the decimal position suggesting that other levels might conceivably be interpolated.

At level 1.0 the execution of the update operations is trivial. Executing one more update on the trace obtained thus far consists of simply adding the record of the operation to the trace. The queries, on the contrary, are relatively complex. For instance, on encountering an operation that would be appropriate to produce the effect being checked through the query, we cannot conclude that the effect holds, without looking for the preconditions of the operation (which in turn may depend on the successful execution of other operations, and so on).

For readability, we have been using the more conventional notation for traces, but the nested notation with the state parameter is more convenient for recursive handling and will be assumed for the trace parameters and variables (of type edb) in the procedures.

The procedures for the operations at the various levels will be written in the procedural presentation style, used in [Furtado and Veloso]. In the procedure bodies the statements are to be examined in sequence, the value returned by the procedure being the first expression to the right of a "=>" symbol, having on the left a true logical expression. If in a statement no logical expression is specified, the right-hand side expression is returned whenever the statement is encountered. The match statement is a case-like construct, inside which a recursive pattern-matching process takes place; the value of the statement is again the right-hand side expression whose left-hand side pattern was the first to match successfully the trace supplied as argument.

Figure 4 contains the procedures (called op's) for level 1.0 traces.

<u>Updates</u>

```
op initiate():edb
  => INITIATE
endop
```

```
op lease(x:customer,s:edb):edb
  => LEASE(x,s)
endop

op sell(x:customer,s:edb):edb
  => SELL(x,s)
endop

op return(x:customer,s:edb):edb
  => RETURN(x,s)
endop
```

Queries

```
op uses(x:customer,s:edb):logical
  var y:customer,t:edb
  match s
    INITIATE => false
    LEASE(y,t)  => if x = y then true
                   else uses(x,t)
    SELL(y,t)  => if x = y then true
                  else uses(x,t)
    RETURN(y,t)  => if x = y then owns(x,t)
                    else uses(x,t)
  endmatch
endop

op owns(x:customer,s:edb):logical
  var y:customer,t:edb
```

```
match's
  INITIATE => false
  LEASE(y,t) => owns(x,t)
  SELL(y,t) => if x = y then true
                  else owns(x,t)
  RETURN(y,t) => owns(x,t)
endmatch
endop
```

Figure 4: level 1.0 specification

At level 2.0 updates involve the previous testing of preconditions , but, if the preconditions hold they still simply add the record of the operation to the input trace. Queries become simpler, since the burden of testing preconditions has been shifted to the updates. Figure 5 contains the procedures for level 2.0 traces.

Updates

```
op initiate():edb
  => INITIATE
endop

op lease(x:customer,s:edb):edb
  uses(x,s) => s
  => LEASE(x,s)
```

```
endop

op sell(x:customer,s:edb):edb
  owns(x,s) => s
  => SELL(x,s)
endop

op return(x:customer,s:edb):edb
  owns(x,s) or ¬uses(x,s) => s
  => RETURN(x,s)
endop
```

Queries

```
op uses(x:customer,s:edb):logical
  var y:customer,t:edb
  match s
    INITIATE => false
    LEASE(y,t) => if x = y then true
                  else uses(x,t)
    SELL(y,t) => if x = y then true
                 else uses(x,t)
    RETURN(y,t) => if x = y then false
                   else uses(x,t)
  endmatch
endop

op owns(x:customer,s:edb):logical
```

```
var y:customer,t:edb
match s
  INITIATE => false
  LEASE(y,t) => if x = y then false
                 else owns(x,t)
  SELL(y,t) => if x = y then true
                else owns(x,t)
  RETURN(y,t) => if x = y then false
                  else owns(x,t)
endmatch
endop
```

Figure 5: level 2.0 specification

At level 3.0 updates become more complex, because an internal manipulation of the trace is required for replacing records of operations whose effects have been altered or eliminated. Queries are simplified, being processed on shorter traces and not having to account for all kinds of updates; notice, for example, that return will never be recorded on a trace - its execution simply deletes the corresponding lease.

Figure 6 contains the procedures for level 3.0 traces.

Updates

```
op initiate():edb
```

```
  => INITIATE
endop

op lease(x:customer,s:edb):edb
  uses(x,s)  => s
  => LEASE(x,s)
endop

op sell(x:customer,s:edb):edb
  var y:customer,t:edb
  owns(x,s)  => s
  match s
    INITIATE => SELL(x,s)
    LEASE(y,t)  => if x = y then SELL(x,t)
                   else LEASE(y,sell(x,t))
    SELL(y,t)  => SELL(y,sell(x,t))
  endmatch
endop

op return(x:customer,s:edb):edb
  var y:customer,t:edb
  owns(x,s) or ¬uses(x,s)  => s
  match s
    LEASE(y,t)  => if x = y then t
                   else LEASE(y,return(x,t))
    SELL(y,t)  => SELL(y,return(x,t))
  endmatch
endop
```

Queries

```
op uses(x:customer,s:edb):logical
  var y:customer,t:edb
  match s
    INITIATE => false
    LEASE(y,t) => if x = y then true
                  else uses(x,t)
    SELL(y,t) => if x = y then true
                 else uses(x,t)
  endmatch
endop

op owns(x:customer,s:edb):logical
  var y:customer,t:edb
  match s
    INITIATE => false
    LEASE(y,t) => if x = y then false
                  else owns(x,t)
    SELL(y,t) => if x = y then true
                 else owns(x,t)
  endmatch
endop
```

Figure 6: level 3.0 specification

Level 4.0 traces are of the same size as level 3.0 traces, differing only with respect to the ordering requirement. Updates bear the additional burden of maintaining the order. Queries do not become simpler but are somewhat more efficient, taking advantage of the ordering. We may stop searching for an operation if the scanning goes beyond the place in the trace where it could appear.

Figure 7 contains the procedures for level 4.0 traces.

Updates

```
op initiate():edb
  => INITIATE
endop

op lease(x:customer,s:edb):edb
  var y:customer,t:edb
  uses(x,s) => s
  match s
    INITIATE => LEASE(x,s)
    LEASE(y,t) => if x < y then LEASE(x,s)
                  else LEASE(y,lease(x,t))
    SELL(y,t) => if x < y then LEASE(x,s)
                 else SELL(y,lease(x,t))
  endmatch
endop
```

```
op sell(x:customer,s:edb):edb
  var y:customer,t:edb
  owns(x,s) => s
  match s
    INITIATE => SELL(x,s)
    LEASE(y,t) => if x = y then SELL(x,t)
                  else if x < y then SELL(x,s)
                       else LEASE(y,sell(x,t))
    SELL(y,t) => if x < y then SELL(x,s)
                 else SELL(y,sell(x,t))
  endmatch
endop

op return(x:customer,s:edb):edb
  var y:customer,t:edb
  owns(x,s) or ¬uses(x,s) => s
  match s
    LEASE(y,t) => if x = y then t
                  else LEASE(y,return(x,t))
    SELL(y,t) => SELL(y,return(x,t))
  endmatch
endop
```

Queries

```
op uses(x:customer,s:edb):logical
  var y:customer,t:edb
  match s
```

```
      INITIATE => false
      LEASE(y,t)  => if x = y then true
                     else if x < y then false
                          else uses(x,t)
      SELL(y,t)  => if x = y then true
                    else if x < y then false
                         else uses(x,t)
    endmatch
  endop

  op owns(x:customer,s:edb):logical
    var y:customer,t:edb
    match s
      INITIATE => false
      LEASE(y,t)  => if x ≤ y then false
                     else owns(x,t)
      SELL(y,t)  => if x = y then true
                    else if x < y then false
                         else owns(x,t)
    endmatch
  endop
```

Figure 7: level 4.0 specification

An obvious question, after considering these four different specifications for the same data base application is: what is the "best" level? All four specifications are

executable, and queries applied on the respective traces yield precisely the same answers. In fact, since the set of valid traces at a level is a subset of the valid traces at the previous one, the query procedures designed for a level will work correctly on traces of all subsequent levels (albeit less efficiently than those specifically designed for the more restricted traces). A criterion to choose the preferred specification might be the usefulness of the extra information contained in the traces. Particularly during the phases where the specification is being designed and tested as well as during the phase where the users are experimenting with it, full traces may be very convenient (traces in general are regarded as testing tools).

For accessing the extra information special operations must be specified. We recall that such information includes what operations have been unsuccessfully attempted, history of previous states and the chronological order of updates. Data bases where information pertaining to different states is kept as time-stamped records have been treated in [Bubenko].

At each level, only traces consistent with the requirements of the level will be generated. One may show that, at level 4.0, for each different state there will be a unique trace, which is thus the <u>canonical representative</u> of the equivalence class of all possible traces (at any level) denoting the state.

This uniqueness makes it trivial to show whether two sequences of operations lead to the same state: it suffices to execute both sequences using the level 4.0 procedures and check if the final resulting traces are identical.

At level 1.0 the execution of updates was trivially simple and, as more work was being shifted to updates at the successive levels, one would expect that the execution of queries would become trivial at level 4.0. However, this does not happen, although some simplification has been achieved. Let us examine the reasons for this fact.

In the data base practice the situation where an update affects a single query and a query is affected by a single update is <u>not</u> the usual one. If this situation occurred, queries and updates would organize the items of information (parameters of operations) in exactly the same way. One would feel justified to name an update dedicated to produce the effect observable by a query pred as assert_pred, for example. In this case we would be able to attain traces, registering such updates, on which the execution of queries would indeed be trivial.

Yet the usual situation is one where an update affects several queries, and conversely. Thus, the ultimate simplification of queries will generally require that we pass from the trace representation to a representation where the positive ground predicates are registered as the result of executing the operations [Veloso, Castilho and Furtado]. It

is as if, as each update takes place, we would "pre-process" the queries. Systems based on logic [Gallaire and Minker] in fact take this approach, using positive literals (positive ground predicates) as data structures.

Another remark about the use of traces at various levels for constructing specifications is that one may follow a different strategy than that of procedural presentations. In the latter, we can show that, by starting with the "empty" trace and then manipulating the traces only through the update procedures of a certain level, we only generate traces that are valid with respect to the level. The alternative strategy uses equations, a subject that was mentioned in passing in section 2. In the next section we shall discuss the relationships between procedural specifications at multiple levels and equations.

4. From procedures to equations

One finds in the literature on abstract data types two distinct viewpoints on what constitutes a "good" specification. Using the terminology of queries and updates they may be stated roughly as follows:

i. I do not care about the states as long as my specification enables me to find out the answers to any queries (this corresponds to the sufficient completeness of [Guttag]). In other words, traces on which exactly the same queries (predicates) yield a positive logical result are indistinguishable.

ii. Besides answering queries, I want to find out the results of updates (this corresponds to the initial algebra approach of [Goguen, Thatcher and Wagner]). Here, we should be able to transform a trace into another (or both into the same canonical representative) if we want to characterize the two traces as equivalent.

For the latter viewpoint only trace level 4.0 is adequate, in that each state is represented by a unique trace. For the former point of view, on the other hand, any one of the four levels is adequate.

We have seen that as we go from trace level 1.0 to trace level 4.0 the complexity of the corresponding procedural specifications shifts its center from the queries towards the updates. Let us examine now what is involved in transforming a trace at level 1.0 into a corresponding one at another level.

For this purpose we shall use as an example the sequence of operations in figure 2, with the traces of figure 3.

We shall consider two kinds of equations: Q-equations, related to the query operations, and U-equations, related to the update operations. As will be seen, they closely resemble the statements in the symbolic procedures. However there is a striking difference in that we cannot require that they be applied to traces in a predetermined sequence. As a consequence the Q-equations for unrestricted 1.0 traces cannot be simplified at the subsequent levels as the

procedures were. Figure 8 contains the Q-equations. They are the only equations needed in the level 1.0 specification.

1. uses(x,initiate()) = false
2. uses(x,lease(x,s)) = true
3. x ≠ y -> uses(x,lease(y,s)) = uses(x,s)
4. uses(x,sell(x,s)) = true
5. x ≠ y -> uses(x,sell(y,s)) = uses(x,s)
6. owns(x,initiate()) = false
7. owns(x,lease(y,s)) = owns(x,s)
8. owns(x,sell(x,s)) = true
9. x ≠ y -> owns(x,sell(y,s)) = owns(x,s)

---

10. uses(x,return(x,s)) = owns(x,s)
11. x ≠ y -> uses(x,return(y,s)) = uses(x,s)
12. x ≠ y -> owns(x,return(y,s)) = owns(x,s)

Figure 8: Q-equations

In going from level 1.0 to level 2.0, we have to discard operations that caused no change of state, because their preconditions were not satisfied. This is what the first four (conditional) U-equations of figure 9 assert. Notice that the first three refer to redundant updates, whereas the fourth one concerns violation of requirements.

Since we can determine the results of queries, we can use them in U-equations to eliminate from the level 1.0 trace

the application of return(b) and the second activation of lease(c), thereby converting it into a level 2.0 trace. Note that a "simplified" equation (adapted from the uses procedure) like

10'. uses(x,return(x,s)) = false

that we might be tempted to substitute for equation 10 would be incorrect, since we cannot assume that the appropriate U-equations would be applied, prior to the application of 10', to eliminate any invalid return's.

At level 3.0, in addition, we discard updates whose effects have been later cancelled or superseded. This is asserted in U-equations 5 (which states that a return cancels a lease) and 6 (asserting that a sell supersedes a lease).

However what these two equations actually state is that an update return or sell cancels or supersedes an immediately preceding application of lease. They are not enough to cancel the application of lease(b) from the level 2.0 trace in figure 3, since it is not adjacent to the application of sell(b).

What we need now is U-equations allowing us to permute updates so that the trace can be brought into condition for the application of equations 5 and 6. This is the purpose of the commutativity equations 7 through 11. These fall into two

categories: unconditional commutativity (such as equations 7, 8, 9, 10), and conditional commutativity (such as equation 11 which only holds for different parameters).

Using these equations we can rearrange the level 2.0 trace until it is ready for the application of equations 5 and 6, thereby obtaining the level 3.0 trace. Note, in particular, that no application of return will remain; that is why we do not need an equation for permuting return's. For the same reason we can dispense with the Q-equations 10, 11 and 12 in the level 3.0 specification. If these equations are no longer present no Q-equation will apply to, say, uses(x,return(x,...)), but the remaining Q-equations will apply, correctly, after the return's have been removed by applying U-equations. It seems fair to assign to this level techniques where only certain updates called constructors are kept in the traces [Guttag; Bartussek and Parnas].

Finally, we may rearrange the updates in the level 3.0 trace into the order of the level 4.0 trace by means of commutativity equations. In general this last step may need some commutativity equations in addition to those used in the previous passage.

This classification of equations does not attempt to be exhaustive. Rather, the intention is pointing out the kinds of symbolic manipulations related to the various trace levels.

```
1.  uses(x,s) = true   ->  lease(x,s) = s
2.  owns(x,s) = true   ->  sell(x,s) = s
3.  uses(x,s) = false  ->  return(x,s) = s
4.  owns(x,s) = true   ->  return(x,s) = s
--------------------------------------------------
5.  return(x,lease(x,s)) = return(x,s)
6.  sell(x,lease(x,s)) = sell(x,s)
--------------------------------------------------
7.  lease(x,lease(y,s)) = lease(y,lease(x,s))
8.  sell(x,sell(y,s)) = sell(y,sell(x,s))
9.  lease(x,sell(y,s)) = sell(y,lease(x,s))
10. return(x,sell(y,s)) = sell(y,return(x,s))
11. x ≠ y -> return(x,lease(y,s)) = lease(y,return(x,s))
```

Figure 9: U-equations

5. Conclusion

We have shown that the specification methodology discussed relies on the knowledge of the application area involved. The symbolic procedures specifying the application-oriented operations can be easily translated into some symbol-manipulation language (e.g. SNOBOL, ICON, LISP, REDUCE), thereby permitting early experimental usage and testing of the specifications [Furtado, Veloso and Castilho].

The procedures are meant to be implemented eventually as sub-programs to be called from programs in some general purpose data base management system (DBMS). Therefore our approach is neither so rigid as that of the "canned transactions" of closed systems, nor so free and flexible as would allow the unrestricted use of DBMSs' primitive update and query commands.

However, by losing some freedom and flexibility we are given an effective way to enforce semantic integrity constraints. One can show that, if our example data base is handled only through the sub-programs implementing the update procedures, then (in the data base) customers will use at most one machine at any time, and machines that have been bought will not be returned. By judiciously granting to users the right to call only the query and update sub-programs referring to what they are entitled to observe or modify, authorization constraints can be conveniently enforced as well.

References

W. Bartussek and D. L. Parnas, Using traces to write abstract specifications for software modules, UNC Report 77-012, Univ. of North Carolina at Chapel Hill, 1977.

J. A. Bubenko jr., On the role of 'understanding' models in conceptual schema design, Proc. 5th Very Large Data Bases Conference, 1979, pp. 129 - 139.

A. L. Furtado and P. A. S. Veloso, Procedural specifications and implementations for abstract data types, ACM/Sigplan Notices, vol. 16, n. 3, 1981, pp. 53 - 62.

A. L. Furtado, P. A. S. Veloso and J. M. V. de Castilho, Verification and testing of S-ER specifications, Proc. 2nd International Conference on Entity-Relationship Approach, to appear in 1981.

H. Gallaire and J. Minker (eds.), Logic and Data Bases, Plenum Press, 1978.

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

J. Guttag, Notes on type abstraction (version 2), IEEE Transactions on Software Engineering, vol. 6, n. 1, 1980, pp. 13 - 23.

C. Hewitt, Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot, Ph.D. thesis, Dept. of Math., MIT, 1972.

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

D. L. Parnas, The use of precise specifications in the development of software, Information Processing 77, B. Gilchrist (ed.), North-Holland 1977, pp. 861 - 867.

P. A. S. Veloso, J. M. V. de Castilho and A. L. Furtado, Systematic derivation of complementary specifications, Proc. 7th Very Large Data Bases Conference, to appear in 1981.