MULTI-LEVEL SPECIFICATIONS BASED ON TRACES

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

Dept. Informática, Pontifícia Universidade Católica
22453 Rio de Janeiro, RJ; Brazil

ABSTRACT

A methodology for the formal specification of data base applications is presented. Central to the methodology is the idea of proceeding through successive levels of "traces", which enables an initial algebra specification to be obtained from one based on preconditions and effects. Besides being executable these trace levels provide a way to go from a natural language specification to an algebraic one.

INTRODUCTION

This paper proposes a methodology to specify a data base application using the very same terminology of the application area. Starting from an informal, natural-language specification one passes to a description of update and query operations by means of preconditions and effects. From this description one can pass easily to an executable specification where "programs" manipulate "traces"[1]. This specification can be refined by proceeding via a series of levels of trace (which have simple intuitive meanings and are sufficiently complete[4]). From a level of unique representatives one can then pass, if desired, to an algebraic specification.

This approach permits avoiding premature consideration of details extraneous to the application area (as choice of data model, data organization, etc.), while avoiding lack of precision due to free usage of natural language[6]. However, the adoption of this abstract data type approach, does not mean that one ends up with a closed data base such as an airline reservation system; it only means that the otherwise unrestricted application programs can only manipulate the data base through the predefined operations.

A SIMPLE EXAMPLE

Consider the data base of a company ACME, marketing a single kind of machine. ACME can either lease or sell a machine to a customer. In both cases, the customer uses the machine but only in the latter does he own it. In the former case the customer can decide to buy the machine. If a machine has been leased but not bought the customer can choose to return it to ACME. We assume each customer to have at most one machine.

The words underlined, together with phi (which initializes the data base to an "empty" state) corresponds to the 4 updates and 2 queries available.

We shall be using queries to form predicates and selectors, by taking advantage of a PLANNER-like[5] notation. For instance, if A is a name of a customer, owns(A,s) is a predicate as is owns(?,s) (short for $\exists v$owns(v,s)). On the other hand, owns(?v,s) is a selector in that in addition to checking whether there exists some customer owning a machine it also assigns to v the name of one such customer.

We assume each data base state s to be observable by means of the queries, in that s can be completely characterized by the positive ground instances of the predicates holding in it.

Thus the effect of an update is to change the logical value of certain predicates, which may constitute a precondition for the application of other updates. (By convention, if the preconditions for an update fail the state remains unchanged).

This leads to a formal specification of updates by means of preconditions and effects. For instance, the precondition for the update t:=sell(x,s) is $\neg$owns(x,s) and its effects is uses(x,t) $\wedge$ owns(x,t). For t:=return(x,s), the precondition is uses(x,s) $\wedge \neg$owns(x,s) and the effect is $\neg$uses(x,t).

TRACE LEVELS

One can visualize a state graph, where each node is labelled by a set of positive ground predicates and edges are labelled by update operations. In this graph, states are represented by the nodes. But we can conceive an alternative, namely, representing a state by a sequence of updates (edge labels) leading to it. These sequences of operations can be recorded as traces[1].

In fact, by inspecting a trace, one can determine the result of a query. Also, processing an update consists of manipulating the trace. So, traces can be used as (a universal kind of) data structures.

One can start by denoting each state by the actual record of the sequence of operations that created it. In this case, executing an update consists simply of appending the name of the operation to the trace. On the other hand, the execution of queries is more complex: some of the operations recorded in the trace may have had no effect or may have been cancelled or superseded by others. However one can write a procedure *owns* to evaluate the query owns on such traces as follows.

```
op  owns(x:customer, s:state) : logical
    var  y:customer, t:state
    match s
      phi ⇒ 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
```

This procedure is written in a procedural style of presentation[2]: the match statement is a case-like construct, inside which a recursive pattern-matching process takes place; the value of the statement is the value of the expression to the right of a '⇒' whose lefthand side matches successfully the trace supplied as argument. In this case, we have *owns*(x,s) = true iff the trace s has the form

....sell(x,....)....

The above traces keep an actual record of all the updates invoked. So, they contain some redundant information as far as an answering queries is concerned. This remark leads to the idea of "levels of trace": by discarding redundant information one gains the flexibility of restructuring the sequences of updates.

Here we shall be considering 4 levels of trace, called 1.0 through 4.0 (the decimal notation suggests that other intermediate levels may be conceived). We call level 1.0 the one just considered, where traces are the actual records of the sequences of updates, analogously to the so-called "audit trails".

At level 2.0 updates involve the previous testing of preconditions: if they hold, and only then, the name of the operation is appended to the trace. Queries become simpler as the burden of testing pre conditions shifts to the updates. (Level 2.0 traces are like "logs" kept for recovery purposes). The procedure corresponding at level 2.0 to the update lease is

```
op  lease(x:customer, s:state) : state
    uses(x,s) ⇒ s;
    ⇒ lease(x,s)
endop
```

whereas for the query owns we have

```
op  owns(x:customer, s:state) : logical
    var  y:customer, t:state
    match s
      phi ⇒ 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
```

At levels 1.0 and 2.0 traces become longer with time. Now at level 3.0 the traces record only those operations whose effects have not been superseded or cancelled. For instance a level-3.0 trace corresponding to

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

At level 3.0 the procedures for updates become more complex, e.g.

```
op sell(x:customer, s:state) : state
   var  y:customer, t:state
   owns(x,s) ⇒ s;
   ¬uses(x,s) ⇒ sell(x,s);
   match 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
```

On the other hand, queries become simpler not having to account for all kinds of updates. (Notice, for instance, that return will never be recorded on a level-3.0 trace, its execution simply deletes the corresponding lease.)

There can be more than one level-3.0 trace denoting a state. Certain sequences of updates will cause the same end effects no matter the order in which they are executed. So, we are free to impose some external ordering criterion, such as decreasing lexicographic order of customer names.

On level 4.0 traces have the same size as level 3.0, differing only with respect to the ordering requirement. Now, updates bear the additional burden of maintaining the order, e.g.

```
op sell(x:customer, s:state) : state
   var  y:customer; t:state
   owns(x,s) ⇒ s;
   match s
     phi ⇒ 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
```

Compared to level 3.0 the new procedures for queries do not become simpler but can be made more efficient by taking advantage of the ordering.

A natural question, due to the multiplicity of trace-level specifications for the same data base application, is: what is the "best" level? All four specifications considered here are executable and sufficiently complete[4]. (Notice that query procedures designed for a level work correctly - albeit less efficiently - on traces of subsequent levels.) While designing and testing the specification it appears that the extra information kept on levels 1.0 and 2.0 may be useful. For other purposes other levels may be more convenient. For instance, the uniqueness of level-4.0 traces for each state can be very handy in showing whether two sequences of updates lead to the same state.

ALGEBRAIC SPECIFICATION

One finds in the abstract data type literature two viewpoints on what is a "good" specification. Namely, sufficient completeness[4] and initial (or

terminal) algebra[3]. For the latter viewpoint only trace level 4.0 is adequate, whereas all levels are adequate for the former.

We shall now indicate how an algebraic specification can be methodically obtained from the preceding ones. We shall consider two kinds of equations: Q-equations and U-equations (related to queries and updates, respectively). They closely resemble the statements in the symbolic procedures, but for the fact that one cannot require equations to be applied to traces in a predetermined order.

Twelve equations like the following ones describe the result of the level-1.0 query procedures and form a level-1.0 specification

$$\underline{uses}(x, \underline{sell}(x,s)) = \underline{true}$$
$$\underline{owns}(x, \underline{lease}(y,s)) = \underline{owns}(x,s)$$
$$x \neq y \rightarrow \underline{owns}(x, \underline{sell}(y,s)) = \underline{owns}(x,s)$$

In going from level 1.0 to level 2.0 one has to discard operations that caused no change of state. This is what the following 4 (conditional) U-equations

$$\underline{uses}(x,s) = \underline{false} \rightarrow \underline{return}(x,s) = s$$
$$\underline{uses}(x,s) = \underline{true} \rightarrow \underline{lease}(x,s) = s$$
$$\underline{owns}(x,s) = \underline{true} \rightarrow \underline{sell}(x,s) = s$$
$$\underline{owns}(x,s) = \underline{true} \rightarrow \underline{return}(x,s) = s$$

At level 3.0 in addition one discards updates whose effects were cancelled or superseded. This is asserted in the following 2 U-equations

$$\underline{return}(x, \underline{lease}(x,s)) = \underline{return}(x,s)$$
$$\underline{sell}(x, \underline{lease}(x,s)) = \underline{sell}(x,s)$$

However, these equations apply only to adjacent updates. Further U-equations are needed to take care of other cases. Five commutativity U-equations like the following ones will do the job

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

By means of these equations one can take any trace (from any level) and rearrange it into an equivalent level-4.0 trace.

CONCLUSION

The several levels of formal specification are all useful by themselves, besides providing a constructive methodology to go from the starting informal specification to less intuitive formal ones[7].

From the broader viewpoint of software engineering, we think that the notion of trace levels is important in that it relates the two apparently disjoint approaches of sufficient completeness and initiality.

REFERENCES

1. 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.
2. 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.
3. 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.
4. J.Guttag, Notes on type abstraction (version 2), IEEE Transactions on Software Engineering, vol. 6, n. 1, 1980, pp. 13-23.
5. 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.
6. 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.
7. 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, 1981, pp. 409-421.