

TOWARDS MULTI-LEVEL AND MODULAR CONCEPTUAL SCHEMA SPECIFICATIONS†

ULRICH SCHIEL‡

Departamento de Sistemas e Computação Universidade Federal da Paraíba, Campina Grande/PB, Brazil

ANTONIO L. FURTADO

Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, Brazil

ERICH J. NEUHOLD§

Institut für Informatik, Universität Stuttgart, Stuttgart, West Germany

and

MARCO A. CASANOVA

Centro Científico de Brasília, IBM do Brasil, Brasília, Brazil

(Received 21 April 1982; in revised form 4 June 1983)

Abstract—The specification of the conceptual schema for a data base application is divided into levels. It is argued that, at the highest level, a direct description of the characteristics of the information kept in a data base and of the constraints governing their existence and transformation of its components characterizes what a particular data base is in a more fundamental way (hence at a higher and more stable level) than the operations that happen to be used for data base manipulation. At a next lower level a specification based on operations, using the encapsulation strategy of abstract data types, is introduced, followed by a specification based on representations used in semantic data models. The discussion includes constraints involving temporal aspects. Modularization is also discussed as another dimension in the specification process, orthogonal to the division into levels.

1. INTRODUCTION

At present the specification methodologies most widely used in practice are based on conventional data models such as the relational, the network or the hierarchical model. An advantage of this approach is that corresponding to these data models a number of data base management systems are available. Thus the way towards an actual implementation is clearly identified. Semantic database models, although not so directly implementation oriented, still force the designer to immediately classify the universe of discourse according to the requirements of the data model into notions such as classes, attributes, relationships, values and hierarchies.

If we establish a parallel between the treatment of ordinary data structures in the area of programming languages with the representation trends in the data base area, we feel justified to compare the first approach with the specification of, say, queues, by showing how they can be represented as arrays of circular lists. Arrays and circular lists are, in a sense, different models for representing queues, very much like tables (the relational model) or some ring-like structure (CODASYL sets) are used to represent a

particular data base. We shall say that such approaches characterize the *representation level* specification or the *application modelling*. The resulting product of this level represents a *conceptual schema*. The *application implementation*, which produces an *internal schema*, will not be considered in this paper. Especially when the data model is very abstract (i.e. free from machine oriented data structures) it is fair to compare the representation level with the "use of a fixed discipline"[30], such as V-graphs or sets, as a specification technique.

In spite of its obvious merits, the representation level is no longer the one recommended by programming language researchers to be taken as the initial model. It has been argued that representations involve problems that are specific to the model itself and have nothing to do with the information that we originally wanted to specify. Looking too early into those problems distracts the specifiers' attention and may bias the entire model in an undesired fashion. It has the designers worrying about normalization, introduction of "dummy" record types, decisions about what segment should be chosen as parent in hierarchical relationships, classification of an object as an entity or as an attribute, etc. before the application area has really been understood.

The axiomatic and the algebraic approach to abstract data types[35, 22, 24] have been proposed as representation-free specification techniques. Instead of relying on representation they rely on the operations used to manipulate (*update operations*, in data base jargon) or interrogate (*query operations*) the

†Partial financial support from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), the Stifterverband der Deutschen Wissenschaft and FAPESP under nb. 82/0043-1 is gratefully acknowledged.

‡Presently at Institut für Informatik, Univ. Stuttgart.
§On temporary leave to Universidade de São Paulo, Brazil.

instances of the data type. The fundamental principle in such approaches is that a specification can be achieved by indicating how updates affect the results of queries[36]. Also the pre- and post-conditions of updates can be expressed in terms of expressions involving queries[45]. We shall say that these approaches correspond to the *operations level* of the specification.

The phrase *integrity constraints* is conspicuously absent in the programming language papers on abstract data types. As data base researchers have noted [46], integrity constraints do not appear explicitly in specifications at the operations level, since they are implicit in the pre-conditions/post-conditions expressed in the axioms or equations that govern the behaviour of operations. Conceivably this level is good enough for the programming languages area, where programs and operations play a fundamental role. Yet for data bases, where the more permanent and important role of information as contrasted to the relatively transient repertoire of operations used for its manipulation is recognized[4], an operations-free level seems to be desirable. Also, we should be able to express the integrity constraints *explicitly*, without having to deduce them from whatever the operations happen to do ([46, 7], p. 72).

Logic has been exerting an important role in this respect[20, 29, 14]. It is possible to specify a data base state by a conjunction of ground positive predicates. In turn a data base application is specified by declaring the predicate types (which may be seen as a first version of the basic query operations) and the *static* and *transition* (dynamic) *integrity constraints* restricting, respectively, the valid states and the valid transitions between states[12, 23]. We call this level the *states/transitions level*. Our three level approach is well in the line of recent proposals[28, 23]. In other references it is suggested that the operations level itself be developed again in a multi-level fashion[45, 16].

Considering the entire application design a four step process

- (1) Informal application description
- (2) Formal application description
 - states/transitions level*
 - operations level*
- (3) Application modelling
 - representation level*
- (4) Application implementation

this paper is concerned with the two intermediate steps. The continuation to step 4 was analysed in[41]. The first formal layer, the states/transitions level, will be extracted from the informal description of the information that is to be maintained in the data base. It is expected to be the most stable description, although it may still happen that new kinds of facts will become part of the data base (and hence new predicates will be introduced), and that the rules (and hence the constraints) will change. The states/transitions level is explained in Section 2.1

Also as part of the formal application description,

but as a lower layer, appears the operations level. Here a repertoire of update operations is chosen, their pre- and post-conditions ensuring that no static or transition constraints are violated. Perhaps not all states and transitions allowed at the states/transitions level will be achievable (i.e. the first level can be more general than the intermediate one). The operations level will usually not be as stable since basic update operations may be added and existing ones deactivated in correspondence to the more frequently changing usage patterns of the data base; such changes will require a revision of the pre-conditions/post-conditions interplay of operations. Passing from the states/transitions level to the operations level corresponds to the notion of *refinement* [12]. Properties that should be verified at these two levels, such as consistency, non redundancy, etc. have been formalized in[10].

After discussing the operations level in Section 2.2 the application modelling at the representation level will be introduced in Section 2.3. The formal application description is used to design the conceptual schema using a semantic database model whose features will be gradually introduced along the paper, as they are needed. The sorts and predicates become classes and relationships and the operations are translated into “procedures” using the data model language DML. The correctness of the representation must then be proven (much like the correct representation of queues by circular lists is proven in ([24] p. 75; for a data base application see [19]).

In Section 3 we discuss special problems which arise when the enforcement of certain transition constraints requires:

- considering information existing at states other than the current state;
- explicit reference to time;
- distinguishing situations where a transition *can* from situations where a transition *must* take place, the latter case reflecting the presence of a capability of the “system” to initiate an action by detecting an *event*.

Since the conceptual schemas of realistic data bases will generally be of considerable size and complexity *modularization* techniques are mandatory[46] and shall be briefly discussed in Section 4, taking into consideration notions such as abstractions[44, 39], construction of theories from other theories[8] and frames[33, 38]. Modularization must not be confused with external schema design. Modules are semantically closed and stable units of a system in whose scope operations over the conceptual schema will be defined. An external schema represents a user view, using pieces of several modules and must be changeable, as user requirements change. A user operation can have side effects on pieces outside of his schema and the data of one external schema can be affected by operations on another external schema.

A simple example will be used to illustrate the discussion and to convey a more intuitive understanding of the proposed approach.

2. THE THREE LEVELS OF SPECIFICATION

Specification at each level will be illustrated through the simple example[45, 19] of an employment agency data base.

2.1 The states/transitions level

The set S of valid states is formed by all states that can be denoted by a conjunction of positive instances of the predicates:

- iscandidate(x)—person x is a candidate applying for a job
- haspositions(y, n)—company y has n positions open
- worksfor(x, y)—person x works for company y
- hassalary(x, s)—person x has salary s
- minimum-salary—a constant indicating the current minimum salary

and, in addition, set S obeys the following static constraints:

- S1. $\forall x (\text{iscandidate}(x) = \sim \exists y \text{ worksfor}(x, y))$
—if x is a candidate he cannot be working for any company.
- S2. $\forall y \forall n (\text{haspositions}(y, n) \Rightarrow n \geq 0)$
—the number of open positions cannot be negative.
- S3. $\forall x \forall y (\text{worksfor}(x, y) \Rightarrow \exists n \text{ haspositions}(y, n))$
—a person cannot be working for a company whose number of open positions is not listed in the database (i.e. for a company that is not present in the database).
- S4. $\forall y \forall n \forall m (\text{haspositions}(y, n) \wedge \text{haspositions}(y, m) \Rightarrow n = m)$
—the number of open positions is unique.
- S5. $\forall x \forall y \forall z (\text{worksfor}(x, y) \wedge \text{worksfor}(x, z) \Rightarrow y = z)$
—a person cannot be working for more than one company.
- S7. $\forall x \forall s (\text{hassalary}(x, s) \Rightarrow s \geq \text{minimum-salary})$
—the salary of all employee must be greater than the minimum salary.
- S8. $\forall x (\exists s \text{ hassalary}(x, s) \Leftrightarrow \exists y \text{ worksfor}(x, y))$
—a person has a salary iff he works for some company.

The set T of valid transitions consists of all pairs from $S \times S$ not violating the transition constraints below. We will express transition constraints using temporal logic, as considered in Manna and Pnueli[31]. Briefly, temporal logic augments first-order logic with the modalities “o” (next) and “□” (henceforth). The wff $\text{o}P$ means that P is true in any state reachable from the current state by a single transition; the wff $\square P$ means that P is true in any state reachable from the current state by zero or more transitions. We refer the reader to[31, 11] for further discussions.

The transition constraints go as follows:

- T1. $\square \forall x \forall y (\sim \text{worksfor}(x, y) \wedge \sim \exists n (\text{iscandidate}(x) \wedge \text{haspositions}(y, n) \wedge n > 0) \Rightarrow \text{o} \sim \text{worksfor}(x, y))$

—for any state $S1$, if x is neither working for y in $S1$ nor x is a candidate and y has some open position in $S1$, then x cannot be working for y in any state $S2$ immediately reachable from $S1$.

- T2. $\square \forall x (\exists y \text{ worksfor}(x, y) \Rightarrow \square (\sim \exists z \text{ worksfor}(x, z) \Rightarrow \text{iscandidate}(x)))$
—if x has ever worked for a company he cannot disappear from the database; if he is not working for any company he regains the candidate status.
- T3. $\square \forall x \forall m \forall n \forall p \forall q (\text{haspositions}(y, m) \wedge \text{SUM}(u/\text{worksfor}(u, y)) = n \Rightarrow \square (\text{haspositions}(y, p) \wedge \text{SUM}(u/\text{worksfor}(u, y)) = q \Rightarrow m + n \leq p + q))$
—the potential personnel staff of a company is never reduced: the sum of the number of vacant positions and the number of employees cannot be reduced.

Note that constraint T3 cannot be readily expressed using first-order temporal logic. We have thus used the extension proposed in[9], which allow the use of aggregation operators, such as SUM.

2.2 The operations level

The query operations will be the predicates introduced in the previous section. Predicates can be used not only in the usual way but also as value selectors by introducing a PLANNER-like[27] notation as in the example below:

haspositions('a', ?n)

which, besides testing whether company 'a' has some number of open positions, i.e. whether

$\exists n \text{ haspositions}('a', n)$

is true, assigns to n the value found. Thus ? n corresponds to a Skolem function associated with the existentially quantified variable.

The update operations will be:

- initiate(): initializes the data base to “empty”
- apply(x): person x applies to the employment agency
- subscribe(y, n): company y subscribes to the agency, offering (initially or additionally) n positions
- hire (x, y, s): company y hires person x with salary s
- fire (x, y): company y fires person x .

We will not use a programming language to define the operations, because coding is an implementation detail that pertains to the representation level. We will rather describe the effects of each operation axiomatically with the help of dynamic logic[26, 9]. Briefly, dynamic logic extends first-order logic with new wffs of the form $[s]P$, meaning that P holds after program s terminates.

The axiom governing the behaviour of initiate() reads as follows:

O1. $[\text{initiate}()](\sim \exists x \text{iscandidate}(x) \wedge \sim \exists y \exists n \text{haspositions}(y, n) \wedge \sim \exists x \exists z \text{worksfor}(w, z) \wedge \sim \exists u \exists s \text{hassalary}(u, s))$
 —after $\text{initiate}()$ all relations are empty or, rather, there are no true instances of predicates.

O1 serves two purposes. First, at the operations level, it is the only assertion we can use to prove facts about $\text{initiate}()$. Second, any implementation of $\text{initiate}()$ given at the representation level must satisfy O1. Therefore we call O1 the *defining axiom* of $\text{initiate}()$.

The $\text{apply}(x)$ operation transforms x into a candidate, if x is not already a candidate and is not working for any company. Thus, the defining axiom of $\text{apply}(x)$ could, in principle, be:

$$\forall x (\sim \text{iscandidate}(x) \wedge \sim \exists y \text{worksfor}(x, y) \Rightarrow [\text{apply}(x)] \text{iscandidate}(x)). \quad (1)$$

Indeed, the above wff tells us that $\text{apply}(x)$ behaves correctly in so far as person x is concerned. However, it leaves open what $\text{apply}(x)$ should do when the antecedent does not hold. Moreover, it does not specify that $\text{apply}(x)$ should not interfere with the rest of the database (see the frame assumption, for example, in [45]).

For example, the following implementation of $\text{apply}(x)$ satisfies (1), but is clearly undesirable:

```
operation apply(x)
  pre-conditions not(iscandidate(x))
                  not(worksfor(x, y))
  procedure initiate()
    insert x into CANDIDATE. \quad (2)
```

To remedy this situation, we replace (1) by:

$$\forall x \forall y \forall u \forall n \forall w \forall z \forall t \forall s (([\text{apply}(x)]P) \equiv (B \Rightarrow Q) \wedge (\sim B \Rightarrow P)) \quad (3)$$

where

$$\begin{aligned} P &= \text{iscandidate}(y) \wedge \text{haspositions}(u, n) \wedge \text{worksfor}(w, z) \wedge \text{hassalary}(t, s); \\ Q &= (\text{iscandidate}(y) \vee y = x) \wedge \text{haspositions}(u, n) \wedge \text{worksfor}(w, z) \wedge \text{hassalary}(t, s); \text{ and} \\ B &= \sim \text{iscandidate}(x) \wedge \sim \exists v \text{worksfor}(x, v). \end{aligned}$$

Since all subsequent formulas are quite similar to (3), we stop here to explain how it was obtained. Suppose that we want P to hold after $\text{apply}(x)$ is executed. Then, what is the weakest precondition [15] that the initial state must satisfy? If B does not hold in the initial state, then $\text{apply}(x)$ must have no effect and, so, P must hold in the initial state. This generates the conjunct $\sim B \Rightarrow P$. If, on the other hand B holds, $\text{apply}(x)$ must add x as a candidate. So, if y is a candidate in the final state, then either y is already a candidate in the initial state or y is equal to x . This generates the first conjunct of Q . The other conjuncts of Q capture the fact that $\text{apply}(x)$ must not modify any other predicate. Therefore, the wff $[\text{apply}(x)]P$, that expresses that P must hold after $\text{apply}(x)$ is

executed, must be equivalent to $(B \Rightarrow Q) \wedge (\sim B \Rightarrow P)$. This concludes the discussion of (3).

The defining axiom for $\text{apply}(x)$, although correct, has the inconvenience of being clumsy and highly dependent on the existing predicates. Thus, we go further and replace (3) by a much more concise way of defining the effect of $\text{apply}(x)$. We introduce an axiom schema that generates (infinitely many) defining axioms for $\text{apply}(x)$.

Given a wff P , a *generalization* of P is any wff of the form $\forall x_1 \dots \forall x_n P$.

A defining axiom of $\text{apply}(x)$ is then any generalization of

$$\text{O2. } [\text{apply}(x)]P \equiv (B \Rightarrow Q) \wedge (\sim B \Rightarrow P)$$

where B is $\sim \text{iscandidate}(x) \wedge \sim \exists y \text{worksfor}(x, y)$;
 Q is $P[A/\text{iscandidate}(y)]$;
 and A is $\text{iscandidate}(y) \vee x = y$.

That is, Q is a wff obtained by replacing each subformula of P of the form $\text{iscandidate}(y)$ by $(\text{iscandidate}(y) \vee x = y)$. Before the replacement, P should be replaced by an alphabetical variant [43] where x does not occur bounded. This proviso avoids that x becomes inadvertently bounded when $\text{iscandidate}(y)$ is replaced by $(\text{iscandidate}(y) \vee x = y)$. We stress here that O2 defines a family of formulas. In fact, P is any wff whatsoever and Q is obtained from P as indicated.

Using O2 we can prove, for example, that $\text{apply}(x)$ does not interfere with $\text{haspositions}(y, n)$. Indeed, O2 directly implies the following wff:

$$\forall x \forall y \forall n (\text{haspositions}(y, n) \equiv [\text{apply}(x)]\text{haspositions}(y, n)). \quad (4)$$

This follows by taking P as $\text{haspositions}(y, n)$ in O2 and generalizing the resulting formula. In fact, we can go further and directly derive (3) from O2. The converse also holds, but it is not so easy to obtain.

Likewise, a defining axiom of each operation is any generalization of the corresponding scheme:

$$\text{O3. } [\text{subscribe}(y, m)]P \equiv (B \Rightarrow Q) \wedge (\sim B \Rightarrow P)$$

where B is $m > 0$;

Q is $P[A/\text{haspositions}(z, k)]$;

and A is $(z \neq y \Rightarrow \text{haspositions}(z, k)) \wedge (z = y \Rightarrow ((\sim \exists n \text{haspositions}(y, n) \Rightarrow k = m) \wedge (\exists n \text{haspositions}(y, n) \Rightarrow k = n + m)))$.

—company y offers m new positions if it has not yet offered any, or y offers m new positions in addition to those it has already offered.

$$\text{O4. } [\text{hire}(x, y, s)]P \equiv (B \Rightarrow Q) \wedge (\sim B \Rightarrow P)$$

where B is $\text{iscandidate}(x) \wedge \exists n (\text{haspositions}(y, n) \wedge n > 0) \wedge s \geq \text{minimum-salary}$;

Q is $P[A_1/\text{iscandidate}(y), A_2/\text{haspositions}(z, n), A_3/\text{worksfor}(z, u)]$;

A_1 is $\text{iscandidate}(y) \wedge y \neq x$;
 A_2 is $(z \neq y \Rightarrow \text{haspositions}(z, n)) \wedge (z = y \Rightarrow \text{haspositions}(z, n - 1))$; and
 A_3 is $\text{worksfor}(z, u) \vee (z = x \wedge u = y)$.

—if x is a candidate and y has open position then, if the operation is executed, x ceases to be a candidate, the number of openings of y is decremented by one and x works for y .

O5. $[\text{fire}(x, y)]P \equiv (B \Rightarrow Q) \wedge (\sim B \Rightarrow P)$

where B is $\text{worksfor}(x, y)$;

Q is $P[A_1/\text{iscandidate}(z), A_2/\text{haspositions}(u, n), A_3/\text{worksfor}(t, v), A_4/\text{hassalary}(r, s)]$;

A_1 is $\text{iscandidate}(z) \vee z = x$;

A_2 is $(u \neq y \Rightarrow \text{haspositions}(u, m)) \wedge (u = y \Rightarrow \text{haspositions}(u, m + 1))$;

A_3 is $\text{worksfor}(t, v) \wedge \sim(t = x \wedge v = y)$;

and A_4 is $\text{hassalary}(r, s) \wedge r \neq x$.

—if x works for y , which happens to have n free positions open, then after execution of the operation, x becomes again a candidate, the number of openings of y is incremented and x ceases to work for y .

Is it true that, if the data base is handled exclusively by these operations, a person will never be working simultaneously for two companies (static constraint 5)? This can be shown (and in fact could be rigorously proven by an inductive reasoning) by examining the behaviour imposed by the axioms: 'a' can only be hired by 'b' if, among other conditions, he is a candidate; he can only become a candidate if he applies (but for that he must not be working for any company) or is fired from the company where he currently works; finally, on being hired by 'b' he ceases to be a candidate. Similarly, the reader can check that the potential staff of each company is non-decreasing (transition constraint 3), from the fact that there is no inverse operation to subscribe and that hire exchanges one vacant position for one occupied position and fire does the exact opposite.

However it should be clear even from this simple example that expressing constraints only via the intended behaviour of the operations provides a very obscure way of documenting them (hence the need for the explicit declarations at the states/transitions level). On the other hand, operations suggest an effective way of enforcing constraints.

2.3 The representation level

2.3.1 *The data model.* For the application modelling we present a semantic database model, called THM (Temporal-Hierarchic Model)[40, 41], which is a Binary Relationship Model in the ISO terminology[28] and was influenced by several approaches in this area[1, 3, 4, 5, 18, 32, 39]. It uses the hierarchy concepts of aggregation and generalization due to Smith and Smith[44] and the concept of grouping[25] (also called correspondence[39] or association[6]). The idea of role, used in different

senses by Bachmann[2] and Falkenberg[18], will be used to give a significant characterization of the generalization/specialization.

In order to distinguish the real world (composed of the concrete world and the abstract world) from the (data-) model world we introduce the following terminological correspondences:

Real World		Model World
informal	formal	
object, thing	constant	entity
type	sort	class
attribute, property	predicate	relationship
relationship	predicate	relationship
subtype	unary-predicate	class, subclass

The *objects* of the real world (universe of discourse) correspond to entities (in the universe of discourse description[28]). The properties of an object are given as *relationships* to other objects (also called attributes in some models). Figure 1 gives as an example the entity "Universität Stuttgart" with relationships "haspositions" and "worksfor".

Entities of the same type, i.e. with the same relationships, occur as members of *classes*. The relationships are then extended to relations between classes. In spite of the fact that binary relations can always be seen in two directions we can give a unique direction to the arcs in order to identify the relations of interest. In this sense we speak about a relation from A to B .

For each relation there exist two associated *cardinalities* in the sense of Abrial[1]. It means that if we have a relation r

$$A \xrightarrow[r, s]{r(n, m)} B$$

then for each member 'a' of A there exist at least n and at most m members of B related to 'a'. Conversely, for each member 'b' of B there exist at least r and at most s of A related to 'b'.

In the same sense as the Semantic Data Model SDM of Hammer and McLeod[25] distinguishes member and class attributes were identify member and class relationships. Member relationships are relationships between entities, and class relationships are relationships between one class and one entity, i.e. there are unique values associated with the whole class, such as mean salary, number of members, etc. One ever present class relationship exists between a class and its name.

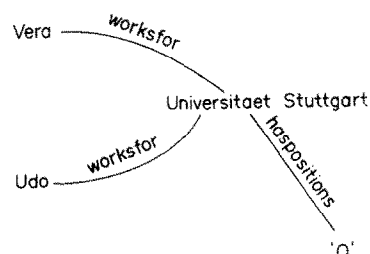


Fig. 1. Entities and relationships.

There are certain basic classes whose members never change and in general can be characterized by range limits. Such classes are called *domain classes* (similar to the value classes of SDM).

Class names are presented in capital letters and relations in lower case letters.

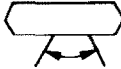
2.3.2 *Modelling*. Modelling involves interpretation of the predicates given in the states/transitions level. The predicates are stated in many-sorted logic and the sorts identify the classes of the model.

Note that the predicates completely specify all sorts, because the operations at the operations level only use existing sorts.

“iscandidate” as a unary predicate, leads to a class CANDIDATE, “haspositions” and “worksfor” lead to relationships between classes. The second argument of “haspositions” determines a typical domain class which will be called FREE-POSITIONS and has a range “[0..] of integer” (the entities are integers with lower bound 0 and no upper bound). Static constraint S8 guarantees that the sorts of the first argument of “worksfor” and of “hassalary” are the same, giving the class EMPLOYEE. The constant “minimum-salary” gives a class-relationship from EMPLOYEE to SALARY. Fig. 2 shows the corresponding conceptual schema.

All static constraints (S1 to S8 of Section 2.1) are implicitly contained in Fig. 2, as follows

S1: the notation



means that the classes are disjoint;

S2: is guaranteed by the domain of FREE-POS;

S3: is given by the first argument of the cardinality (1, 1) of “haspositions”;

S4: is specified by the second argument of the cardinality of “haspositions”;

S5: is given by the second argument of the cardinality (1, 1) of “worksfor”;

S6: is specified by the cardinality of “minim.-salary”;

S7: the domain of the class SALARIES (ms. . .);

S8: the first argument of the cardinalities of “hassalary” and “worksfor”.

As an illustration of the THM/DDL we show the description of the class EMPLOYEE.

class EMPLOYEE:

class-relationship

minimum-salary: SALARY(1, 1)

member-relationships

hassalary: SALARY(1, 1)

worksfor: COMPANY(1, 1)

Operations are specified in the THM/DML, whose syntax will be introduced here only through the examples. If the *pre-conditions* are fulfilled the statements in the *procedure* are executed, which automatically must guarantee the post-conditions of the operation axioms. The basic operations are *insert* and *delete* of entities and *establish* and *remove* of relationships. A sequence of delete and insert for the same entity can also be specified with a *move* state-

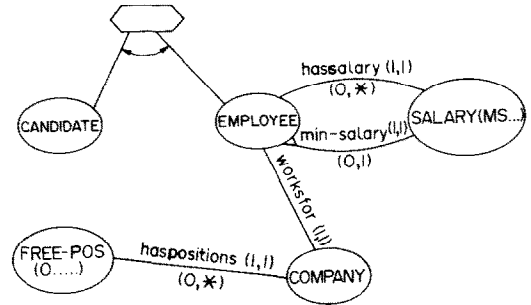


Fig. 2. The conceptual schema.

ment. For a predicate $p(x, y)$ and an element ‘a’ of the first sort we denote with $p(a)$ the element b (if p is functional) or the set of elements b_i of the second sort such that $p(a, b)$ or $p(a, b_i)$ is true. This element b or set b_i can be given a name n by using

let n be p(a).

The operation axioms O1–O7 are implemented as follows:

operation *apply*(x):

pre-conditions

not(x in CANDIDATE)

not(x in EMPLOYEE)

procedure

insert x into CANDIDATE

operation *subscribe*(y, m):

pre-conditions

$m > 0$

procedure

if not(y in COMPANY)

then insert y into COMPANY

establish y haspositions m

else let n be haspositions(y)

establish y haspositions n + m

remove y haspositions n.

The statement “*remove y haspositions n*” is not obligatory because as a consequence of the cardinality (1, 1) of the relation “haspositions” it will be executed automatically.

operation *hire*(x, y, s):

pre-conditions

x in CANDIDATE

haspositions(y) > 0

$s \geq$ minimum-salary()

procedure

move x from CANDIDATE to EMPLOYEE

let n be haspositions(y)

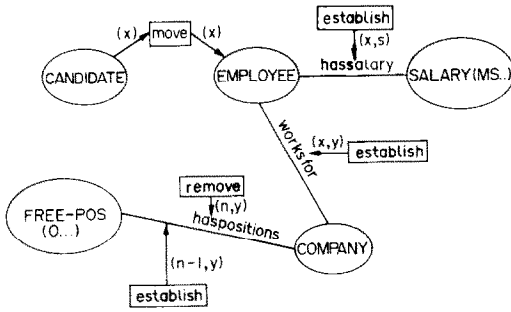
establish x worksfor y

x hassalary s

y haspositions $n - 1$.

The exact meaning of the pre-condition haspositions(y) > 0 is

*if y in COMPANY then haspositions(y) > 0
else false.*

Fig. 3. Hire(x, y, s).

Note that after the *move* statement it is clear that the x from *establish* is an EMPLOYEE. Another observation is that, as a consequence of the fact that SALARY and FREE-POSITIONS are domain classes, the entities s and $n - 1$ need not be inserted.

operation *fire*(x, y):

pre-conditions

$$y = \text{worksfor}(x)$$

procedure

move x from EMPLOYEE to CANDIDATE

let n be *haspositions*(y)

establish y *haspositions* $n + 1$.

The *delete* x from EMPLOYEE (implicit in the *move* statement) automatically removes all relationships between x and other entities.

All illustration of what happens in the schema when the hire operation is executed is given in Fig. 3. The dashed lines are the preconditions.

3. SOME TEMPORAL ASPECTS

Staying with the agency data base example, we shall examine what additional features are required at each level to cope with three slightly different versions of transition constraints referring to restrictions imposed on a person who has once worked for a certain company and then ceased to do so.

3.1 Keeping historical aspects

3.1.1 *States/transitions level*. Consider the transition constraint:

$$T4: \quad \square \forall x \forall y (\text{worksfor}(x, y) \Rightarrow \text{o}(\sim \text{worksfor}(x, y) \Rightarrow \square \forall z (\text{worksfor}(x, z) \Rightarrow y \neq z)))$$

—if x ceases to work for y then he can never work for y again.

As the states/transitions level, there is therefore no special difficulty.

3.1.2 *Operations level*. At the operations level, extra machinery must be added, because historical (superseeded) information is not assumed to be kept (which is why such constraints are not allowed in [28]). A simple solution is to introduce the auxiliary predicate

hasworkedfor(x, y)— x has worked for y at some time (which time in particular, does not matter).

We also have to introduce new defining axioms indicating how the operations behave with respect to *hasworkedfor*(x, y). Due to the use of schemas, as explained in Section 2.2, we only have to add new axioms for *initiate*() and *fire*(x, y), and modify those of *hire*(x, y, s). Thus, in addition to O1, we have:

$$O1': [\text{initiate}()] \sim \exists u \sim \exists v \text{hasworkedfor}(u, v).$$

A defining axiom for *hire*(x, y) is now any generalization of (note that O4' replaces O4):

$$O4': [\text{hire}(x, y, s)]p \equiv (B' \Rightarrow Q) \wedge (\sim B' \Rightarrow P)$$

where B' is *iscandidate*(x) $\wedge \exists n$ (*haspositions*(y, n) $\wedge n > 0$) $\wedge s \geq$ *minimum-salary* $\wedge \sim$ *hasworkedfor*(x, y); and Q is as for O4.

A defining axiom for *fire*(x, y) is now also a generalization of:

$$O5': [\text{fire}(x, y)]P \equiv (B \Rightarrow Q') \wedge (\sim B \Rightarrow P)$$

where B is as for O5; Q' is $P[A/\text{hasworkedfor}(w, u)]$; and A is *hasworkedfor*(w, u) $\wedge (w = x \wedge u = y)$. O5' does not replace O5, but complements it, since the *fire*(x, y) operation remains basically the same.

3.1.3 *Representation level*. At the representation level the past and future of a given class is specified by the so-called “pre-post relations”. If entities deleted from a class A are subsequently inserted in a class B , A is called the pre-class of B and B the post-class of A . This relationship is denoted as $A > \dots > B$. If all entities from A must go to B we write $A > > \dots > B$ and if all entities of B come from A we write $A > \dots > > B$. Pre-post relations are class-to-class relationships. For the example we have the following pre-post relationships

$$\text{EMPLOYEE} > > \dots > \text{CANDIDATE} > > \dots > > \text{EMPLOYEE} > > \dots > > \text{EX-EMPLOYEE}$$

and the new class EX-EMPLOYEE is defined as

```
class EX-EMPLOYEE
class-relationships
  pre-class: EMPLOYEE exclusive
  member-relationships
    ex.worksfor: COMPANY
```

With the terminology “ex.worksfor” the worksfor relationship of EMPLOYEE is automatically inherited by the post-class. Other relationships of EMPLOYEE are lost. Since a fired employee receives also the candidate status, CANDIDATE is pre- and post-class of EMPLOYEE. To the definition of CANDIDATE the class-relationships

```
post-class: EMPLOYEE exclusive
pre-class: EMPLOYEE
```

are added and for EMPLOYEE we have:

```
class-relationships
  pre-class: CANDIDATE exclusive
  post-class: CANDIDATE exclusive.
  post-class: EX-EMPLOYEE exclusive.
```

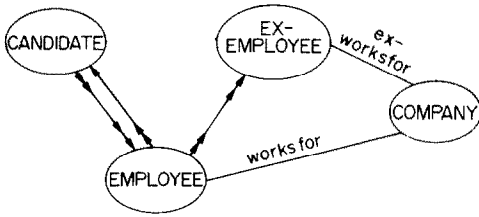


Fig. 4. Pre-post relations.

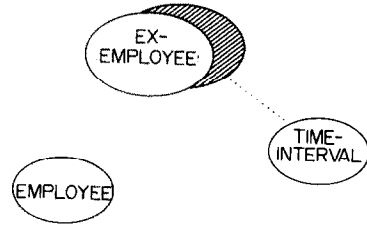


Fig. 5. Explicit time.

The resulting schema is shown in Fig. 4. Such as the pre-post relations retain “old” entities in the database another feature of THM, not explained in this paper, allows to retain “old” relationships.

For the DML the changes are minimal. To the pre-conditions of *hire* we must add the condition

not(x in EX-EMPLOYEE) or not(ex-worksfor(x, y))

The *fire* operation remains the same, because the

move x from EMPLOYEE to CANDIDATE

automatically inserts *x* into EX-EMPLOYEE and maintains the connection to the company *y*, but now via the *ex-worksfor* relation. In fact, instead of the move we can even state only “*delete x from EMPLOYEE*”.

3.2 Explicit reference to time

3.2.1 *States/transitions level.* The notion of transition implies for the concepts introduced in Section 3.1 an implicit reference to time: the *immediate future*, encompassing all states reachable by one transition, and the *general future* incorporating states reachable by any number of transitions.

Certain constraints however may require an explicit reference to time, in terms of points in time (dates) or time intervals. An example is the constraint T4' below, expressed at the states/transitions level with an added notation “*T*”, which refers intuitively to some clock device which is increased linearly. Transitions follow the axiom

$$T = t \Rightarrow \square T > t$$

—the value of the clock always increases as transitions occur.

The exact amount of the time increments is left unspecified.

T4': $\forall x \forall y \forall t \square (\text{worksfor}(x, y) \Rightarrow \circ (\sim \text{worksfor}(x, y) \wedge T = t \Rightarrow \square (\text{worksfor}(x, y) \Rightarrow T - t \geq 30)))$

—if *x* ceases to work for *y* then he can only work again for *y* after 30 units of time.

3.2.2 *Operations level.* At the operations level the clock mechanism is also required and, in addition, the auxiliary predicate *hasworkedfor* must have a time parameter, recording when the person was fired. Axiom O1' is replaced by

O1': $[\text{initiate}()] \sim \exists u \exists v \exists t \text{hasworkedfor}(u, v, t)$.

Axiom scheme O4' remains the same, except that *B'* is now $\text{iscandidate}(x) \wedge \exists n (\text{haspositions}(y, n) \wedge n > 0) \wedge s \geq \text{minimum-salary} \wedge \forall t (\text{hasworkedfor}(x, y, t) \Rightarrow T - t \geq 30)$.

Likewise axiom scheme O5' remains the same, except that now *Q'* is $P[A/\text{hasworkedfor}(w, u, t)]$; and *A* is $\text{hasworkedfor}(w, u, t) \vee (w = x \wedge u = y \wedge t = T)$.

3.2.3 *The representation level.* Modelling this case requires the addition of a time parameter to EX-EMPLOYEE. But now, for the ex-employees arises a special problem. The same entity ‘*e*’ can occur several times as member of EX-EMPLOYEE, related each time to a different time-slice. For this reason the ex-class needs a special notation indicating the time coordinate (Fig. 5), and thereby allowing the representation of “multiple” occurrences of the same entity at different time intervals.

The time coordinate is specified adding the parameter

“with time”

to the definition of the class. This means that a class with time is an aggregation of a common class with a special class of time intervals, indicating the period during which the entity was a member of the class. The time interval class is composed of pairs of time points called “from” and “to”, indicating the limits of the time interval. These points are vectors or tuples (t_1, \dots, t_n) of a calendar system, each component is of one time unit, for example (1982, 7, 10) or (10h, 5m, 30s) are typical time points. A special function *clock* indicates the time point “now” and *clock.unit* is unit component of the now vector (e.g. *clock.day*). By an insert the clock time is assigned to the from point and the to point remains variable. A delete does not really delete the entity from the class but only assigns the actual clock time to the to point.

To the operations the pre-conditions

not(x in EX-EMPLOYEE at t)
t in <clock.day-30, clock.day>

must be added to *hire* and the explicit description of *fire* remains the same. The statement

move x from EMPLOYEE to CANDIDATE

in the *procedure* of *fire* has now as implicit consequence the actions

insert x in EX-EMPLOYEE at clock
establish x ex-worksfor y at clock.

3.3 Active systems

The most frequent situation in data bases is that transition constraints only determine which transitions cannot occur. Those which can occur are on the other hand not mandatory; the data base can simply remain at a certain state indefinitely. Similarly, as we noted, the assumption underlying the dynamic logic notation at the operations level was that operations *might* be executed (conceivably through the initiative of some user) if the pre-conditions held.

We shall now look at the case where transitions *must* occur and, correspondingly, the “system” will take the initiative to invoke certain operations.

3.3.1 *States/transitions level.* At the states/transitions level no principal change is necessary as compared to Section 3.2.1. The explicit reference to time or time interval will suffice to specify the proper behaviour through the constraints. An example is

T4". $\forall x \forall y \forall t \square (\sim \text{worksfor}(x, y) \Rightarrow \text{o}(\text{worksfor}(x, y) \wedge T = t \Rightarrow \square (T \geq t + 100 \Rightarrow \sim \text{worksfor}(x, y))))$
—jobs now are temporary; a person cannot stay with a company for more than 100 units of time.

3.3.2 *Operations level.* At the operations level it becomes convenient to redefine the predicate *worksfor*, adding a time parameter indicating the date when the person was hired. The reader will appreciate that we add time parameters and modify the basic predicates only when indispensable. It is in general not feasible to have time parameters for all predicates in real data bases.

A more fundamental change is that *fire*(*x*, *y*) must be invoked whenever *C* holds, where *C* is “*worksfor*(*x*, *y*, *t*) $\wedge T \geq t + 100$ ”.

This fact cannot be expressed in dynamic logic however. In order to capture it, we have to resort to temporal logic again and introduce a new predicate, *at-fire*(*x*, *y*), indicating that *fire*(*x*, *y*) will begin execution. A similar approach is taken in [31] to specify properties of parallel programs, such as liveness.

Thus, we specify that *fire*(*x*, *y*) must be invoked whenever *C* holds as follows:

O5". $\square \forall x \forall y \forall t (\text{worksfor}(x, y, t) \wedge T = t + 100 \Rightarrow \text{at-fire}(x, y))$.

3.3.3 *The representation level.* At the representation level a *monitor* must be available, with the capabilities of:

- (a) detecting the occurrence of events
- (b) triggering all the corresponding operations for the forced action axioms, e.g. O5" above.

THM/DML allows the definition of events using a clock and triggers. *EMPLOYEE* must have the time relationship as in Section 3.2. and the specification now becomes

event *timeout*(*x*, *y*, *t*)
on *change*(*clock.day*)
condition
y = worksfor(*x*) *at* *t*
clock.day $\geq t + 100$

trigger *firing*(*x*, *y*, *t*)
pre-conditions
timeout(*x*, *y*, *t*)
procedure
fire(*x*, *y*).

4. REMARKS ON MODULATION

Due to the size and complexity of actual data base specifications, it is unrealistic to expect that a conceptual schema can be specified as a single construct. A modular strategy is in order, ideally allowing that properties proven in a module do not have to be proven again when passing to a higher-order module.

This section will be rather informal, mostly relying on examples, a rigorous treatment still requiring further research. We shall draw from the published literature, especially with respect to abstractions [44, 39], construction of theories from other theories [22, 8] and the application of the latter idea to data bases [17, 46]. The notions of frames [33, 38] taken from artificial intelligence, and cohesion and coupling [47] from software engineering provide some guidelines for determining the scope of modules.

A module is a closed unit within a data base schema, with fixed sets of sorts, predicates, constraints and operations. In the algebraic sense it is an abstract data type and in the data base framework it constitutes a sub-schema. The encapsulation strategy, inherent to abstract data types, requires that only the operations defined inside a module can directly manipulate the objects of the module. This approach facilitates the design process, since the designer can concentrate on a smaller universe and, as a consequence, also simplifies integrity checking.

The specification of higher-order modules from lower-order modules is done roughly along three steps:

- (a) constructing
- (b) expanding
- (c) hiding.

To *construct* means to indicate how a module is formed from lower-order ones. Distinct modules can be constructed using the same lower-order module, which then constitutes a *shared module* [8]. Four specific constructors will be introduced. The first constructor is based on the idea of frame and the others correspond to abstractions:

(i) *Combining.* The modularization process begins with *starting modules*, whose granularity can be decided in each case with respect to pragmatic considerations. In our example, the entire employment agency data base was specified as a single module. Modules must be small enough to be understandable and large enough to be designed as self-contained.

Combining is known as constructor in the area of abstract data types[8].

(ii) *Specialization*. The new module is formed as a special case of an existing module. More precisely, specialization involves the *restriction of a given sort s* , appearing in the original module, and consequently the restriction of the predicates and operations where elements of s occur as parameters. Because the restriction defining the new module is expressible in the same notation as constraints, the process is comparable to forming a new type as a quotient[22] (by adding equations to the given type). The inverse process is called generalization[44] in the data base literature.

(iii) *Grouping*. The new module is formed by an internal grouping process. The constructor involves the *replacement of a given sort s by the powerset of s* , which leads to the redefinition of predicates and operations where elements of sort s occur as parameters. Often, the sets are introduced by the application of some partitioning criterion. In the terminology of [22], adding a sort is called an extension, and the formation of sets of elements of some sort is characterized as parameterization. Grouping has been proposed as an abstraction called correspondence in [39]; see also the concept of association [6].

(iv) *Aggregation*. The new module is formed by putting together two or more existing modules. The process can be viewed as a *Cartesian product of modules* (as will become clear in Section 4.1). This is analogous to tupling in [22] and to the TUP constructor in [17]. Aggregation also appears as an abstraction in the data base literature [44].

In most cases we construct a new module in order to be able to describe properties that were not present and could not even be expressed in the original constituent modules. Once a new module is constructed, we can *expand* it by adding new predicates and new operations (an enrichment, in the terminology of [22]), and possibly additional constraints and sorts (which is also regarded as enrichment, in the broader sense used in [8]).

On the other hand, the general rule is to consider that whatever property is present in the original modules to be also present in the new modules and to be equally visible and accessible. In the cases where this is not wanted, we must explicitly *hide* the property (see the derive declaration in [8], the hidden operations in [24] and the notion of information hiding of Parnas [36]).

At this point, we should remark that the definition of external schemas does not necessarily coincide with the division of the conceptual schema into modules discussed here. The guiding principle for the design of external schemas is user authorization, which often contradicts self-containment; each user is frequently affected by operations that can only be performed by other users, or his and their operations are needed together to perform some change in the data base.

We shall now examine modularization at each of the three levels introduced earlier in this paper. The discussion will be based on the employment agency example, which will be regarded as a starting module—*Personnel*. We now assume that the agency, besides serving its client companies by procuring employees for them, can also obtain a machine (at most one per company) on their behalf. Accordingly, another starting module—*Equipment*—is introduced.

In *Equipment*, two predicates are specified: $uses(y)$ and $owns(y)$, meaning, respectively, that company y is using and is the proprietor of a machine. There is one static constraint declaring that being a proprietor implies being using, and a transition constraint declaring that ownership is irrevocable. The operations are $lease(y)$, $return(y)$ and $buy(y)$; lease and buy have the effect that the company starts using the machine, but only buy results in ownership. Return means that the machine is given back to the vendor, which is not allowed if the machine has been bought.

4.1 Modularization at the states/transitions level

Consider a state sp of the *Personnel* module denoted by the conjunction of ground positive predicates below:

$$sp = \text{haspositions}(c1, 3) \wedge \text{haspositions}(c2, 1) \wedge \\ \text{haspositions}(c3, 2) \wedge \text{worksfor}(e1, c1) \wedge \\ \text{worksfor}(e2, c1) \wedge \dots \wedge \text{worksfor}(e10, c1) \wedge \\ \text{worksfor}(e11, c2).$$

Thus, in sp there are 3 companies, where $c1$ has 10 employees, $c2$ has 1 and $c3$ has no employees.

Now, suppose that we want to characterize certain states and transitions in terms of the entire *staff* (a collective name for employees) of each company, which cannot be done in the *Personnel* module where only individual employees are considered. We can construct a new module *Personnel-by-Companies* by using the grouping constructor; in this case the partitioning criterion is to group the employees by company. The predicate $worksfor$ provides then the basis for the predicate set $worksfor(x, y)$ (see the Setof predicate of Kowalski [29]).

In *Personnel-by-Companies* we would have a state spc , obtained from state sp , denoted as follows:

$$spc = \text{haspositions}(c1, 3) \wedge \text{haspositions}(c2, 1) \wedge \\ \text{haspositions}(c3, 2) \wedge \text{set.worksfor}(\{e1, e2, \\ \dots, e10\}, c1) \wedge \text{set.worksfor}(\{e11\}, c2) \wedge \text{set.} \\ \text{worksfor}(\{ \}, c3).$$

Once this module is constructed, it is useful to expand it, for example, by adding a new predicate $\#staff$ (i.e. cardinality of staff), which can be defined recursively as in [29].

The transitions to be considered are those involving a uniform treatment of all elements in each staff (for individual transformations the *Personnel* module suffices). Examples are a transition of spc to a state where $c1$ has an empty staff, or to a state

where $c1$ has as staff the union of its former staff with that of $c2$ and $c2$ has an empty staff.

Suppose, to continue, that companies with a staff of 10 employees or more constitute a special case. We can construct a module *Personnel-by-Large-Companies*, as a specialization of *Personnel-by-Companies*; the defining restriction being that the cardinality of staffs must be greater or equal to 10. The states of the new module are denoted by sub-expressions of the original ones. From *spc* we obtain:

$$splc = \text{haspositions}(c1, 3) \wedge \text{set.worksfor}(\{e1, e2, \dots, e10\}, c1).$$

As before, we might expand the constructed module with predicates and constraints. A possible (static) constraint would be an upper bound of 20 on the size of a staff.

A transition between two states, say *splc1* and *splc2*, is valid only if in the original module transitions between states denoted by expressions having those of *splc1* and *splc2* as sub-expressions are valid.

We must note that transitions violating the defining restriction are not necessarily disallowed; however, they appear in the module as causing the information involved to vanish in the new state; so, if $c1$ loses its staff, the resulting state will *not* be denoted by $\text{haspositions}(c1, 13) \wedge \text{set.worksfor}(\{\}, c1)$, but by an empty expression (recalling there where no other companies in *splc*).

Also, only constraints involving information present in the states of the module can be formulated. For instance, in *Personnel-by-Large-Companies* the transition constraint that the size of the staff of a large company cannot increase while there exist companies with empty staffs is meaningless, since the latter companies are excluded from the module (the constraint can, however, be expressed in the *Personnel-by-Companies* module).

Suppose, finally, that we want to include the following transition constraint:

—a machine can only be owned by a company which had a staff of ten people or more at the moment when ownership was established.

Obviously this constraint cannot be expressed in the *Equipment* module, but makes perfect sense in a module encompassing information about machines and also about staffs. This is accomplished by using aggregation. Note that the order of application of the constructors is significant; the module that we want—*Procurement-by-Large-Companies*—should be obtained by first aggregating *Personnel-by-Companies* with *Equipment*, and then specializing to companies with staffs of ten employees or more. If we had aggregated *Personnel-by-Large-Companies* with *Equipment* we might lose equipment information about smaller companies. A state *sprlc*, obtained from *spc* and some state *se* of *Equipment*, might well be:

$$sprlc = \text{haspositions}(c1, 3) \wedge \text{set.worksfor}(\{e1, e2, \dots, e10\}, c1) \wedge \text{uses}(c1).$$

The valid states of modules obtained by aggregation must belong to the Cartesian product of valid states from the constituent modules, and the valid transitions must be such that each pair of state components constitute a valid transition in the respective constituent modules.

We are now in a position to formulate the intended transition constraint in the *Procurement-by-Large-Companies* module:

$$\Box \forall y (\sim \text{owns}(y) \wedge \sim \exists z \text{ set.worksfor}(z, y) \Rightarrow \sim \text{owns}(y))$$

—if y does not already own a machine nor is present as an employer in the module, then in the immediate future it cannot own a machine.

To this expression, the module restriction is implicitly added, resulting in:

$$\Box \forall y (\sim \text{owns}(y) \wedge \sim \exists k (\# \text{staff}(y, k) \wedge k \geq 10) \Rightarrow \sim \text{owns}(y)).$$

4.2 Modularization at the operations level

At the operations level, each module is an abstract data type and modularization creates a partial order of types. Most of the formal work on modularization has in fact been done at this level within the algebraic approach[8].

In order to guarantee that constraints pertaining to lower-order modules be enforced, we shall adopt the strategy that any new operations introduced by expanding constructed modules will be defined in terms of the operations of the constituent modules.

In our example, we can introduce an operation $\text{buy}'(y)$ in the *Procurement-by-Large-Companies* module, whose action would consist of testing whether company y is present as an employer and, if this is the case, calling the $\text{buy}(y)$ operation defined for the *Equipment* module.

Generally speaking, modules formed by specialization require *conditional* clauses, such as the test above, whereas aggregation requires *composition* and grouping requires *iteration* clauses[6]. In our example, buy only alters information in the *Equipment* module, needing access to staff information just for checking a condition; as in[46], the reader may easily formulate examples for compositions of updates on the two constituent modules.

In the introductory part of Section 4 we mentioned hiding as a third step in the design of modules. Hiding is particularly relevant at the operations level. Normally operations defined in different modules remain accessible everywhere. However, in order to enforce constraints added at some higher-order module, we may need to disallow a direct call to a constituent module operation. In our example the lease and return operations of the *Equipment* module can be directly invoked (noting that simply using a machine, as opposed to owning it, is not further constrained outside the *Equipment* module), but buy should only be accessible through the call performed within the buy' operation. Thus we say that buy is hidden.

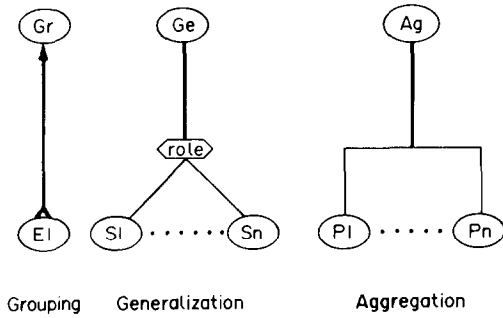


Fig. 6. The three abstractions.

Other examples of operations at higher-order modules would be a staff dismissal operation (iterative application of the fire operation, perhaps prior to the extinction of a company) or the transfer operation of the staff of a company to another company (in case one company is merged with another), both operations being appropriate at the Personnel-by-Companies module.

4.3 Modularization at the representation level

A class, i.e. the representation of a sort in our data model, is defined as an *elementary module*. All such modules have exactly the four operations insert, delete, establish and remove. Continuing with the modular design means combining some elementary modules in a meaningful way or applying one of the three abstractions to some elementary modules inside the module in question.

The schematic illustration of the abstractions is given in Fig. 6, whose meaning will be explained in the following.

The definition of a non-elementary module involves a schematic description (submodules, abstractions) and the specification of operations. These can be new operations, re-definitions of operations of submodules and combinations of lower-level operations. In order to guarantee all post-conditions, hidden operations (identified by an asterisk in the module representation) cannot be used directly as

some constraints at the “higher-level” version of those operations could then be violated.

The *combining* strategy is used for obtaining the two starting modules PERSONNEL and EQUIPMENT (Fig. 7). Note that the class COMPANY is shared by both and appears in each module only with the relations of interest.

A *grouping* is obtained using some criterion to form a partition of the set of members of a given class, and these subsets then are the members of the new class. For the PERSONNEL-BY-COMPANIES module we use the inverse of the worksfor relation (which is of cardinality $(0, n)$) and obtain a grouping of the employees into staffs of companies. Worksfor is transformed in a set.worksfor relation between the new class STAFF and COMPANY. Note that the cardinality of the inverse of set.worksfor now is $(0, 1)$. A relation #staff, from the new class to a domain class CARDINALITY can be added. The resulting module, illustrating only the classes of interest, is shown in Fig. 8.

For the *specialization* a criterion called *role* can be applied to a given class in order to obtain one or more subclasses. It is also possible to apply more than one role to the same class. The splitting into subclasses via a role can be determined by a predicate, by an index set or explicitly by the user[40]. In the PERSONNEL-BY-LARGE-COMPANIES module the predicate #staff > 10, applied to COMPANY via the set.worksfor relation, determines one new class called LARGE-COMPANIES (Fig. 9).

Finally in order to obtain the PROCUREMENT-BY-LARGE-COMPANIES module (Fig. 10) we must execute two steps. First an *aggregation* of STAFF, COMPANY and MACHINE, using the relations “set.worksfor” and “uses” to obtain a class PROCUREMENT. This class is now specialized to PROCUREMENT-BY-LARGE-COMPANIES using the same role #staff as before. This is possible because COMPANY is one of the components of PROCUREMENT. In general the aggregation may not always be done via preexisting relations; the criteria can also be given explicitly by the user.

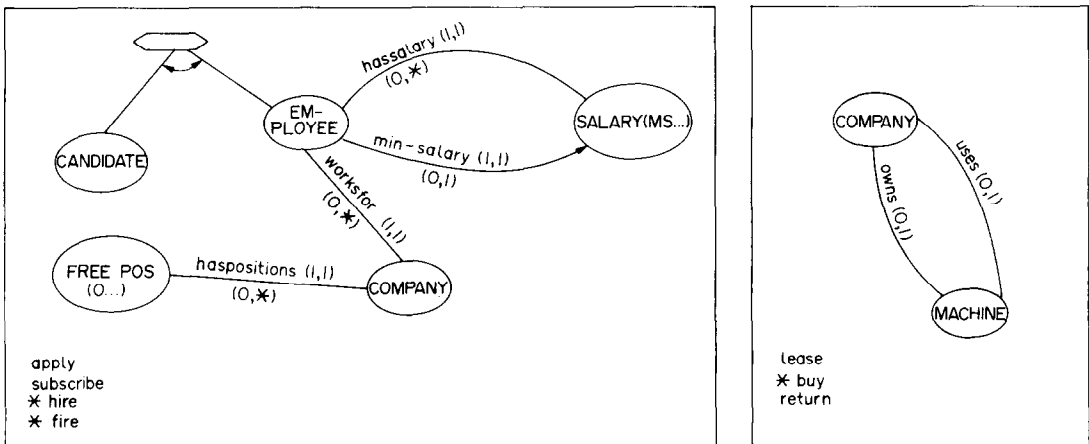


Fig. 7. Starting modules.

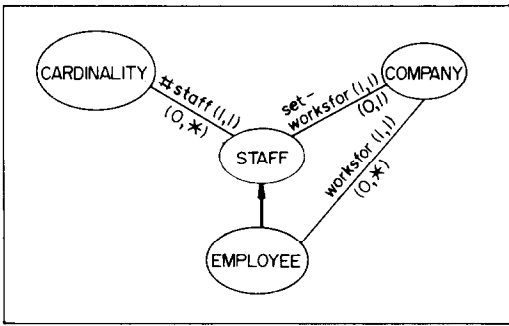


Fig. 8. Grouping.

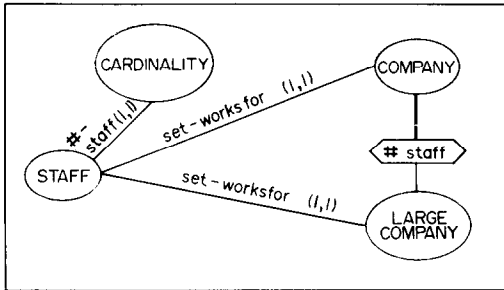


Fig. 9. Specialization.

As a final illustration of the example we show the modular organization in Fig. 11 and the complete Conceptual Schema (without modules), including the temporal aspects of Section 3, in Fig. 12.

5. CONCLUSIONS

The size and complexity of conceptual schemas pose considerable difficulties for their specification, leading us to divide the specification task along two orthogonal dimensions: levels and modules.

The resulting proposed methodology tries to identify the different aspects involved and to treat them in a natural sequence. Both the division into levels

and into modules aims at stability, in the sense that a change localized in a level or module should affect as little as possible the remaining components; also, the change itself should become easier to integrate. An especially important situation arises when dropping an operation that was, with its pre- and post-conditions, part of the strategy to enforce some integrity constraint. Because such a constraint must have been explicitly declared at the states/transitions level, we have some guidance for

- detecting the fact that there is a constraint no longer effectively enforced by the remaining operations;
- adjusting adequately the pre- and post-conditions of the remaining operations.

We have been using formal notations. In fact, precision is the really fundamental requirement but, if in addition, formal notations are used, we gain the ability to prove in a rigorous way properties of our specification, and automated tools (in the style of [20], for example) for helping in the design and maintenance of the specification become feasible.

The problems created by the use of formalisms are of course the problems of communication between the prospective users (and application area experts) and the especially trained designers. The communication therefore should not use formalisms, they should rather be done in natural language and via diagrammatic techniques. Initially the designers will ask the users certain questions whose answers should help to construct the specification. A possible partial list of such questions is given below. They have to do with predicates, static constraints, transition constraints, operations, choice of data model, definition of modules, aggregation, specialization and partitioning.

- q1. What facts will be registered in the data base?
- q2. What situations (co-existing facts) are not valid?
- q3. What situation changes are not valid?

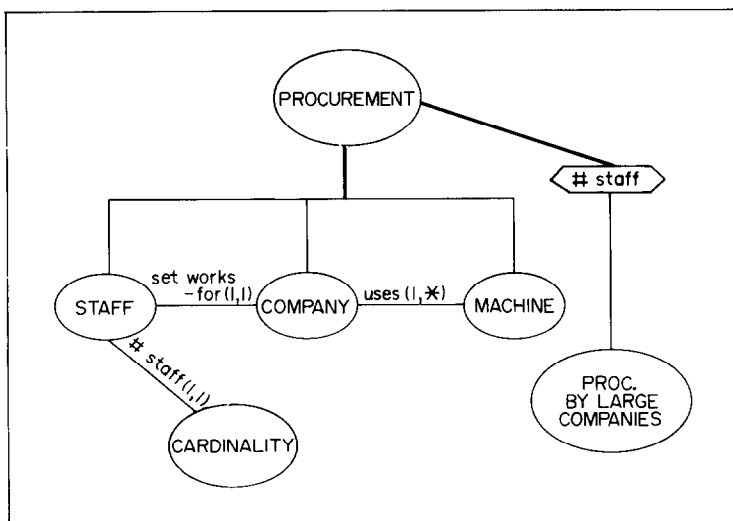


Fig. 10. Aggregation.

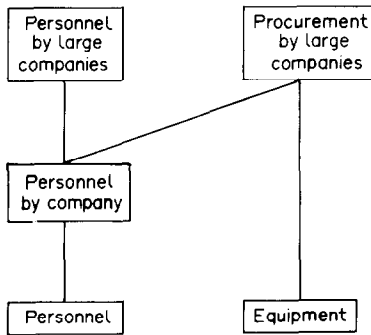


Fig. 11. Modular interdependence.

- q4. What actions performed in the enterprise can affect the information?
- q5. What class of data base management system is envisaged?
- q6. What sub-systems can be identified?
- q7. What are the interconnections among the sub-systems?
- q8. What sub-systems constituting special cases of other sub-systems should be distinguished?
- q9. What sub-system components should be regarded collectively and treated in a uniform way?

Techniques as developed in the area of software design should be helpful in answering these questions constructively. Similarly, after the specification has been constructed and verified, it should be commu-

nicated to the users graphically or in "natural" language, much as we did in our example by "translating" into English each integrity constraint.

A specification should contain all the information supplied by the users or found appropriate by the designers, regardless of future decisions on implementing them by machine, or manually, or not at all. It is a known fact that certain kinds of constraints are too expensive to enforce, and a cost-effectiveness study may, for instance, recommend periodical auditing as a viable alternative. To give another example, active systems require special hardware or software features which may not be available. Also, in view of human factor considerations, we may decide not to let the system initiate automatically what we may still want to deliberate on for some reason.

Among the points in the proposed methodology requiring further research is what we might call *sensitivity analysis*, i.e. the systematic determination of what adjustments in a specification are needed when some kind of change is introduced. More work is also needed to characterize formally, in full detail, the modularization process at each level of a specification.

REFERENCES

- [1] J. R. Abrial: Data semantics. In *Data Base Management* (Edited by J. W. Klimbie and K. L. Koffeman). North-Holland, Amsterdam (1974).
- [2] C. W. Bachmann: The role data model approach to

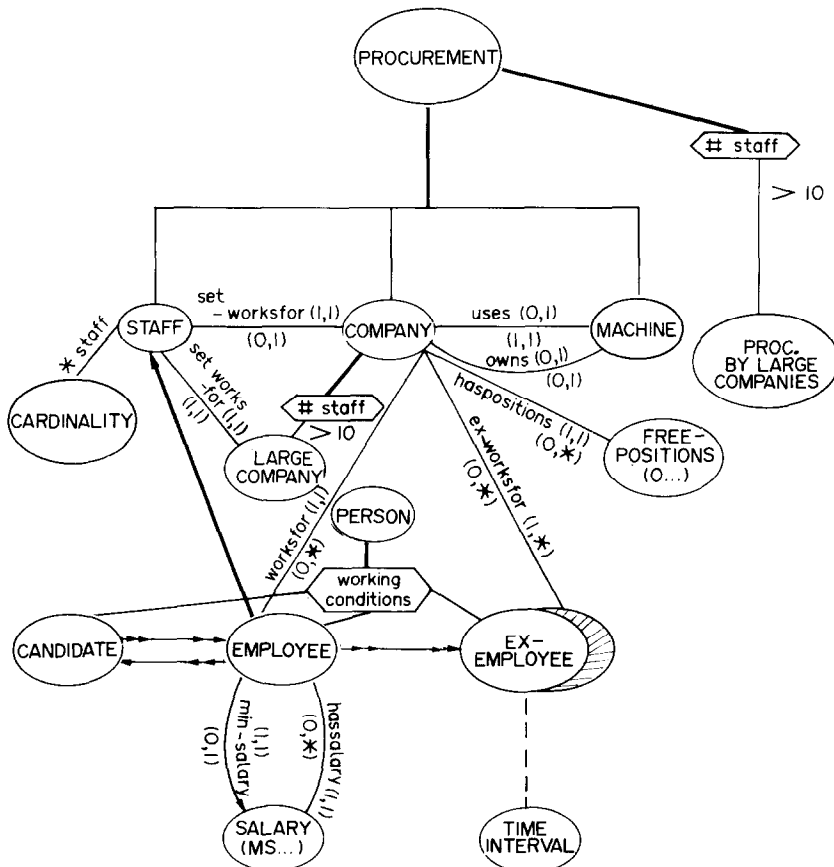


Fig. 12. Complete conceptual schema.

- data structures. *Proc. Int. Conf. on Data Bases*, Aberdeen, Scotland (1980).
- [3] H. Biller and E. J. Neuhold: Semantics of data bases: the semantics of data models. *Inform. Systems* 3 (1978).
- [4] A. Borgida and H. K. T. Wong: Data models and data manipulation languages: complementary semantics and proof theory. *Proc. Very Large Data Bases*, pp. 260–271 (1981).
- [5] M. L. Brodie: On modelling behavioural semantics of databases. *Proc. Very Large Data Bases*, pp. 32–42 (1981).
- [6] M. L. Brodie: Association: a database abstraction for semantic modeling. *Proc. 2nd Int. Conf. on Entity-Relationship Approach*, Washington, pp. 583–608 (1981).
- [7] M. L. Brodie and S. N. Zilles (Eds.): *Proc. Workshop on Data Abstraction, Database and Conceptual Modelling* (1980).
- [8] R. M. Bustall and J. A. Goguen: An informal introduction to specification using CLEAR. In *The Correctness Problem in Computer Science* (Edited by R. Boyer and J. Moore). Academic Press, New York (1981).
- [9] M. A. Casanova and P. A. Bernstein: A formal system for reasoning about programs accessing a relational database. *ACM TOPLAS* 2(3), 386–414 (1980).
- [10] M. A. Casanova, J. M. V. de Castilho and A. L. Furtado: Properties of conceptual and external database schemas. *Proc. Formalization of Programming Concepts* Peniscola (1981).
- [11] M. A. Casanova and A. L. Furtado: A family of temporal languages for the description of transition constraints. *Proc. Workshop on Logical Bases for Databases* (1982).
- [12] J. M. V. de Castilho, M. A. Casanova and A. L. Furtado: A temporal framework for database specifications. *Proc. Very Large Data Bases*, Mexico pp. 280–291 (1982).
- [13] B. Chellas: *Modal Logic: An Introduction*. Cambridge University Press (1980).
- [14] K. L. Clark: Negation as failure. In *Logic and Data Bases* (Edited by H. Gallaire and J. Minker), pp. 293–322. Plenum Press (1978).
- [15] E. W. Dijkstra: *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey (1976).
- [16] H. Ehrig and W. Fey: Methodology for the specification of software systems: from formal requirements to algebraic design specifications. *Proc. GI-11. Jahrestagung* (Edited by W. Brauer), pp. 255–264. Springer-Verlag (1981).
- [17] H. Ehrig, H. J. Kreowski and H. Weber: Algebraic specification schemes for data base systems. *Proc. Very Large Data Bases*, pp. 427–440 (1978).
- [18] E. Falkenberg: Concepts for modelling information. In *Modelling in Data Base Management Systems* (Edited by G. M. Nijssen) North Holland, Amsterdam (1976).
- [19] A. L. Furtado, P. A. S. Veloso and J. M. V. de Castilho: Verification and testing of S-ER representations. In *Entity-Relationship Approach to Information Modelling and Analysis* (Edited by P. P. Chen). ER Institute (1981).
- [20] H. Gallaire: Impacts of logic on data bases. *Proc. Very Large Data Bases*, pp. 248–259, Cannes (1981).
- [21] S. L. Gerhart et al. An overview of Affirm: a specification and verification system. *Proc. IFIP*, pp. 343–348 (1980).
- [22] 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 *Current Trends in Programming Methodology* (Edited by R. T. Yeh), Vol. IV, pp. 80–149. (1978).
- [23] M. R. Gustafsson, T. Karlsson and J. Bubenko, Jr.: A declarative approach to conceptual information modeling. T. R. 8. SYSLAB, Göteborg (1981).
- [24] J. V. Guttag, E. Horowitz and D. R. Musser: The design of data type specifications. In *Current Trends in Programming Methodology* (Edited by R. T. Yeh), Vol. IV, pp. 60–79 (1978).
- [25] M. Hammer and D. McLeod: The semantic data model: a modelling mechanism for data base applications. *Proc. ACM/SIGMOD Int. Conf. on Management on Data* (1979).
- [26] D. Harel: *First-order dynamic logic*. In *Lecture Notes in Computer Science*, Vol. 68. Springer-Verlag, Berlin (1979).
- [27] C. Hewitt: “PLANNER: a language for providing theorems in robots”. *Proc. IJCAI* (1971).
- [28] ISO TC97/SC5/WG3-81. Concepts and terminology for the conceptual schema and the information base. J. J. van Griethuysen (Ed.), (1982).
- [29] R. Kowalski: Logic as a database language. T. R. Department of Computing, Imperial College (1981).
- [30] B. Liskov and S. Zilles: Specification techniques for data abstractions. *IEEE Trans. on Soft. Engng SE-1* (1975).
- [31] Z. Manna and A. Pnueli: Verification of concurrent programs: the temporal framework. In *The Correctness Problem in Computer Science* (Edited by S. Boyer and J. S. Moore). Academic Press, New York (1981).
- [32] D. McLeod and R. King: Applying a semantic database model. In *Entity-Relationship Approach to Systems Analysis and Design* (Edited by P. P. Chen), pp. 193–210. North Holland, Amsterdam (1979).
- [33] M. Minsky: Minsky’s frame theory. *Proc. Theoretical Issues in Natural Language Workshop*. Cambridge, Mass. (1975).
- [34] S. B. Navathe and J. Lembke: On the implementation of a conceptual schema model with a three-level DBMS architecture. *Proc. Nat. Comput. Conf.* (1979).
- [35] P. Paolini: Verification of views and application programs. *Proc. Workshop on Formal Bases for Databases*. Toulouse (1979).
- [36] D. Parnas: On the criteria to be used in decomposing systems into modules. *Comm. ACM*. 15(12) (1972).
- [37] Rescher and Urquhart: *Temporal logic*. Library of Exact Philosophy. Springer Verlag, Berlin (1971).
- [38] N. Roussopoulos: CSDL: a conceptual schema definition language for the design of data base applications. *IEEE Trans. on Soft. Engng SE-5*(5), 481–496 (1979).
- [39] C. S. dos Santos, E. J. Neuhold and A. L. Furtado: A data type approach to the entity-relationship model. In *Entity-Relationship Approach to Systems Analysis and Design* (Edited by P. P. Chen), pp. 103–119. North-Holland, Amsterdam (1980).
- [40] U. Schiel: The temporal-hierarchical data model (THM). Bericht 10/82, Institut für Informatik, Universität Stuttgart (1982).
- [41] U. Schiel: A semantic data model and its mapping to an internal relational model. In *Databases: Role and Structure* (Edited by P. Stocker). Cambridge University Press (1983) to appear.
- [42] U. Schiel: An abstract introduction to the temporal-hierarchical data model (THM). *Proc. Very Large Data Bases*, Florence (1983).
- [43] J. R. Shonfield: *Mathematical Logic*. Addison-Wesley, New York (1967).
- [44] J. M. Smith and D. C. P. Smith: Database abstractions: aggregation and generalization. *ACM TODS* 2 (1977).
- [45] P. A. S. Veloso, J. M. V. de Castilho and A. L. Furtado: Systematic derivation of complementary specifications. *Proc. Very Large Data Bases*, Cannes pp. 409–421 (1981).
- [46] H. Weber: Modularity in data base system design. In *Proc. Joint IBM/Univ. Newcastle upon Tyne Seminar* (Edited by B. Shaw), pp. 15–48 (1979).
- [47] E. Yourdon and L. L. Constantine: *Structured Design*. Yourdon Inc. (1975).