A MODAL SYSTEM OF ALGEBRAS FOR DATABASE SPECIFICATION AND QUERY/UPDATE LANGUAGE SUPPORT

F Golshani, T S E Maibaum, M R Sadler

Department of Computing, Imperial College

Abstract

Rather than formalising general properties of database systems and defining abstract languages for databases, in this paper we provide a formal system for reasoning about specific properties of each application and the specification of query/update functions which are particular to that application. We regard a database as a dynamic object and use a system of modal logic similar to Hoare-style program logic for its specification. The possible worlds in our modal system are the (correct) database instances. Each database instance is defined as a many-sorted algebra where the signature of the algebra constitutes the basis for the database schema. Concepts related to database instances such as queries and (static) integrity constraints are simply well-formed expressions on the signature. Similarly, at the dynamic level, we define notions such as transition constraints and update operations as expressions of the modal system. The paper includes a section on the areas where further work has been done.

1. Introduction

To remain faithful to the real world, databases are continuously modified. To maintain the correctness of the database system through modifications, certain rules and criteria called consistency constraints must be observed at all times. Depending upon the "modification/retrieval ratio" of the particular database system (that is the number of times that a typical item is queried before being updated to something else [Sch 71]) and the "life time" of data objects in the database, the importance of these constraints becomes more (or less) obvious. In a banking environment, for example, where the database has a relatively long life time, the rate of modifications to personal accounts is high; the database is therefore more prone to inconsistency and rigorous rules are introduced to (at least partly) avoid errors.
Two different classes of consistency constraints can be identified: static (integrity) constraints which restrict each database instance to (ranges of) correct values, and dynamic constraints which guard the database through updates. Within the first group one can again recognise two slightly different types. The first type, which we call "simple data constraints", are those which restrict the values of the individual data objects, eg: "age of no employee can be less than 16", or "all salaries are more than 15K". The second type which will be called "aggregate data constraints", are those which state certain restrictions on the values for a collection of data objects; eg: "the total of all salaries in a certain department is less than a given number", or "the number of tickets sold for any particular flight must not be greater than the number of seats in the aircraft assigned to it". A common example for rules governing updates (called "transition constraints" [CaFu 82]) is "salaries must not decrease".

In this paper we intend to provide a setting for the specification of databases in a formal system which lends itself easily to the specification of individual applications as well as to the design of general purpose query and update languages suitable for any application specified. Below is an outline of our approach.

We clearly distinguish between query facilities and update operations. Update operations change the state of the database; thus at update level databases are dynamic. On the other hand, at query level we deal with only one instance of the dynamic database; we therefore, in turn, formally define what we mean by a 'database instance'. The aims of this work will thus be:

1a - to provide a specification language for specifying the database schema which determines the properties of individual instances in a particular application;

1b - to indicate how a general purpose query language can be based on the formalism used for the specification of database instances;

2a - to specify the database application, ie the dynamic object;

2b - to design a general purpose update language for the database based on the formalism used for specifying particular applications.

In our approach 1a and 1b are developed hand in hand as are 2a and 2b. At the static level a database instance is regarded as a collection of sets together with a collection of functions on these sets. The database instance is therefore seen as a 'many-sorted algebra'. There are names associated with every set and every function. These symbols are contained in a "signature" (see eg [ADJ-78]). The signature also gives the typing rules for the database mappings. Thus, the signature is the specification for the "type checker" and the "syntax checker" of the language. We will see that 'type errors' in queries can therefore be detected statically. We extend the ordinary notion of functions in two ways. Firstly, functions which return sets of data objects are permitted. Secondly, we introduce a new object 0 standing for the value "inapplicable" or "domain error" for those mappings which are not everywhere defined (eg: the function grade-of which when given a student and a course as arguments may return a number as the mark, is not defined for all combinations of students and courses; not all students take all courses); see [Gol 82a] for details. Full computation power is provided in the query language by including a wide range of operations which are fixed across all applications. Queries are simply expressions which are built up out of the symbols in the signature of the algebra together with the operation symbols and which comply with the formation rules given by the query/specification language. The semantics of a query is the value which is assigned to it by the algebra representing a database instance. 'Static constraints' are simply expressions of type boolean which are constructed in the same way as queries and must hold in all algebras. These issues are discussed in section 3.1.

At this point we should mention that although the above development is somewhat non-standard it does not disagree with previous developments. For example, a set-valued function may more conveniently be thought of as a relation (as may a boolean valued function). Static constraints can also be thought of as formulae in the sense of first order logic. Thus we note that an equivalent formalism may be developed based on many-sorted logic rather than universal algebra.

For the specification of 2a and 2b we develop a system based on a special kind of modal logic. Modal logic, which began as an extension to predicate logic, is the logic of necessity and possibility: a proposition is "necessary" if it holds in all reachable worlds, and "possible" if it holds in some reachable world. Modal logic is particularly suited for reasoning about dynamic systems such as databases. In this work, our modal system has similarities to Hoare-style program logic [Gold 82]. The admissable worlds of our modal system are the database instances (ie: many-sorted algebras). Transition constraints then can naturally be viewed as modal expressions built up from the modal operators (yet to be defined) and the symbols in the (common) signature of the algebra. These issues are disscussed in section 3.2.

2. Comparison with extant work

Attempts to provide formal settings for the specification and design of databases and database languages date back several years. Similarities between concepts in mathematical semantics and in database modelling were analysed in [Mai 77]. Based on ideas taken from abstract data types and the notion of higher order functions a primitive formalisation of our present ideas was provided. The concept of database instance (static) was later formalised in [Mai 81]. In [CaBe 80] a language based on a variant of dynamic logic was defined which incorporated the aggregation operators. Using this language, various concepts such as database schema, transactions, database states and integrity constraints were developed. The use of an extended form of logic enabled them to express 'aggregate consistency constraints' in a natural manner.

In [CaFu 82] a family of languages are defined which are based on an extension of temporal logic. (Temporal logic is a special kind of modal logic, see eg [MaPn 79].) Although this extension does not seem to increase the expressive power of the language, it is claimed that it facilitates the description of transition constraints. This work is based on Wolper's extended temporal logic [Wol 82] and contains proofs about decidability and solvability problems. The constraint "salaries never decrease", for example, is expressed as:

$\neg \exists n \exists s\ (\Diamond\ (EMP(n,s);\ \exists s'\ (EMP(n,s') \wedge s>s'\ )))$

where EMP(x,y) indicates that employee x has the salary y. A technical defect in this paper is that it is not clear over which range the variables are quantified. For example, while being in a particular database instance, how can we talk about the objects which may exist in a future database instance? [Nic 83]. s' in the above expression is an example of this phenomenon.

In contrast to our work which clearly distinguishes between queries and updates, in [MSF 80] a database is defined as a set of axioms of many-sorted first order predicate logic which specify all the valid states. Queries and updates are considered uniformly as theorems which must be proved by a theorem-proving process with respect to the database state.

On the algebraic specification of databases, the work presented in [DMW 82] stands out. In this rigorous study, precise specifications are given for many aspects of databases. After defining notions such as conceptual models and external views, they present abstract definitions of query and update operations.

Our work differs from above because we design concrete query/update languages in addition to talking about general (and abstract) properties of such languages. Many other research reports are related to this topic. See [Web 76], [Tod 77], [NiYa 78], [CaBe 80] and [Nic 82] for the study of integrity constraints, and [DaBe 82], [FVU 83] and [SeFu 78] for reasoning about correct updates.

3. Database specification and database language design

3.1 Given an alphabet Λ , we define the vocabulary of our language as the collection of four groups of symbols (a symbol is a sequence of characters): "sort symbols", "variable symbols", "function symbols" and "operation symbols". We assume that the form of each symbol determines to which group it belongs. The operation symbols form an invariant part of the language (as they are the application independent constituent of the query/specification language) and stand for various kinds of standard operations such as arithmetic, boolean, set-theoretic, aggregation, and more complicated ones such as quantifiers and the set-building operator. The difference between operations and functions is that functions are particular to the database application and instances thereof. It is assumed that the two sorts boolean and integer together with the associated operations are present in all specifications.

Simple-type-expressions are inductively defined to be sort symbols or of one of the forms:

$\alpha_1 U \alpha_2$, $(\alpha_1 * \alpha_2 * \ldots * \alpha_n)$

and $P(\alpha_1)$ where for some n for $1 \leqslant i \leqslant n$, α_i is a simple-type-expression.

Given a natural number n, a function-type-expression of arity n has the form

$\alpha_1, \alpha_2, \ldots, \alpha_n \dashrightarrow \beta$

where for $1 \leqslant i \leqslant n$, α_i is a simple-type-expression and β is a simple-type-expression. Operation-type-expressions are defined in a similar manner. For example, the operation-type-expression for the operation symbol "+" is int,int --->int.

A signature is a function which assigns a function-type-expression to each function symbol and a sort symbol to each variable symbol. Thus, the signature is the specification for the type-checker as well as for the syntax-checker of the language. Notice that the variables are typed by the signature and not by the user. There is an unlimited supply of variables of each sort.

Example : We can specify part of a university database as follows: sort symbols 'students', 'courses', 'lecturers', 'integers' and 'boolean'; function symbols 'courses-of', 'is-taking', 'enrollers-of', 'prerequisites-of', 'age-of', 'grade-of', 'lecturer-of', and so on. The unique function-type-expressions for (some of) these function symbols are given below:

lecturer-of	courses ---> lecturers
grade-of	students, courses ---> integers
courses-of	students ---> P(courses)
age-of	students U lecturers ---> integers
prerequisites-of	courses ---> P(courses)

Given a signature, we define the set of well-typed expressions on that signature in the usual inductive way. For example, if Ω_1 is an expression of type α , and Ω_2 and Ω_3 are expressions of type $P(\alpha)$, then

Ω_1 isin Ω_2

is an expression of type boolean,

Ω_2 is-subset-of Ω_3

is an expression of type boolean,

Ω_2 union Ω_3

is an expression of type $P(\alpha)$.

Similarly,

$(\Omega_1$ isin $\Omega_2)$ and $(\Omega_2$ is-subset-of $\Omega_3)$

is an expression of type boolean.
("isin", "is-subset-of", "union" and "and" are all operations of the query language.)

Bound and free occurrences of variables in expressions can be detected syntactically in the usual way. For instance, given an expression of the form $\{\Omega_1 | \Omega_2\}X$, any occurrence of X in Ω_1 or Ω_2 is a bound occurrence. The set building operator $\{...|...\}X$ is a variable binding operator (see [KMM 80], [Gol 82]) in the same sense that $\forall$ and $\exists$ are in normal logic. Of course, the type of the expression Ω_2 must be boolean.

Closed expressions are those in which there are no free occurrences of any variables.

Given a signature Σ, a (static) integrity constraint is any well-formed expression of type boolean on Σ. We will use the symbol Γ_Σ for a set of integrity constraints on signature Σ. A database schema is a signature together with a (possibly empty) set of integrity constraints on that signature.

Example: Here are some examples of integrity constraints on our university database.

- No student can be registered for a course unless she has passed all the prerequisites of that course.

```
forall STUDENT  forall COURSE
        ((STUDENT is-taking COURSE) implies
(prerequisites-of(COURSE)
 is-subset-of accumulated-courses-of(STUDENT))
```

- Maximum number of enrollers for any course is 50.

```
forall COURSE
          (No-of( enrollers-of(COURSE)) LT 50)
```

- The fact that the two functions 'courses-of' and 'enrollers-of' and the relation 'is-taking' represent exactly the same information can be expressed by three constraints of the form:

```
 forall STUDENT  forall COURSE
           ((STUDENT is-taking COURSE) implies
((STUDENT isin enrollers-of(COURSE)) and
            (COURSE isin courses-of(STUDENT)))
```

A query is a closed expression in which any variable is bound only once. We continue our illustration of the university database by constructing a sample query:

- lecturers of all those courses which must be taken before taking Maths.

```
{ lecturer-of(COURSE) |
 COURSE isin  prerequisites-of(Maths) } COURSE
```

The type of the object returned by this expression is P(lecturers) because the function lecturer-of has the function-type-expression courses --->lecturers. COURSE is a variable of type courses. (The appearance of COURSE on the very right indicates the variable which is being bound by the set-building operator).

So far we have only discussed syntactic issues, we shall use the notion of algebra to reason about the semantics. A many-sorted algebra is a function which assigns a set (called carrier to each sort symbol and a function to each function symbol. For a simple-type-expression α the set of all objects of type in an algebra **A** denoted by $|\mathbf{A}|_\alpha$ is defined as follows:

- if α is a sort symbol then $|\mathbf{A}|_\alpha = \mathbf{A}(\alpha)$
- if α is $\alpha_1 U \alpha_2$ then $|\mathbf{A}|_\alpha = |\mathbf{A}|_{\alpha_1} U |\mathbf{A}|_{\alpha_2}$
- if α is $(\alpha_1 * \alpha_2 * .... * \alpha_n)$ then $|\mathbf{A}|_\alpha = |\mathbf{A}|_{\alpha_1} * .... * |\mathbf{A}|_{\alpha_n}$
- if α is $P(\alpha)$ then $|\mathbf{A}|_\alpha = P(|\mathbf{A}|_\alpha)$

The evaluation in **A** of queries is carried out in the usual way.

Defining a database scheme $S=(\Sigma, \Gamma_\Sigma)$ to be a signature and some constraints on it, an algebra **A** is an S-algebra iff:

1. for each function symbol ϕ in the domain of **A**, if $\Sigma(\phi)$ is $\alpha_1, ..., \alpha_n \dashrightarrow \beta$ then $\mathbf{A}(\phi)$ returns an element of $|\mathbf{A}|_\beta$ when given an element of $|\mathbf{A}|_{\alpha_1}$, an element of $|\mathbf{A}|_{\alpha_2}$,.. and an element of $|\mathbf{A}|_{\alpha_n}$.
2. **A** evaluates all the expressions of Γ_Σ as true.

We are now ready to define database instances. A database instance over a schema $S = (\Sigma, \Gamma)$ is the ordered pair (S,**A**) where **A** is a Σ S-algebra.

Notation: Given an expression P of type boolean, for a database instance i, we write $i |= P$ iff i evaluates P as true.

Readers interested in details of the above are referred to [Gol 82] and [Gol 83].

3.2 Databases as dynamic objects

We begin this section by giving our main definitions (1a and 1b below were presented in 3.1). A specification of a database is:

1a - A schema $S = (\Sigma, \Gamma_\Sigma)$, where Σ is a many-sorted signature and Γ_Σ is a collection of well formed boolean expressions over Σ.

1b - A collection of domains $\{D\alpha\}$ of values, one for each sort α of Σ.
The collection DB of S-algebras (database instances) over $\{D\alpha\}$.

2a - An extension Σ^1 of Σ^1 to include update symbols u_0, u_1,.... and the modal construct [ ], and to include functions 'in-α', one for each sort α, of type α ---> bool.
And an extension $\Gamma^1{}_{\Sigma^1}$ of Γ_Σ to include transition constraints.

2b - A collection U of update functions $\underline{u}_0$, $\underline{u}_1$,.... where each $\underline{u}$ is a mapping from DB to DB, and such that these functions satisfy $\Gamma^1{}_{\Sigma^1}$. (We shall define what is meant by satisfaction below. We underline the names of update functions, as in $\underline{u}_1$, to avoid confusion with the corresponding symbol u_1 in Σ^1.)

The rest of this section is concerned with explaining and illustrating 2a and 2b above. Note that 1a and 2a are syntactic components of the definition, and that 1b and 2b are related to semantic concepts. Essentially it is the syntactic components that are used in connection with proving correctness of implementations and reasoning about database properties whereas the semantic components are used in connection with evaluation of expressions denoting queries. We shall indicate how both these functions are supported by our definition. We assume the presence of a deductive system for the language, although here we omit any such detail.

At the dynamic level we want to be able to talk about objects which are not necessarily present in a given database instance - that is, we want to reason about potential objects as well as about the "concrete" ones of any given instance. To facilitate this we define each S-algebra over the same collections of objects (the $D\alpha$'s), and pick out those objects which are "real", "actual" or "concrete" in each algebra by use of the in- α functions.

Quantifiers now range over all potential objects; but we could also introduce local quantification by use of the construct:

forall x (in-α(x) implies...

See [HuCr 68] for a detailed treatment of the problems associated with the range of quantifiers in a modal logic setting. And see also [Man 81] where a distinction is made between local and global symbols (in particular variables) for an alternative way of handling the potential/actual distinction.

We turn now to the more interesting parts of the definition, those dealing with updates. We explain our syntactic treatment first. Although there are strong similarities with program logic [Gold 82] the material is probably unfamiliar to most readers. So we proceed more carefully (and when necessary, formally).

The definition of well-formed expressions over Σ is extended to well-formed expressions over Σ^1 by including the construct:

$[u_m]$P

as an expression of type boolean, where P is an expression of type boolean and u_m is an update symbol. The constructs $[u_m]$ have no effect on whether or not variables are bound and we use the square brackets to exploit the analogy with [Gold 82]. Intuitively the expression $[u_m]$P is read as 'after the update u_m is performed, P will be true'; that is, the $[u_m]$ act as operators in a similar way to the, perhaps, more familiar modal operators $\Box$, $\Diamond$ and **Next**.

The logic used for deriving consequences from the schema can now be extended by adding the following axiom schemata:

Distribution:
[u](P implies Q) iff ([u]P implies [u]Q)

Negation:
not [u] P iff [u] not P

Quantification:
(forall x [u] P(x)) iff [u](forall x P(x))

and the rule: $\dfrac{P}{[u]P}$

ie if P is a theorem, then so is [u]P.

In the above P and Q are expressions of type boolean over Σ^1, that is, they themselves may include modal symbols; P(x) is an expression of type boolean over Σ^1 with at most the variable x free; and u is an update symbol of Σ^1, that is, we are using u as a metavariable over the u_m.

The quantification axiom might seem strange, but note that we are quantifying over all potential objects. It is worth noting that expressions such as:
"forall x [u]P(x)" and "[u] forall x P(x)" should not be confused with the similar expressions using "local quantification", for which the quantification axiom does not hold.

One important feature of this system is that our modal operators can be "pushed around" quite freely within our logic. The static constraints and the transition constraints act independently and our logic reflects this. The behaviour of

$\underline{u}_0, \ldots, \underline{u}_m, \ldots$ is governed only by the transition constraints.

The notion of satisfaction is easily extended to cope with the modal operators. Given a database instance i, an update symbol u_m and a boolean expression P :

$$i \models [u_m]P \text{ iff } \underline{u}_m(i) \models P.$$

It is straightforward to check that the axioms (distribution, negation and quantification) and the rule presented previously are sound. What we are doing here is replacing the more conventional relational semantics for modal logic by a functional semantics.

Examples: Let us look at some examples of transition constraints for our university database:

- ages cannot be reduced:

```
forall x  forall y ((age-of(x)  is  y)
             implies ([u] (age-of(x)  GE  y)))
```

This expression reads as follows: for any student x and any age y, if the age of x at present is y then after performing any update the age of x will be at least y.

- certain course, say EE1, once inserted to the database can never be deleted.

```
in-course(EE1) implies [u](in-course(EE1))
```

Both these examples can, of course, be captured by the general operator $\Box$. Literally, these two examples only talk about a "next" state, but a simple application of the rule and schemata presented above allow the modal constructs to be iterated to any length. (See 4.1 below). The specific operators come into their own when we wish to make assertions about particular updates:

```
not in-student(Jack)
          implies  [u0] in-student(Jack)
```

for example, asserts that $\underline{u}_0$ involves entering "Jack" into the database. (In practice we would write 'add-Jack' instead of 'u_0'.)

It should be noted that the $[u_m]$'s are specific and are not parameterized with respect to the data being manipulated. For example in the case of adding a new student to the database: add-Jim, add-Jack, add-Carol, etc., all have to be included in the list $u_0, \ldots, u_m, \ldots$ This situation is clearly not ideal. Neither do we want to have to specify a separate update for each change that we might like to make to a function value. We therefore introduce parameterized updates. Syntactically we need to modify 2a of our basic definition by requiring that each update symbol be typed (to pick out those expressions which can be used as parameters). For example:

'add-student' would be of type 'students' and we can make assertions about the adding of "Jack" by using

```
[add-student(Jack)]
```

or about the adding of arbitrary students by

```
[add-student(x)]
```

where x is any expression of type 'students'. Or if 'increase-sal' is of type 'person * nat'

```
forall x  forall y  forall z (  sal-of(x) is y
implies  [increase-sal(x,z)] sal-of(x) is (y+z))
```

would be a suitable transition constraint about the increase of salaries.

And, of course, we can form more complicated expressions by using constructs like:

```
[increase-sal(employee-of(Jack),y/10]
```

We can manipulate the [u(, ,...)] in the same way as the [u]. Note that occurences of variables in such parameterised update constructs are free. And the semantics extends straightforwardly by modifying 2b so that each $\underline{u}_m$ is a mapping from DB x (Dα x ...) to DB, where u_m is of type α *... .

4. Some related aspects

In this section we will address two further issues: how our modal system relates to others, and how transactions can be specified using our modal system.

4.1 Since we have the specific modal operators $[u_m]$, the other modal operators $\Box$ and $\Diamond$ are not essential for the expression of transition constraints. ($\Box$ is to be read, as 'all reachable database instances', and $\Diamond$ is to be read as 'some reachable database instance'.) We can provide semantics as follows:

$$i \models \Box P \quad \text{iff } \underline{u}_n(\underline{u}_{n-1} \ldots \underline{u}_0(i) \ldots) \models P$$
for all sequences $\underline{u}_0, \underline{u}_1, \ldots, \underline{u}_n$ of updates,

and

$$i \models \Diamond P \quad \text{iff } \underline{u}_n(\underline{u}_{n-1} \ldots \underline{u}_0(i) \ldots) \models P$$
for some sequence $\underline{u}_0, \underline{u}_1, \ldots, \underline{u}_n$ of updates.

Given a particular sequence of updates, **Next** and **Until** can be given semantics in the usual way [Man 79].

Syntactically we can regard $\Diamond$ as an abbreviation for not $\Box$ not, and add the axiom schema:

$$\Box P \text{ implies } P \text{ ;}$$

$$\Box (P \text{ implies } Q) \text{ implies } (\Box P \text{ implies } \Box Q) \text{ ;}$$

$$\Box P \text{ implies } \Box\Box P;$$

and the rule $\frac{P}{\Box P}$ to reason about $\Box$ and $\Diamond$.
This system is usually referred to as S_4, and it is easy to check that these axiom schemata (and rule) are sound, as long as the null update is included in the collection U of update functions. That is, null: DB ---> DB is such that null(i) = i.

4.2 By a transaction we mean a sequence of update operations. For a sequence of updates u_0, u_1, ... u_n, we can use the construct $[u_0;u_1;$ given a database instance i and a boolean expression P we have:

$$i \models [u_0; \quad u_1; \quad \ldots \quad u_n]P$$
$$\text{iff } u_n(\ldots u_1,(u_0(i))\ldots) \models P$$

As each u_m has been defined as a mapping from DB to DB (ie. from instances to instances) this amounts only to the composition of updates.

$[u_0; \ldots u_n]$ is equivalent to $[u_n]\ldots[u_1][u_0]$

However we could weaken our definition of the u_m so that they are mappings from the collection of algebras to the collection of algebras over the language. Thus integrity constraints need not be satisfied during the transaction, only at the end (algebras need not satisfy the integrity constraints). We are currently investigating this area.

5. Conclusion

One of the main contributions of the theory of abstract datatypes to programming has been the introduction of application dependent objects and operations to be manipulated directly at a logical level by programs (and programmers). This has eased the burden of program design because analysis can be performed at the abstract, logical level by both the designers and users of the program. We feel that database designers and users should benefit from the same approach.

Much of the effort in database design in the past has concentrated on the implementation oriented approach exemplified by the various traditional models: relational, hierarchical, etc. These models provided general purposes tools for query, formation, information representation, and the definition of static constraints (eg: various kinds of dependencies). The use of these languages in particular applications was unstructured in the same sense that data representation and manipulation was before the use of abstract data types. Users have to formulate queries in terms of the representation (eg: relations and types) and its associated operations (eg: join, projection, etc., in the case of relational algebra) instead of the concepts which might be more familiar. Updates were even less formalised as none of the traditional models addressed this problem directly. Typically, the only update operations available were again primitive, implementation dependent ones (eg: insert a tuple in a relation).

Recently efforts have been made to apply the techniques of abstract data types to data base specification. Thus application dependent objects and operations are becoming more acceptable. However, these presentations have of updates and database instances and have tended to concentrate on static constraints.

We have attempted above to provide a theory of databases which allows designers and users to deal with the objects and operations logically relevant to the application both for queries and updates.

Designers can specify the properties of the primitive (application dependent) query operations, ie: static constraints, using what is essentially first order logic augmented with general query forming operators thought to be suitable for database specification. The dynamic properties of databases are again defined using application dependent primitive update operations by means of a modal logic. These general operators, first order logic and the modal system, are a fixed specification language for database applications. They have the further advantage that a user can formulate queries and updates using this formal system. Thus the specification language is also a general purpose query and update language.

Moreover, such specifications offer the same advantages as abstract data type specifications. One can decide on an optimal implementation method and then prove its correctness with respect to the specification.

6. References

[ADJ 78] Goguen J A , Thatcher J W , Wagner E G "An initial algebra approach to the specification, correctness, and implementation of abstract data types" In "Current trends in programming methodology", Vol. IV , pp 81-149 , Prentice Hall 1978.

[CaBe 80] Casanova M A , Bernstein P A "A formal system for reasoning about programs accessing a relational database" ACM TOPLAS , Vol. 2 , No. 3 , pp 386-414 , July 1980.

[CaFu 82] Casanova M A , Furtado A L "A family of temporal languages for the description of transition constraints" Workshop on logical bases for data bases, Toulouse 82.

[DaBe 82] Dayal U , Bernstein P A "On the correct translation of update operations on relational views" ACM TODS , Vol. 8 , No. 3 , pp 381-416 , Sept 1982.

[DMS 82] Dosch W , Mascari G , Wirsing M "On the algebraic specification of databases" Proc. of 8th VLDB Conf. Mexico City , Sept 1982.

[Gold 82] Goldblatt R "Axiomatising the logic of computer programming" Lecture Notes in Computer Science 130 , Springer-Verlog 1982.

[Gol 82] Golshani F "Varqa, a functional query language based on an algebraic approach and conventional mathematical notation" PhD thesis, Theory of Computation Report No. 43 Warwick University , UK.

[Gol 82a] Golshani F "Growing certainty with null values" Research Report DOC 82/22 , Imperial College , UK.

[Gol 83] Golshani F "A mathematically designed query language" Research Report DOC 83/1 , Imperial College , UK.

[HuCr 68] Hughes G E , Cresswell M J "An introduction to modal logic" Methuen and Co. Ltd , London , 1968.

[KMM 80] Kalish D , Montague R , Mar G "Logic, techniques of formal reasoning" Harcourt Brace Jovanovich inc. , 2nd ed., 1980.

[Mai 77] Maibaum T S "Mathematical semantics and a model for databases" Proc. IFIP 77 , (Gilchrist ed.) pp 133-138.

[Mai 81] Maibaum T S E "Database instances, abstract data types and database specification" To appear in the Journal of Computing.

[MSF 80] Maibaum T S E , dos Santos C S , Furtado A L "A uniform logical treatment of queries and updates" Research Report CS-80-11 , University of Waterloo , Canada.

[Man 81] Manna Z "Verification of sequential programs: Temporal Axiomatization" Report No. STAN-CS-81-877 , Stanford University 1981.

[MaPn 79] Manna Z , Pnueli A "The modal logic of programs" Report STAN-CS-79-751 , Stanford University , 1979.

[Nic 82] Nicolas J-M "Logic for improving integrity checking in relational databases" Acta Informatica 18 , pp 227-253 , 1982.

[Nic 83] Nicolas J-M Private communication.

[NiYa 78] Nicolas J-M , Yazdanian K "Integrity checking in deductive databases" in "Logic and databases" (Gallaire, Nicolas eds.) , pp 325-344, Plenum Press , New York , 1978.

[Sch 71] Schwartz J T "Abstract and concrete problems in the theory of files" in "Database systems" Courant Computer Science Symp. 6 , (Rustin ed.), Prentice Hall , 1971.

[SeFu 78] Sevick K C , Furtado A L "Complete and compatibale sets of update operations" In Int. Conf. on Management of data (ICMOD) , Milan Italy , June 1978.

[Tod 77] Todd S "Automatic Constraint maintenence and updating defined relations" Proc. IFIP 77 , (Gilchrist ed.) North-Holland , 1977.

[Web 76] Webber H "A semantic model of integrity constraints on a relational database" Modelling in database management systems, North-Holland , 1976.

[Wol 81] Wolper P
"Temporal logic can be more expressive"
Proc. of 22nd Symp. on Foundation of Computer
Science , Nashville, TN,
October 1981.