Meta-Functions and Contexts in an Object-Oriented Database Language

Michael Caruso
Innovative Systems Techniques
One Gateway Center, Suite 910
Newton, MA 02158

Edward Sciore[1]
Boston University CS Department
111 Cummington St.
Boston, MA 02215

ABSTRACT

VISION is an object-oriented database system currently used commercially to develop investment analysis and other large statistical applications. Characteristic of these applications, beside the standard issues of structural and computational richness, is the need to handle time, versions, and concurrency control in a manner that does not produce combinatoric complexity in object protocol. This paper describes the approach taken by VISION in addressing these issues.

1. Introduction

A lot of recent effort has been directed towards extending database technology to non-traditional domains, such as CAD/CAM [KW], software engineering [PL], and office automation [BP]. In this paper, we consider another non-traditional problem domain -- the application and analysis of large statistical databases in general and financial databases in particular. Throughout the rest of this paper, examples are drawn from the investment analysis domain, although the principles discussed are generally applicable.

We assume that the reader has a basic familiarity with the ideas and principles of object-oriented languages, and can appreciate the abstraction power that object-orientation provides. The issues of how to use an object-oriented language efficiently for database access have been addressed in [CS,MSOP,OBS]. Much of that work is also relevant to our application domain, so we shall not repeat it here. Instead, we focus on some other issues that have been relatively overlooked. These issues are crucial for investment databases, and therefore have been strong influences on the design of the VISION system. We contend that they are also important for any database system.

Section 2 provides a brief introduction to the area of investment databases and the issues they raise. Section 3 discusses the problems with traditional data languages, and motivates the use of object-oriented database languages. Section 4 overviews the VISION database system and language. VISION takes a functional approach to data modelling, and its insistance on representing all information as functions turns out to be the key to the rest of the paper.

Sections 5 and 6 form the heart of the paper. In Section 5, we consider functions as first-class objects. This extension to the language leads to the notion of a meta-function, which provides an elegant way of organizing different types of function into a single hierarchy. Section 6 introduces the notion of context, and shows how it can be used to model time, versions, update, and concurrency control in a single framework.

2. Investment Analysis

Considerable structural and behavioral complexity exists in investment databases and applications. An investment analyst may require data on companies, securities, portfolios, and so on. Such data is diverse in structure, and needs to be modeled at varying levels of abstraction. For example, consider the class *Company*. Although all companies have some common properties, different categories of company can be modeled quite differently. For example, large companies have a different structure from small companies, and each company has certain industry-specific properties.

[1] Work by the second author was funded in part by NSF grant IRI-8710137

Moreover, companies will have to be modeled at different levels of detail, according to the kind of data available.

The class of securities is another example of the complex structure of investment objects. Investment instruments are not simply common stocks and corporate bonds, although even these instruments have significant structural and behavioral differences. Numerous modern financial instruments such as options, futures, and indices are derived from or associated with other instruments. Investment decisions are typically made in terms of strategies representing synthetic aggregations of other instruments.

In addition to the large number and flexible structure of classes in an investment database, there is also a substantial amount of data. Much of this data is historical. For example, a database may represent 10,000 companies. Each company may have daily pricing along with quarterly and annual report data stored for the past 10 or more years. Typically, databases are on the order of several hundred megabytes, although databases an order of magnitude larger are possible and required for certain types of analysis.

Queries against investment databases tend to be more complex than traditional database queries. Relatively few are simple information retrieval queries. Typical queries involve restructuring objects hierarchically, and applying arbitrary selection predicates to them. Because time is a fundamental part of the world modeled by these databases, all queries are at least sensitive to their temporal context. Many queries deal explicitly with time-varying data, performing operations such as regression, filtering, and zero-crossing analysis.

Updates to investment databases tend not to fit a single model of concurrency control In general, three types of data are present -- metadata representing application and schema refinements, real-time market data and news. and infrequently updated bulk historical data Each imposes different consistency rules on its application which, in general, cannot be mutually satisfied by a single model of concurrency control.

3. Object-Oriented Database Systems

In order for a system to be usable for investment analysis, it must have two basic capabilities: *structural expressiveness*, in order to model the investment data effectively, and *operational expressiveness*, in order to write complex queries on the data. Both spreadsheets and traditional relational database systems are unsatisfactory, since their flat model of data is inappropriate for highly structured data types. Semantic database models [KM] do provide structural expressiveness; however, they share with spreadsheets and database systems a limited operational expressiveness. In order to write complex queries, users of these systems must rely on an external, "host" programming language.

Traditionally, database management systems are used to codify the declarative semantics of an application, with host language programs codifying its operational semantics. As applications become more complex, this separation is not so clean. A database management system can only manage the declarative semantics of the shared, persistent data; the application programs must define and manage transient data themselves. Although application programs support the bulk of an application's operational definition, the database's data manipulation language provides some operational capabilities as well. And because a significant component of a complex type's definition is operational, it becomes increasingly difficult to separate the declarative structure of the data from its operational semantics. In addition, although application programs can be considered part of a shared, persistent information base, they typically are not managed by the data management system.

Object-oriented languages provide a solution to these problems. Their abstraction capablity and their ability to reference objects of any structure make them ideal for complex data modelling. Combining this abstraction with general-purpose programming facilities creates a language with a remarkable expressiveness for writing complex queries. Thus what we need is an object-oriented database manipulation language. There are several well-known issues that must be addressed: the system must allow unbounded collections, and provide fast access and update to them; it must be able to optimize queries to take advantage of alternative access paths; and the distinction between permanent and transient data must be as transparant as possible. Discussion of these issues can be found in [MSOP,OBS]. This paper considers yet more issues that are just as important, but less well understood.

4. The VISION Data Language

VISION is an object-oriented system which integrates programming language and database system. VISION adopts a functional view of data. Consequently, it shares many of the same properties of other functional data languages (such as DAPLEX [Sh]), although its syntax looks more like Smalltalk [GR]. In order to clarify our presentation, we present a simplified view of the language; for more details, the reader is referred to [CS,V].

A VISION database consists of several *objects*. Each object is an instance of a particular *class*. Objects themselves are non-information bearing. That is, there is no specification of what these objects are, what values they have, or how they relate to any other objects. Instead, all information about an object is embodied in its *functions*. A function is a mapping from one class into another. The set of available functions for an object is determined by its class, and is sometimes called the *scheme* or *behavior* of the class. Classes can be organized into class hierarchies, with objects *inheriting* the functions defined in its superclasses. The issues surrounding inheritance in VISION are not relevant to this paper, but are discussed in [S].

Consider Figure 1, which we shall use as our running example. Six classes are shown. The function *industryOf* maps *Company* to *Industry*, the function *name* maps *Industry* to *String*, and so on.

Functions can be either enumerated or procedural. An enumerated function contains a single stored value for each of its objects; such a function is often called an *instance variable*. A procedural function is also known as a *method*. VISION is different from typical object-oriented languages [GR, OBS] in that it does not distinguish between these different types of function. In particular, a user query does not need to know if a given function is procedural or not. Note that in Figure 1 there is no clue as to which functions are procedural.

Functions are applied to objects using *messages*. VISION uses Smalltalk-like syntax to represent messages. That is, if x is an object in class c, and f is a function defined in c, then the expression "x f" applies the function f to the object x, and returns the appropriate object as its value. Parameters are expressed using colons, so that the expression "x p:c" calls function p with argument c. The only parameterized function in Figure 1 is *balSheetFrom:To:*. Executing this function prints a balance sheet for the company between two given dates.

The class *ENV* in Figure 1 is special, in that it denotes the *user environment*. Functions in this class correspond to objects that the user has named individually. So for

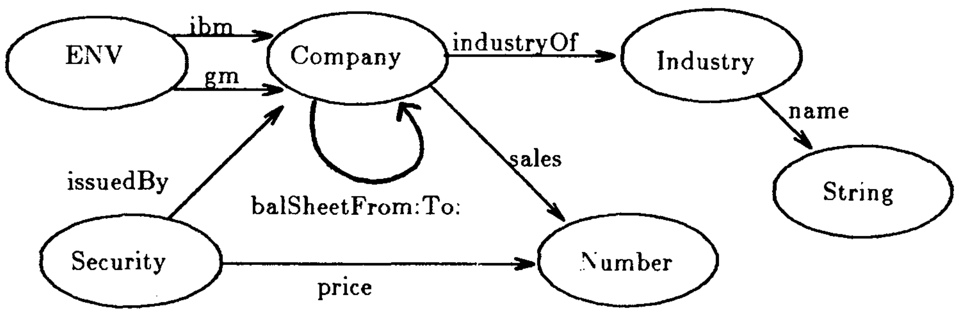

Figure 1. Some Classes and Functions

example the function *gm* maps to the appropriate object in *Company*. These functions can be thought of as direct links to the database; calling them avoids having to search associatively. For example, if the user of this scheme wanted to access another company (say, "ford"), he would have to search in some way through the collection of all company objects to find it.

Figures 2a and 2b provide examples of some simple queries on the scheme of Figure 1.

Because VISION is a database language, queries tend to operate on *collections* of objects. One important collection is the collection of all objects in a class. This collection is obtained by using the function *instanceList*; for example, the expression "*company instanceList*" returns the collection of all companies in the database.

Behavior is expressed by means of *blocks*, which are parameterized sequences of statements. For example, the block *[| :x | x + 2]* adds *2* to its argument. Blocks can be used to define methods. For example, the method *peRatio* can be defined by the block *[price / earnings]*. Blocks can also be used to define selection conditions for queries. The queries in Figures 2c and 2d illustrate these points.

5. Functions and Meta-Functions

5.1. Functions as Objects

In typical object-oriented langauges, it is possible to store certain *meta-information* about each object, such as the date it was last changed, and who changed it. In a functional language, it seems more appropriate to keep such information for each function of the object. This change in perspective allows us to keep information at a finer granularity than objects: functions can change independently, and meta-information about these changes can be recorded.

Functions are clearly a natural unit of abstraction in a functional database, even more so than objects. Moreover, there are times that the added flexibility of functions is essential. One particular example is in the field of historical databases [CT]. Early work in historical databases tried to keep historical data for each tuple (i.e. object) in a relation, but it was later observed that keeping a history for each attribute (i.e. function) was more natural for the user, and led to a simpler formalism.

Since functions are a fundamental unit of abstraction, it is natural to consider treating functions as objects. If x is an object that can call function f, then the expression "x $:f$" denotes the function-object for f with respect to x. Thus there is a many-one relationship between function-objects and instance objects: a function-object is associated with exactly one instance object, and an instance object has one function-object for each function that it can call.

We specify a function-object by prepending a colon to the function name. Note that

a) *Return the name of GM's industry.*

```
gm industryOf name
```

b) *Print a balance sheet for IBM from 1981 to 1984.*

```
ibm balSheetFrom: 1981 To:1984
```

c) *Return a collection of auto companies.*

```
company instanceList select:[industryOf name = "auto"]
```

d) *Create a collection of industries having some company with sales>50.*

```
company instanceList select:[sales>50] send:[industryOf]
```

Figure 2. Some VISION Queries

there is a considerable difference between the expressions "*x f*' and "*x :f*'. In the first expression, we are referring to the value of the function for *x*; in the second expression we are referring to the function-object itself. In the terminology of logic, the latter expression is called the *intension* of the function, and the former expression its *extension.*

5.2. Meta-Functions

Since function intensions are objects, they can have functions defined for them. We call such functions *meta-functions.* One meta-function that is defined for all function-objects is *value*, which returns the extension of the function. That is, the expression "*gm :sales value*" is exactly equivalent to "*gm sales*". Meta-functions are an elegant way of obtaining access to meta-information. For example, "*gm :sales storedAt*" returns the date when the function was stored, and "*gm :sales storedBy*" returns who stored it.

Function-objects also must belong to classes. The class of a function-object determines the meta-functions that are available to it. Function-classes are organized into a hierarchy, as shown in Figure 3.

We have seen examples of three meta-functions: *value*, *storedBy*, and *storedAt*. These meta-functions are available to all function-objects, and thus are defined in the class *Function.* Functions in the class *Enumerated* get their values explicitly; that is, objects in this class can also respond to the assignment message "<-". Since methods get their values procedurally, function-objects in the class *Method* cannot be assigned to; however, they are able to respond to meta-functions that access their parameters and procedure body.

Note that since assignment is a meta-function, it must be used on function-objects only. For example, to change the sales of *gm* we say "*gm :sales <- 61*". Thus, the assignment meta-function can be thought of as instructing its intension to change its value to another object. This is an interesting and elegant treatment of assignment, that avoids the need to introduce concepts such as location or l-value into the language.

TimeSeries functions are enumerated functions that can hold a series of values indexed by date. Thus they are used to model historical information. TimeSeries functions can respond to meta-functions such as *asOf:* and *asOf:Put:*, which access and modify this historical data. We will look at TimeSeries functions in more detail in Section 6.1.

The class of versioned functions is analogous to TimeSeries. This class defines meta-functions that create new versions, access or modify the value in a particular version, and so on. Section 6.2 discusses versioning in more detail.

The class of a function is defined in the database scheme. For example, the function *sales* in Figure 1 might be declared as a *TimeSeries* function. As a result, the function-objects *gm :sales*, *ford :sales*, etc., are all instances of the class *TimeSeries*.

When we defined functions back in Section 4, we mentioned that there is no

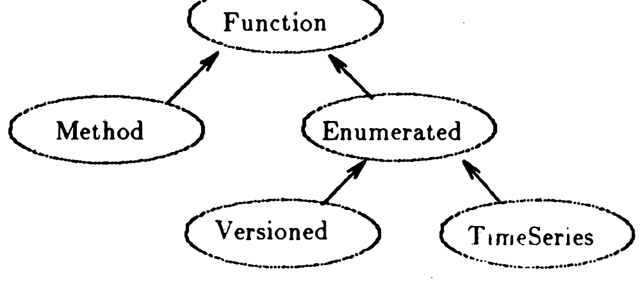

Figure 3. Part of The Function Hierarchy

distinction made between instance variables (i.e. enumerated functions) and methods. At this point, we can now see exactly what is going on. As long as a query only uses function extensions, there can be no difference between any type of function. For example, in the query "*gm ceo birthday*", it is absolutely irrelevant how the functions are defined (including the function *gm*!). The distinction becomes apparent and necessary only at the intensional level. The reason should be clear: If we want to assign a new value to a function, we need to make sure that the function is enumerated; if we want to look at a different version of the function, we need to know that it is versioned.

6. Contexts

In our previous examples, a function such as *sales* maps a company object to a number. That is, it returns a single value. However, the function might contain a lot more information than this. For example, if it were a *TimeSeries* function, then it would encode several values, indexed by date. Our principle is that the type of a function should not be visible except intensionally; thus, the only possible result of the expression "*gm sales*" is to have the system choose a default date, and return the value for that date. Informally speaking, this default value can be thought of as the *context* in which the expression is evaluated.

There is an analogy here to natural language: An object calling a function corresponds to the subject and verb respectively in a sentence, and the function's parameters correspond to the objects of the verb. Natural language has a flexibility that is missing from computer languages, in that it can rely on context to supply additional meaning. Typical mechanisms for supplying the context for an action are the global state built as part of a dialog and optional phrases used to qualify that action. Both mechanisms are appropriate for inclusion in an object-oriented language.

The easiest way to understand how context works in VISION is to examine some examples.

6.1. Object History

An important use of intensions is in modelling historical data. Consider again the function *sales*, which maps a company to a sales value. By declaring this function to be of type *TimeSeries*, we enable it to contain values for any set of dates. These values are accessed via meta-functions, two of the most important being *asOf:* and *asOf:Put:*. Figure 4 gives a scenario using these meta-functions.

The function type *TimeSeries* is called a *context dimension*, because it allows each function to have a (one dimensional) range of values associated with it. When a particular function-object is being executed, the system will determine which of the dates the user is intending. This date is called the *context* of the function. So in the query "*gm :sales asOf:1984*", the function is being executed in the context *1984*.

Context can change during the execution of the query. Initially, there is a default context, which typically is the date *now*. In the query "*gm :sales asOf:1984*", the meta-function *asOf:* changes the context to *1984*, returns the value of *gm sales* in the new context, and restores the previous context. The function *evaluate:* can be used to change the context for an entire block. For example, in "*1984 evaluate: [gm sales / gm earnings]*", the functions *sales* and *earnings* are both evaluated in the new context *1984*.

Context changes can be nested. In the query "*1984 evaluate:[gm :sales asOf:1985 > gm sales]*" the meta-function *asOf* temporarily rebinds the context within the block; the query thus compares the sales of *gm* in 1985 and 1984. In general, context behaves in a stack-based way.

When a context is not specified, meta-functions rely on the most current one. In the query "*gm :sales value*" (which is equivalent to "*gm sales*"), the meta-function *value* returns the value of the function in the current environment Thus, the query "*1983 evaluate: [gm :sales value]*" returns the value for 1983, since that is the current context of the block. Assignment operates similarly. In the query "*gm :sales <- 52*", the value of *sales* is changed for the current context. So for example if the current context is *now*, then this

```
        gm :sales asOf:1984 Put: 60;
        gm :sales asOf:1988 Put: 80;
        gm :sales asOf:1985 print; gm sales print
Output: 60  80

        1986 evaluate:[gm :sales <- 99];
        gm :sales asOf:1987 print
Output: 99
```

Figure 4. A TimeSeries Example

```
        ibm :sales inVersion:v1 Put:20 ;
        :v2 <- v1 newVersion ;
        gm :sales inVersion:v2 Put:30 ;
        :v3 <- v2 newVersion ;
        ibm :sales inVersion:v3 Put:40 ;
        v2 evaluate: [gm sales print; ibm sales print;
                ibm :sales inVersion:v3 print ]
Output:  30 20 40
```

Figure 5. A Versioning Example

query is equivalent to "*gm :sales asOf:now put:52*".

6.2. Object Versions

A function may have several different versions stored in the system. We can talk about the current version or previous versions; we can create a new version from an existing one; we can compare the values of two versions; and so on. As with object history, versioning provides a context dimension. A versioned function contains values indexed on version names. The analogous meta-functions are *inVersion:* and *inVersion:Put:*. We also need the function *newVersion*, which creates a new version name from its parent version. Figure 5 presents an example scenario, assuming that *sales* was declared to be of type *Versioned.*

The execution of a *Versioned* function requires a context. This context can be specified explicitly using the appropriate meta-functions, or it can be unspecified, in which case the current version is used. The mechanism is exactly the same as with *TimeSeries* functions. In particular, the function *evaluate:* can be defined for version names just as it was defined for dates.

This model of versioning is very flexible, and provides more than is commonly used. For example, a typical practice is to freeze the current version of the database, and start up a new child version. Changes to the database will then affect only the new version. Old versions can be unfrozen if desired. This paradigm can be modelled using the only functions *evaluate:*, *newVersion*, *makeActive*, and <-. Figure 6 gives a scenario where version *v1* is frozen and later unfrozen

It is interesting to examine the differences between versioning and object history. Syntactically, they are quite similar. Both provide a context dimension with analogous meta-functions. A big difference is what they are indexed by. Histories are indexed by date, which has a linear order known to the system. When we insert a new data point, we do not have to specify where it goes in the series. With versions, however, we must specify the relationship between each version name. As a consequence of this, versioning is more flexible; in fact, version names can form a tree.

There is also a semantic difference between the two context dimensions. Object histories are used to model some series of events in the world. Versioning is used to model the events involving changes to the database. These two dimensions are orthogonal, and thus it is perfectly useful to have a function which is both versioned and historical. We will examine this situation more in Section 6.5.

6.3. Advantages of the Context Paradigm

In the previous two sections, we have tried to show how contexts provide a different yet powerful perspective on the old problems of modelling historical and versioned data. In this section, we will attempt to drive home the point by comparing this paradigm with more traditional approaches.

Consider the problem of modelling the historical function *sales* in an object-oriented language without contexts or meta-functions. The traditional approach is to encapsulate the semantics of time at the class level. That is,

```
ibm :sales <- 30 ;                      -- current version is v1
:v2 <- v1 newVersion: ;
v2 makeActive ;                         -- v1 is frozen, v2 becomes active
ibm :sales <- 5000 ;                    -- change version v2
v1 evaluate: [ ibm sales <- 35 ]        -- v1 is unfrozen and changed
```

Figure 6. Freezing and Unfreezing a Version

the query *"gm sales"* must yield an object of type *timeSeries*, which probably is some sort of dictionary indexed by date. The function *asOf:* can then be used to extract the appropriate value from the dictionary; for example, *"gm sales asOf:1984"*. So in order to get the current sales, we must say *"gm sales asOf:now"*. And since this is inconvenient, we also will define a method *currentSales* that abstracts away the date calculation.

We see two basic problems with this approach. First, there is an unstructured proliferation of functions. For each historical function *f*, we have another function *current-f*. For each versioned function *g*, there will be the function *active-g*. What if a function *h* is of both types? Then we will need yet another function *current-and-active-h*. Clearly it is possible to do this, but rather cumbersome and inelegant. Note that the single concept of *sales* gets split up into the syntactically unrelated functions *sales*, *currentSales*, *activeSales*, *current-and-activeSales*, and so on. Not only is there no good way to recognize the set of functions for each concept, there is also no way to see which conceptual operations in a class are implemented similarly.

The other problem is perhaps more serious. Since such a system does not automatically manage contexts, there is really no way to write a function such as *evaluate*, or to have nested context changes. Thus much of the expressive power of VISION is missing.

6.4. Keeping Context Unbound

Typically, a program is interested in the value of a function in a particular context. However, it is also useful to be able to examine all of the values of the function in a particular context dimension. For example, one could ask for the maximum sales value over a period of time, or execute a curve-fitting algorithm over the accumulated sales data.

A function intension with an unbound context dimension can be thought of as a collection containing all of the function's possible values. Consider again *sales, and assume that it is a TimeSeries* function. Then the collection *"gm :sales"* contains several sales values. For an example, the query *"(gm :sales select:[^self > 50]) count"* returns the number of dates in which sales were sufficiently high.

When we treat a *TimeSeries* intension as a collection, each of its elements has a different time context. A user query may take advantage of the time context in the collection, or it may choose to ignore it. For example, the above query ignored the time component of the values. On the other hand, a query that performs regression analysis or curve fitting will depend on knowing the time context of those elements.

6.5. Multiple Contexts

The results we have described so far have depended on a function having only one context dimension. We now want to describe how things change when several context dimensions are present. Consider the meta-function *value*. We generalize it so that it binds all unbound contexts to default values when it is executed. For example, suppose that *sales* is both a timeSeries and versioned function. Then "gm :sales value" binds both contexts to defaults, returning the sales value in the current version and the current time. Since the expression *"gm :sales asOf:1980"* is equivalent to *"gm :sales :asOf:1980 value"*, it explicitly binds the time context, and uses the default version context. Similarly, the expression *"gm :sales inVersion:v1"* explicitly binds the version context and uses the current time.

It is also to bind both contexts explicitly. Note that the expression "*gm :sales :asOf: 1980*" returns an intension in which the time context is bound, but the version context is unbound. Thus "*(gm :sales :asOf: 1980) inVersion: xyz*" explicitly binds both contexts. This last expression is equivalent to "*(gm :sales :inVersion: xyz) asOf: 1980*", since time and versioning are symmetric.

In the previous section, we showed how the values in a single context dimension could be enumerated. There are some issues involved in extending this to several context dimensions. For example, consider the intension "*gm :sales*", in which two contexts are unbound. When we enumerate, as in "*gm :sales maximum*", which dimension has priority? In this example (as well as most financial examples), it is more natural to be aggregating over time. Thus "*gm :sales maximum*" should return the largest historical value of *gm*'s sales in the current version. In other cases (e.g. in CAD or CASE applications), it would be more natural to aggregate over versions. Thus there is an asymmetry that needs to be specified somewhere.

The specification of which context has priority is really a policy decision. The goal of this conceptual model is not to preordain a particular policy for resolving ambiguities among multiple contexts. Rather, we want to provide a flexible framework in which to implement alternative policies. Clearly, the most important decisions are made in the implementation of the "*value*" function. Other policy decisions can be encapsulated by the type as well. For example, the intension produced by "*gm :sales*" represents a first class object. The system need not restrict the type of an intension to a specific set of types. As a result, a type designer can define different types that implement different ambiguity resolution policies.

6.6. Concurrency Control and Contexts

The bulk of an investment database consists of historical and meta-data. Most uses of this data require the provision of a stable version over a long time period. Many transactions are read-only. Under these circumstances, an optimistic multiversion concurrency control strategy is especially appropriate, since read-only transactions always commit without any locking or verification overhead. Some data such as real-time market and news-wire data, however, is inappropriately managed using the stable version semantics of multi-version concurrency control.

Most database systems impose a single, system-wide concurrency control policy on an entire database. As noted, a single policy is not appropriate. Just as with multiple context dimensions, what is needed is a framework for supporting policy implementation without imposing policy. Because functions encapsulate update, they provide a natural home for encapsulating multiple concurrency control policies as well.

Concurrency control policies must deal with the issues of *visibility*, *validation*, and *integration*. *Visibility* deals with the accessibility of updates to other concurrent transactions. *Validation* deals with the serializability of a transaction attempting to commit. *Integration* deals with the creation of persistent structures to reflect the updates made by a transaction. We consider the issues of *visibility* and *validation* briefly below. Issues related to *integration*, along with a discussion of efficient strategies for implementing multiple visibility and validation policies, are not considered here because they require a understanding of a storage architecture that is beyond the scope of this paper.

The issues related to visibility are largely encapsulated in the function type's implementation of operations like <- and *value*. The system provides one service in support of type specific policy implementation. Each transaction creates a new version object derived from some pre-existing version object. That new version object becomes the default version for the transaction. Functions such as *sales* are of a type whose implementation supports versioning and disallows modifications to committed versions of the data base. As a result, transactions evaluating these functions see a stable view of the data base. Functions that support real-time data are generally not versioned. As a result, updates to these functions are immediately accessible to other transactions. Clearly, policies that synchronize transactions based on 2-phase or other locking protocols are also possible within this framework.

The issues related to validation are largely encapsulated in the function type's implementation of the *newVersion: version1 serializableWith: version2* operation. That operation has a straightforward implementation for the two function types discussed in this section. For versioned functions implementing an optimistic, multi-versioning concurrency control policy, each function instance maintains an indication of whether its *version1* value was accessed and *version2* value was modified by this transaction. For function types used to encode real-time relationships, this operation is effectively a no-op. Once again, other policies are conceivable and implementable within this framework.

7. Summary

The use of meta-functions provides an incredible amount of expressive power to a data language. In this paper, we have shown how they can be used to express the notions of time, versions, updates, and concurrency control. As an abstraction mechanism, they allow type specifications to clearly distinguish between structure and usage. The factoring produces compact, comprehensible type specifications capable of robust, extensible interactions with their environment. However, it seems like we have only scratched the surface of this abstraction mechanism. For example, one area under active examination is the application of these mechanisms to the problem of algebraic optimization of object-oriented programs. In practice, first-class functions, contexts, and meta-functions have proven to be powerful abstraction tools.

The notion of meta-message is fundamental to the VISION language interpreter. TimeSeries functions have been implemented, and appear to have fulfilled their promise. We currently are working towards solidifying the versioning and concurrency control ideas expressed here with an eye towards implementation.

8. References

[BP] Bracci and Pernici, "A conceptual Model for Office Information Systems". *Proc. ACM SIGMOD Conference on Business and Office Applications* 1983, pp 108-116.

[CS] Caruso and Sciore, "The VISION Object-Oriented Database System". *Proc. International Workshop on Database Programming Languages*, Roscoff France, 1987.

[CT] Clifford and Tansel, "On An Algebra For Historical Relational Databases: Two Views". ACM SIGMOD Conference, 1985, pp 247-265.

[GR] Goldberg and Robson, *Smalltalk-80: The Language and its Implementation". Addison-Wesley, 1983.*

[KM] King and McLeod, "Semantic Data Models". In Yao, ed., *Principles of Database Design*, 1985, pp 115-150.

[KW] Kemper and Wallrath, "An Analysis of Geometric Modelling in Database Systems". *ACM Computing Surveys*, March 1987, pp47-91.

[MSOP] Maier, Stein, Otis and Purdy, "Development of an Object-Oriented DBMS". *ACM OOPSLA Conference* 1986, pp 472-482.

[OBS] O'Brien, Bullis, and Schaffert, "Persistent and Shared Objects in Trellis/Owl". *Proc. IEEE OODB Conference* 1986, pp 113-123.

[PL] Powell and Linton, "Database Support for Programming Environments". *Proc. ACM SIGMOD Conference on Engineering Applications* 1983, pp 63-72.

[Sc] Sciore, "Object Specialization". TR 87-007 Boston University, 1987.

[Sh] Shipman, "The Functional Data Model and the Data Language DAPLEX". *ACM TODS*, March 1981, pp 140-173.

[V] "The VISION User's Manual". Innovative Systems Techniques, 1987.