ANOTHER LOOK AT DATA-BASES*

Naftaly Minsky
Dept. Of Computer Science
Hill Center, Busch Campus
Rutgers University
New Brunswick, N.J. 08903

1. What is a Data Base?

A data base is often viewed as a kind of glorified file which allows for the storage of complex data strucutures, and which can be shared by many users who have different needs. But there is a more fundamental difference between data-bases and files then just the complexity of data or the generality of its retrieval. The traditional file is essentially a passive collection of coded data which is devoid of any intrinsic meaning. Indeed the interpretation of the data stored in a file is entirely up to its users. The data-base, on the other hand, is usually used as a model, or representation. of some system in the real world. For example, a corporate data base is, in some sense, a model of a corporation; and it must reflect some of its structual and behavorial properties. This is not just a matter of recording correctly everything which happens within the corporation. For example, when we "tell" the DB that an employee is fired, we expect all necessary adjustments to be made automatically, so that only current employees will be on the payroll of the corporation at every moment in time. In short, we expect a DB to "take care of itself" to a certain extent; it therefore cannot be a completely passive collection of data. But if a data-base may contain procedural components and not just data, what kind of thing is it? In what way is it different from what we usually call programs? (It might have never occurred to the reader to compare data-bases with programs, but as we will show later the two concepts have more in common than meets the eye.) In an attempt to clarify the concept of data-base I will now offer a definition for it. The definition is an oversimplification, as any definition of such a fuzzy concept must be; it will emphasize certain characteristics of data-bases, at the expense of other, equally important, properties.

Definition: A data-base is a model (or representation) of some system in the real world, which satisfies all of the following properties:

(a) The model contains a large amount of coded information.
(b) It has a long life time (say, from several days to several years).
(c) The model can be examined and interrogated at any moment of its life time.
(d) The model changes primarily in response to operations applied to it from the outside.

*This work was supported by Grant # DAHCIS-73-G6 of the Advanced Research Projects Agency.

The first sentence of this definition states the general objective of a data base. This objective is not specific to data bases, however. For example, a corporation can be modeled by a simulation program, not just by a DB. The specificity of this definition to data-bases is provided by properties (a) to (d), the first of which we assume to be tacitly understood.

To see how the other properties relate specifically to data-bases consider again a corporate DB. Such a DB functions as a model, or a representation, of a corporation for long periods of time. At every moment during this time the model can be interrogated, which corresponds to retrievals of information from the DB. Note that this is quite different from the behavior and usage of traditional programs.

A program is used primarily as a kind of input-output engine; it is executed for its output, and it is not usually examined while in the process of computation, Indeed, the active period of a program, what may be called its life time, is relatively short and insignificant.

Another difference between data-bases and programs is provided by property (d) which states that a DB changes primarily in response to operations from outside, update operations if you will. This is in sharp contrast with, say, a simulation model of a system whose dynamic behavior is determined primarily by the model itself. This property also introduces a new dimension into the sufficiently difficult problem of correctness in programming systems. When we write a program we have to make sure that the program will do exactly what we want it to do, which is difficult enough. But the designer of a DB has an additional problem at hand. He does not know in advance how the DB will change in time; this is up to the users who will interact with it. At the same time, it is up to the DB designer to establish the invariance of certain structural and behavioral characteristics of the DB whatever the users of the DB might do. In other words, the *integrity* of the DB must be protected against the essentially unpredictable interaction with the outside world. (Of course, this problem is not peculiar to data-bases, it exists to a certain degree in many programming systems, particularly in operating systems; but the need for protection is particularly acute in the case of data- bases.)

Most of the current research on data-bases is directed towards various implications of properties (a) to (d). People are most concerned with subjects like the physical and logical representation of complex data, techniques and languages which allow flexible retrieval from the data-base, etc. But it seems that the objective of data bases, to be a model of a system in the real world, is somewhat discarded, or at least, is not sufficiently emphasized. True, the protection of the integrity of data bases is often stated as one of the main objectives of data base systems. But in most cases this is hardly more than lip service, supported, at best, by simple consistency checking. But if there is a lesson to be

learned from the long experience with programming in general, and with operating systems in particular, it is that protection is not an add-on feature but must be designed deeply into the fundamental structure of a system. This, to the best of my knowledge, is usually not done in the case of data-bases.

In this paper we will examine some aspects of the architecture of data-bases having in mind primarily the need to protect their integrity with respect to the real-life system which they are supposed to represent. The conclusions of this paper are not new, most of them are borrowed from ideas and principles developed in the context of operating systems and programming languages. If there is any originality in this paper it is the application of these well known principles to data-bases.

To remove any possible misunderstanding it should be pointed out that we are dealing here with the concept of data-base not data-base management system (DBMS). The relationship between these two concepts is somewhat analogous to the relationship between a program and its compiler.

2. A Data-Base-Language

It is well known that the structure of programs is strongly affected by the properties of the programming language in which they are written. A similar statement is probably true for data bases as well. Most of this paper will, accordingly, be concerned with the language used for the construction of data-bases. We will begin by reviewing the language proposed by one of the most influential works on data bases, the Data Base Task Group (DBTG) report [1].

Two languages are introduced by the DBTG report: The Data Definition Language (DDL), which is a declarative, non-procedural language, to be used for the definition of a DB; and the Data Manipulation Language (DML) to be used for interaction with an existing DB. This dichotomy of languages, which is somewhat analogous to the sharp distinction made in COBOL between the "data division" and "procedure division", seems to imply that data-bases are essentially passive structures, since they are to be defined by a language which does not feature any data-manipulation capabilities. This approach certainly does not agree with our view of a data-base as a model of a possibly dynamic system. Moreover, it is not even compatible with the modern approach to data structures in general, as we will see later. Indeed, the DBTG report did not really view data-bases as completely passive structures. To compensate for the lack of procedural capabilities in its DDL, it allows for the incorporation of the so called *DB-procedures* within the data base. But the report did not specify how such procedures should be incorporated into a DB, or in which language they should be written (should it be a DML)? These questions are generously left for the implementor, in spite of the fact that the declarative features of the DDL were described in great detail. It appears that the authors of the DBTG report considered the DB-procedure just as an additive feature of marginal importance.

It is my contention, however, that the language to be used for the design of data-bases must be an integration of declarative and procedural capabilities, as most programming languages are; one

may call such a language a Data-Base-Language (DBL). An obvious reason for having such a language is that if procedures are at all necessary for the design of data-bases, there seems to be no good reason to introduce them via the back door, as is done by the DBTG report. But there is a more subtle reason for that. The DB-procedures are the dynamic components of a DB. If we want to have any control over the dynamic behavior of the data base we must be able to control the DB-procedures themselves. In particular, one should be able to restrict a given procedure as to what it can do to the rest of the DB. It appears that the clearest way to achieve that, is to have all DB-procedures be written in a given, well-defined, DBL. The manner in which such a language can be used in order to impose certain disciplines on the behavior of a DB, will be discussed in the rest of this paper.

3. "Information Objects" in Data Bases".

What was said about the data-base being a model of some system in the real world should be true also for the individual components, or records, stored in a DB. At least some of them must represent specific objects in the real world. For example, we might have a data structure which represents an employee in a corporation, and another one which represents a job. The behavior of these data structures within the DB must reflect, in some way, the behavior of the objects they represent within the corporation itself. For instance, one should be able to hire or fire an employee, or to assign him to a certain job; but one should not be able to manipulate arbitrarily a data-structure which represents an employee, or a job. Thus, it seems that we need the ability to form data structures which cen be manipulated only in a certain predefined way. Data structures which have this property will be called here *information objects*.

Note, however, that being an information object is not really a property of the data structure itself, but of the "computing environment" in which it exists, which should prevent any illegal manipulation of the object. Such a computing environment can be created by a programming language, as was demonstrated by a number of researchers.

In a recent paper by Liskov and Zilles [3], a language is described which features a linguistic contruct called *cluster* which serves as a template for a class of information objects. The cluster contains a description of a conventional data-structure, together with a set of procedures which are defined on that structure. For a given cluster c, there is a way to generate objects using c as a template. All such objects are said to belong to class C. Now, the crux of the matter is that an object of class C can be manipulated only by means of operators defined within the cluster, which turns these objects into information objects, according to our definition.

Note that this is not an easy proposition; it is not just a matter of adding yet another feature to an existing language. The entire language must be very carefully designed in order to guarantee that

an object of class C is never manipulated by an operator which does not belong to it. Note also that the combination of both declarative and procedural statements in the same cluster is necessary for the formation of these information objects.

We will now try to demonstrate, by an example, the potential usefulness of information objects for data-bases, assuming that this facility is supported by the DBL at hand.

Example: Conservation of Money

One type of object which has to be represented in any financially oriented DB, is money. One of the most important properties of real money is that it cannot be created out of nowhere, at least not in a legally operated establishment, nor should it be allowed to disappear into thin air. This property may be called *conservation of money* and it should be reflected in the behavior of the data-structure which represents money in a DB.

To impose conservation of money on a DB, we will represent money by means of information objects to be called *safes*. A safe is a kind of money-variable which can be accessed and manipulated only* by means of the following operators:

Content(s)	returns the number of dollars currently stored in safe s.
move(s_1,s_2,k)	if s_1,s_2 are safes and k is a positive number such that *content*(s_1) $\geq k$, then this operator "transfers" units of money (dollars) from s_1 to s_2. (Namely, it leaves the content of s_1 smaller by k, and that of s_2 larger by k).

The operator *move* represents the act of money changing hands in the real world. In order to represent the flow of money into the system we use another class of information objects, to be called *source*. A source is similar to a safe except that any amount can be moved from a source into a sink, but no money can be moved into a source. The content of a source would be negative, its absolute value being equal to the total amount of money moved out of it.

Under these conditions, and assuming that the content of all sources and safes is initially zero, it is easy to see that the sum of all safes and sources in the system is always zero.

In spite of the obvious usefulness of being able to impose such a law of conservation on a DB, it is not quite sufficient. For example, one would like to impose certain disciplines on how the money flows between the various safes, and how it is used within the DB. We will return to these problems in the next section.

*Except for operators which generate and destroy safes, which are not discussed here.

4. Environments of Execution

As was just demonstrated, the ability to create information objects which can be manipulated only by means of given operators can help in enforcing useful disciplines on the behavior of a DB, but this ability alone is certainly not sufficient. One should also be able to specify "who" can apply which operator to a given object. Such a capability is fundamental to the various protection schemes recently developed for operating systems (2), it also forms the basis for many of the techniques used for the protection of data-bases against users who interact with it. For example, according to the DBTG proposal, a DB-user operates within an "environment" formed by a *sub-schema* which specifies, in effect, which parts of the DB the user is allowed to get, and how he is allowed to access them. But although this does provide a DB with certain protection against its users, it does not protect it against itself.* A DB is manipulated not only by its users, but by its own procedures as well; and it is a well known principle in the design of large systems that no module should be allowed to access more of the system than it needs, in order to perform its function. Thus, the DBTG proposal should have assigned a sub-schema to every DB-procedure, in order to limit the procedure as to what it can do to the rest of the DB. This kind of protection scheme was proposed by Wulf, Jones and others (2), for the design of operating systems. Although their techniques must be modified before they can be applied effectively to data bases, the basic philosophy has universal validity, and is briefly described below with some changes, and slightly different terminology.

According to Jones' and Wulf's proposal, every *activity* (or what is usually called "process of computation") in a system is carried out from a certain *environment* E, which can be formally described as a set of pairs $E = (q,a)$, where q is the identifier of an *information object* in the system, and a is one of the *access-modes* defined for it. The pair (q,a) is called *capability* (or *right*). A capability $(q,a) \in E$ serves as a right for a process which operates under E to exercise access a to object q, in the following sense. Every operator defined as an information object in the system, expects the environment from which it was invoked to have certain capabilities. Namely, the operation is carried out only if the environment in which the requesting activity resides contains the required capabilities. An example may clarify this point.

Consider again our money example. Let *get* and *put* be two access modes defined over a safe such that the operator *move* (s_1,s_2,k) can only be carried out if the calling environment has the capabilities (s_1,get) and (s_2,put). This allows us to specify not only which safes a given user can access, but the direction in which he can move money between them.

One of the most interesting aspects of Jones' model is its treatment

*It should be pointed out that even in this respect the sub-schema of the DBTG report has serious deficiencies, as is shown in (4).

of procedures: Every procedure is defined within its own environment, which obviously does not depend on the environment of the potential caller of the procedure. The capabilities in this environment are called the *static* capabilities of the procedure. In addition, when the procedure is called, it may receive some additional, *dynamic* capabilities, from the caller's environment, by means of its parameters.

It is quite clear that such protection schemes can greatly enhance the reliability of a large programming system. But I would like to demonstrate specifically the usefulness of such a scheme in the context of data bases, using for that, our money example.

In the previous section we saw how to establish a global "conservation of money" in a corporate DB. But more than that is required in order to model the behavior of money in the real world. For example, assuming that paychecks are issued under the direct supervision of the corporate DB, one may require that whenever a check is issued on the amount of D dollars, the total amount of money represented in the DB is reduced by D. The implemention of this and other characteristic properties of money are discussed below.

Consider a DB-procedure PAY*(e,s)* which is designed to issue a check payable to employee e for the amount contained in safe s. The following assumptions are made.

a) PAY has a static and exclusive *put* right to a safe called *cash*, (namely, it has the capability *(cash, put)* in its environment).

b) No environment in the DB has a *get* right to *cash*. (The safe *cash* is, therefore, a *sink* of money).

c) PAY has a dynamic *get* right to the safe s given to it as a parameter, and it has no other right to any other safe.

d) PAY is the only procedure which is able to actually print a check.

Now, if the procedure PAY is known to perform the operation *move (s,cash,content(s))* after printing a check for the amount of *content(s)*, then we are assured that every check printed by the DB is balanced by a suitable amount of dollars being transferred to the safe *cash*. Moreover, *content(cash)* is equal to the total amount of dollars which was paid by checks.

This example can be further refined to establish *local conservation of money*, as follows: consider an environment E, which may be the environment of some user, or of a DB-procedure. Suppose that E contains the *get* right to a single safe s_0, and both the *get* and *put* rights to the set of safes $s_1,..s_n$. Suppose also that no other environment has the *put* right to $s_1...s_n$, and that E has the right to invoke the PAY procedure. E may, for example, be the environment used by the clerks in the accounting office of a certain

department in the corporation. It is clear that whoever operates under E, has only as much money available to him as he can get from s_0. which is his only external source of money. For example he can never pay more in checks than what is given to him via s_0. Thus, s_0 can be used by an environment which happens to have the *put* right to it, as a way to allocate budget for E. Similarly, E may grant some of his own money to some other environment via one of the safes s_i.

It is clear that by these kind of restrictions on the manipulation of safes one can model many of the properties associated with money in the real world. A monetary information system built in such a way also lends itself very nicely to auditing. The auditor can know much about the flow of money inside the corporation just by examining the various environments and some DB-procedures such as our PAY procedure.

The implementation of the protection scheme described above is beyond the scope of this paper. I wish only to suggest that it should be entrusted to the Data-Base-Language. This language has a unique position from which it can control what happens within the DB, because the DB itself is presumably defined in it. In this sense, the DBL will play the role of the protection *kernel* of operating systems proposed by Wulf and Jones. (For the reader who is puzzled by the idea that a language can be used for protection, I would like to point out that in a certain sense a programming language can be viewed as a set of capabilities which are provided to a program written in this language. This view about languages, and its implications, to the ability of a language to play a major role in protection is discussed in (4).

Conclusion

A DB has been viewed as a dynamic model of some given system, rather than as a passive collection of data. This paper was concerned primarily with some implications of this point of view to the language used for the construction of data-bases. One obvious implication is that a Data-Base-Language must have procedural capabilities. A somewhat more subtle observation made by this paper, is that a DBL should have certain features which would enable a DB designer to impose some discipline on the behavior of his DB, under its interaction with users. There are, however, other important issues concerning the Data-Base-Language, which were not discussed here. In particular, there is the problem of the *control-structure* of such a language. There are indications that the conventional control-structure of, say, ALGOL-like languages, would not be sufficient. For example, one needs tools to cope with the parallelism in the activity of data-bases. In addition non-conventional control structures such as "event-sequencing" (or "demons"), which were proposed for data-bases by Morgan (5), and which are being used in several AI-languages, appear to be very promising. These and other issues must be investigated in the context of the already existing knowledge about the structure and manipulation of complex data (1,5).

References

1. CODASYL Data Base Task Group (DBTG). April 71 report (available from ACM).

2. Wulf, W.A., et al. "HYDAR: the Kernel of Multiprocessor Operating System". Communication of the ACM, June 1974.

3. Liskov, B. and Zilles, S. "Programming with Abstract Data Types". Symposium on Very High Level Languages, March 1974.

4. Minsky, N. "Protection of Data-Bases, and the Process of User Data-Base Interaction". Rutgers University Technical Report TR-11, Sept. 1974.

5. Morgan, H.L. "An Interrupt Based Organization for Management Information System". Communications of the ACM, December, 1970.

6. Codd, E.F. "A Relational Model of Data for Large Shared Data Banks". CACM, June 1970.