/tex/reports.tex

Views, Objects, and Databases

Gio Wiederhold

Stanford University, Computer Science Dep.

October, 1986

Abstract
The major point of this note is that it appears to be unwise to store persistent objects, which are to be shared, in object format. There appears to be a major conflict between the database paradigm: making information available for sharing using non-procedural access, and the programming paradigm: devising data structures which are effective representations for procedural access. We illustrate the problem with a simple example taken from electronic design applications, where both sharing and object-oriented access is desirable.

We then suggest a procedural approach which permits object-oriented access to use information stored in a relational database. A implementation of the sketched approach will have severe performance penalties unless storage transformations are applied to the database and its interface optimized for object-oriented access. However, having a defined interface will permit strorage reconfiguration and access optimization, notions inherent in non-procedural data access. A direction for further development, leading towards data independence for object access functions, is indicated.

1. Introduction

The intention of this article is two-fold. First of all we show intrinsic differences in the underlying concepts of access to persistent storage in databases and current extensions of object-oriented programming systems intended to incorporate persistence [Asilomar 86] [Lochovsky 85]. Since the objectives of both paradigms are similar we develop a connection between *object* concepts in programming languages and *view* concepts in database systems.

Secondly, we propose an architecture which exploits their commonality, and indicate research and development directions which are needed to make the bridge viable. Such an architecture seems to be especially suitable for computer-aided design tasks.

Engineering Information Systems (EIS) and systems for similar applications must provide suitable abstractions of real-world objects for their users and support long-lived transactions [Katz 83]. The complexity of the design process, and the number of specialized individuals needed to bring major projects to completion is driving the search for more systematized solutions than those provided by the file mechanisms now in use in operational precursors of EISs [Lorie 83] [Udagawa 84]. The issues being raised here pertain to systems with large quantities of data, and long lifetimes. Design tasks which can be handled by single individuals are not likely to benefit from EIS technolgy.

Two figures on objects

Figure 1a: Objects.

Databases and Views

Database concepts provide independence of storage and user data formats. The schema describes the form of the database contents, so that a variety of users can satisfy their data requirements by querying the shared database using non-procedural declarations of the form:

```
SELECT what-I-want WHERE some-set-of-conditions-is-true
```

A database administrator integrates the requirements of diverse users. The shared database can be changed as long as the schema is changed to reflect such changes. Concepts of database normalization help avoid redundancy of storage and anomalies that are associated with updates of redundantly stored information.

The principal formal database mechanism to obtain selected data for an application is a view specification. A view on a database consists of a query which defines a suitably limited amount of data. A database administrator is expected to use predefined views as a management tool. Having predefined views simplifies the user's access and can also restrict the user from information which is to be protected. We are interested here only in the first objective, and not in the protection aspects associated with views.

Views have seen formal treatment in relational databases [SQL], although subset definitions without structural transformations are common in other commercial database systems [DBTG]. We will hence describe views from a relational point of view, without implying that a relational implementation is best for the underlying EIS databases.

A view is defined in a relational model as a query over the base relations, and perhaps also over other views. Current implementations do not materialize views, but transform user operations on views into operations over the base data. The final result is obtained by interpreting the combination of view definition and user query, using the description of the database stored in the database schema.

The view, as any relational query result, is a relation. However, even when the base database is fully normalized, say to Boyce-Codd normal form, the view relation is, in general, only in first normal form. Views are in that sense already closer to objects: related data has been brought together.

The issue that views present data not in second or third normal form seems ignored in database research, except for the update complications that result [Keller 85]. We have no evidence that the unnormalized views are uncomfortable to a user expecting a normalized relational representation. Acceptance of unnormalized views can be taken as a partial validation of the aceptability of structures more suitable to represent objects than normalized tables. Some current research is addressing unnormalized relations independent from the view aspect [Roth 84].

Objects

Object-oriented programming languages help to manage related data having complex structure by combining them into objects. An object instance is collection of data elements and operations which is considered an entity. Objects are typed, and the format and operations of an object instance are inherited from the object prototype.

The prototype description for the object type is predefined and the object instances are instantiated as needed for the particular problem. The object prototype provides then a meta description, similar to a schema provided for a database. That description is, however, fully accessible to the programmer. Internally, an object can have an arbitrary structure, and no user-visible join operations are required to bring data elements of one object instance together.

The object concept covers a range of approaches. One measure is the extent to which messages to external operation interfaces are used to provide access and manipulation functions. Objects may be active as in the ACTORS paradigm [Hewitt 77] or passive, as in CLU [Liskov77], or somewhere in between, as in SIMULA or SMALLTALK in terms of initiating actions.

The use of objects permits the programmer to manipulate data at a higher level of abstraction. Convincing arguments have been made for the use of object-oriented languages and some impressive demonstrations exist. Especially attractive is the display capability associated with objects as seen in THINGLAB [Borning 79]. Object concepts can of course be implemented using non-specialized languages, for instance in LISP [Moreau 85] or PROLOG [Mermet 85]. The tasks in EIS seem to match object-oriented concepts well and many experiments have been conducted [Afsarmanesh 85].

Objects and Databases

Let us assume a database is used for persistent storage of shared objects. A database query can obtain the information for an object instance, and an object-oriented programming language can instantiate that object. An interface is needed, since neither can perform the task independently. A view can define the information required for a collection of objects, but the data will not be arranged as expected for a collection of objects. Linkage to the object prototype and its operations is performed in the programming system. The program queries the database non-procedurally to remain unaffected by database changes made to satisfy other users.

The query needed to instantiate an object may seem quite complex to a programmer. A relation is a set of tuples, and, from an idealized point-of-view, each tuple provides the data for some object. However, normalization often requires that information concerning one object be distributed over multiple relations, and brought together as needed by join operations. The base relations must contain all the data required to construct the view tuples; the composition knowledge is encoded into the SELECT expressions used to construct the views. An ideal composition of an object should allow its data to be managed as a unit, but unless non-first-normal form relations — supporting repeating groups — are supported for views, multiple tuples are still required to represent one entity in a relational view [Stonebraker 83].

Hence storage of objects is not easy in databases, as indicated by the extensions proposed to INGRES for such tasks [Stonebraker 85]. A further complexity is that objects themselves may be composed from more primitive objects. In hierarchical databases records may be assembled from lower level tuples, but in relational databases the programmer has to provide join specifications externally to assemble more comprehensive relations from basic relations.

There is some hope that the formal techniques being developed for databases can permit the management of the information required to manage objects. Performance remains a bottleneck, and we will consider this issue later with our proposal. The database community has to be careful not to promise too much, too soon to the object-oriented folk.

Sharing Information in Objects

More serious are the problems we see in the management of shared information within the object-oriented paradigm. We consider two problems:

1 The growth of objects which contain information for multiple design tasks
2 The conflict when object configurations differ for successive design tasks.

Let us first consider the simpler case, where multiple users deal with the same configuration of objects. We will draw our examples from EIS, and consider that the objects are components of a circuit.

In using an EIS the level of abstraction for the objects changes during the process of design. First the object are simple logic elements, the process of design refines these objects to circuit components, and eventually to simple geometric elements projected from each layer of the chip [Spooner 86]. The final objects will carry many data elements not needed at the higher levels. The sketch in Figure 1 symbolizes how objects grow, and loose their vitality.

Two figures on sharing

Figure 1b: Growth of Objects As They Try To Satisfy Multiple Design Tasks.

As the design process moves from one subtask to the next successor objects are constructed out of earlier objects. Each new generation has new data fields appended.

Unfortunately, since design often requires cycles, old information cannot disappear. An unacceptable geometry may require a change at the circuit level, say adding an inverter to change a polarity. If design is iterative then successive transformations of objects must not lose information. This means that objects suitable for one task must contain information relevant to all tasks that may be successors, although much information may be irrelevant to the task at hand. The information may be hidden within the object, but must be passed on correctly, so it remains available when needed.

The objects become big, and no longer have the qualities associated with the object-oriented paradigm. A solution to this problem of overloaded objects is to have super-objects, owned by an object czar. Objects for each user task type are created by projection from the super-object. Updating privileges must be well defined.

This solution does not solve the second class of problems, namely sharing object information when the object configuration differs. Now, to create objects for a user task objects may have to be created from combinations of several, and perhaps different objects or super-objects.

Figure on complexity

Figure 1c: Complexity.

We show in the next section an EIS example having components and nets connecting. These present different configurations of the same information. In aircraft design the objects serve design tasks as aerodynamics, structures, power, fuel, control, etc., and it is obvious that their configuration is quite different. Even in a simple commercial credit card operation there is the customer as an object for some tasks and the establishments are the objects wanting to be paid in other tasks.

Because of these problems we present later a proposal which will not try to store to objects directly in persistent form. We prefer an new approach to satisfy the demands of database style sharing and object concepts.

2. Looking at some Examples

To clarify the introduction we will take a simple device, a D-type latched flip-flop. At some level of abstraction it is an object, at a lower level it is composed of several gate objects of only three types: `AND`, `INV`, `NOR`, and contacts to the external world. The graphical representation of Figure 2 shows the component objects of the flip-flop at the gate level and the interconnection nets. The components are capitalized and the nets have lowercase labels.

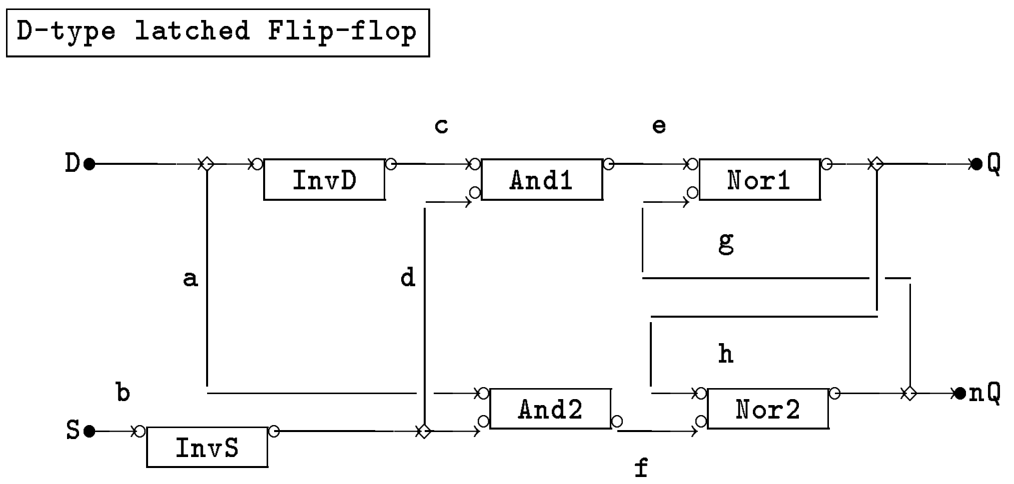

Figure 2: Latched D flip-flop [From Mukherjee 86]

A fully normalized database storing the information describing this circuit requires several relations. We show in Figure 3 the two library relations, which describe the types of gates (`Gates`) and their connection points (`Gate-connects`). Many other values will be kept in such a library. The ruling part or key attributes of each relation are placed ahead of a separator symbol :⟩ .

Relation **Gates:**

Gate-type :⟩	Function,	Area,	No-pins,	Power,	Delay,	. . . ;
Inv	$x = \not a$	30	2		. . .	
And	$x = a \wedge b$	40	3		. . .	
Nor	$x = a \not\vee b$	35	3		. . .	
Cntct	a	900	1		. . .	

Relation **Gate-connects:**

Gate-type,	C-no :⟩	C-id,	IO;
Inv	1	a	in
Inv	2	x	out
And	1	a	in
And	2	b	in
And	3	x	out

(more)

Gate-type,	C-no :⟩	C-id,	IO;
Nor	1	a	in
Nor	2	b	in
Nor	3	x	out
Cntct	1	a	inout

Figure 3: Relations describing the gates library.

Figure 4 presents a non-redundant representation for the specific circuit using two more relations, one for each gate instance and one giving the net connections for each gate. Other design-specific information can be kept within these relations.

Relation **Components:**

Id :⟩	Type,	Role,	Position,	. . . ;
D	Cntct	Data in	1/1	
S	Cntct	Sense in	3/1	
InvD	Inv	Data inverter	1/2	
InvS	Inv	Sense inverter	3/2	
And1	And	And data	1/3	
And2	And	And sense	3/3	
Nor1	Nor	Nor to Q	1/4	
Nor2	Nor	Nor to nQ	3/4	
Q	Cntct	State out	1/5	

Relation **Connections:**

Id,	Pin :⟩	net;
D	1	a
S	1	b
InvD	1	a
InvD	2	c
InvS	1	b
InvS	2	d
And1	1	c
And1	2	d
And1	3	e
And2	1	a
And2	2	d

(more)

Id,	Pin :⟩	net;
And2	3	f
Nor1	1	e
Nor1	2	g
Nor1	3	h
Nor2	1	h
Nor2	2	f
Nor2	3	g
Q	1	h
nQ	1	g

Figure 4: A Fully Normalized Description

The representation of the design shown is quite complete, but also fairly unclear. We need to create views which place all relevant data into coherent tuples. A view is needed to analyze the components and their loads; another view is needed to look at the nets, and other views will be needed for timing analysis, heat dissipation, etc.

In Figure 5 we extract two views, `ComponentsView` and `NetsView` for the database of Figure 4, using also the library relations shown in Figure 3. The `ComponentsView` is intended to be appropriate for checking components and their sources and sinks. It primarily accesses the `Components` relation, and joins with its tuples data about the connected components and from the libraries.

The `NetView` is intended to generate data on the interconnection nets for an application doing checking. The primary relation for the `NetView` view is the `Connections` relation, augmented with library information for the connected components.

The number of objects in `ComponentsView` is equal to the cardinality of the component relation (10), but the augmentation makes the result much larger (30). The eight nets are represented in the primary relation and in the view by twenty tuples, one per connected point ($\circ$) or external contact ($\bullet$). In both views we present the tuples in a logical order.

Assembling objects

Figure 1d: Creating Objects From A Database.

```
CREATE VIEW ComponentsView (Id, Pin, IdC :⟩ Type, N, IO, IOC)
     AS SELECT C.Id, C.Pin, CC.Id, CM.Type, T.No-pins, G.IO, GC.IO
     FROM Connections C CC, Components CM, Gate-connects G GC, Gates T,
     WHERE C.Id = CM.Id AND C.Net = CC.Net AND C.Type = T.Gate-type
     AND C.Type = G.Gate-type AND CC.Type = GC.Gate-type;
CREATE VIEW NetsView (Net, Dev, Pd :⟩ IOd)
     AS SELECT C.net, C.Id, C.Pin, GC.IO
     FROM Connections C, Components CM, Gate-connects G GC, Gates T,
     WHERE C.Id = CM.Id AND CM.Type = T.Gate-type;
```

ComponentsView :=

Id,	Pin,	IdC :⟩	Type,	P,	IO,	IOC;
D	1	InvD	Cntct	1	inout	in
D	1	And2	Cntct	1	inout	in
S	1	Inv1	Cntct	1	inout	in
InvD	1	D	Inv	2	in	inout
InvD	2	And1	Inv	2	out	in
InvS	1	S	Inv	2	in	inout
InvS	2	And1	Inv	2	out	in
InvS	2	And2	Inv	2	out	in
And1	1	InvD	And	3	in	out
And1	2	InvS	And	3	in	in
And1	2	And2	And	3	in	in
And1	3	Nor1	And	3	out	in
And2	1	D	And	3	in	inout
And2	1	InvD	And	3	in	out
And2	2	InvS	And	3	in	out
And2	2	And1	And	3	in	in
And2	3	Nor2	And	3	out	in
Nor1	1	And1	Nor	3	in	out
Nor1	2	Nor2	Nor	3	in	out
Nor1	2	nQ	Nor	3	in	inout
Nor1	3	Nor2	Nor	3	out	in
Nor1	3	Q	Nor	3	out	inout
Nor2	1	Nor1	Nor	3	in	out
Nor2	1	Q	Nor	3	in	inout
Nor2	2	And2	Nor	3	in	out
Nor2	3	Nor1	Nor	3	out	in
Nor2	3	nQ	Nor	3	out	inout
Q	1	Nor1	Cntct	1	inout	out
Q	1	Nor2	Cntct	1	inout	in
nQ	1	Nor2	Cntct	1	inout	out
nQ	1	Nor1	Cntct	1	inout	in

NetsView :=

Net,	Dev,	Pd :⟩	IOd;
a	D	1	inout
a	InvD	1	in
a	And2	1	in
b	S	1	inout
b	InvS	1	in
c	InvD	2	out
c	And1	1	in
d	InvS	2	out
d	And1	2	in
d	And2	2	in
e	And1	3	out
e	Nor1	1	in
f	And2	3	out
f	Nor2	2	in
g	Nor2	3	out
g	Nor1	2	in
g	nQ	1	inout
h	Nor1	3	out
h	Nor2	1	in
h	Q	3	inout

Figure 5: Component View and Net View

This view is still awkward, since component entities require multiple tuples. A more reasonable presentation would delete redundancies and add implicit non-first-normal-form bindings by rearranging columns according to the source relations. The `Nor1` component shown within Figure 5 would then have a object data structure as shown in Figure 6. We add a column `N` which gives the number of connected components. The computation of `N` using SQL requires `GROUP BY` and `COUNT` operations.

Id,	Type,	P,	Pin,	N,	IO,	IdC,	IOC;
Nor1	Nor	3	1	1	in	And1	out
			2	2	in	Nor2	out
						nQ	inout
			3	2	out	Nor2	in
						Q	inout

Figure 6: Reduced Datastructure for a View Object.

We now describe in Figure 7 this structure in the object-oriented extension of C, namely C++ [Stroustrup 86]. In C++ structures have to be mappable at compile-time, so arrays of dynamic extent cannot be embedded:

```
enum io { IN, OUT, INOUT };
 class Cpin;
class CCpin;
 class ComponentsView
{
    char      Id [8];
    char      Type [6];
    short     P;  //  Component pin count
    Cpin* Pin;
};
 class Cpin
{
    io        IO;
    short     Q;  // Connected component count
    CCpin* C;
};
 class CCpin
{
    char*   IdC;
    io      IOC;
};
```

Figure 7: C++ Datastructure for Component Objects.

Since it is permissible in C++ to have a dynamic array as the last element of a class by writing an appropriate constructor, it is possible to bring the `CCpin` array inboard.

Views and Objects

Let us now reconsider the similarities between views and object concepts. Both are intended to provide a better level of abstraction to the user, although the database user is seen to manipulate sets of objects in a non-procedural notation while the objects are manipulated procedurally and iteratively [SmSm 77].

The *collection* of tuples of a view is defined by the query which generates the set, and described by the relation-schema associated with the view query. The set of objects is defined by the user initiated action of generation and described by the prototype. Object-oriented languages may have a `ForAll` primitive to rapidly generate collections of objects of some type.

The description of the relation has to be available to the the relational user, since no implied operations can be kept in the relation-schema. There are proposals to store object defining procedures in relational schemas, but these have not yet been tested [Stonebraker 85].

Both tuples and objects can be selected based on any of multiple criteria, and interrogated to yield data for processing. View update may be severely restricted due to ambiguities in base relation update [Dayal 82]. Object update can be restricted by having only limited access functions, but otherwise no constraints are imposed on the programmer, although consistency problems can easily arise among users sharing objects.

Basic, of course, is the difference in persistence. Databases, and hence views over them, persist between program invocations. Objects must be explicitly written to files to gain persistence. Related to persistence is the critical issue to be addressed, namely the multiplicity of views that can be derived from a base relation. Figure 5 showed a view of `components` and a view of `nets` derived from the relations of Figure 3 and 4.

3. The Proposed Architectural Unit: View-Objects

We now propose to combine the concepts of views and objects, as discussed above, into a single concept: view-objects. This proposal is motivated by noting that systems, suitable for engineering information problem support require both sharability and complex abstract units, i.e., both views and objects. We use the concept of views to generate a working set of objects corresponding to projections of the entities described in the database [Carter 86]. The generators may need to access multiple relations to reconstruct the entities hidden to the relational representation.

Components of the architecture are:

1. A set of **base relations**.
2. A set of **view-object generators**.

In a complete version of this approach we also require the inverse function of 2., namely,

3. A set of **view-object decomposers** and archivers.

The conceptual base relations serve as the persistent database for applications under this architecture. They contain all the data needed to create any specified object type. Conventional database technology should be adequate for development, but eventually performance demands may drive users to specialized databases. We will consider the options for physical organization later.

Concurrent access by long-lived transaction will require a capability to conveniently access prior versions of the data. Where database management systems do not serve that need intrinsically the database must be configured with additional time-based attributes. As experience is gained this service will be included in specialized systems being built.

The view-object generator is the new component in this architecture. It performs the following steps:

a. The view-object generator extracts out of the base database data in relational form as needed — as now done by a query corresponding to a view. A view tuple or set of view tuples will contain projected data corresponding to an object. A view relation will cover a selected set of objects.
b. The view-object generator assembles the data into a set of objects. The objects will be made available to the program by attaching them to the predefined object prototype.

The view-object generator needs information other than the data, i.e., knowledge to perform its functions:

a1. The query portion identifies the base data.
a2. The specification of the primary relation identifies the object-entities.
a3. The object prototype specifies the structure and linkages of the data elements within an object, and the access functions for the object.

Initially the view-object generators will be implemented as code, closely following the translation programs in use now to convert persistent storage representations of engineering data files into representations suitable for specific tools. For experiments a relational database can be used for storage of the base data, but expect that ongoing developments in EIS will provide systems with more appropriate functionality and performance [IDA:86].

The goal is to eventually provide non-procedural access for object-oriented approaches as well. By formalizing the semantics of the objects required by the tools, and interpreting the object type description we believe that the view-object generators may be automated. The programmer then only provides the object type declaration and the `WHERE` clause defining the desired set of object instances.

With increased storage of data semantics in extended schemas one may advance further. With structural or dependency information about the base relations automatic generation of the internal structure of an object type may become feasible. Since access to the objects by the tools is still procedural, the performance benefits of automated object generation would be minor. The major benefit is in the control of access and consistency which may be gained [Neuman 82].

Justification

We are proposing a major extension to database and object-oriented concepts, with the intent to obtain the joint benefits that each approach provides separately today. The effort must justified by these benefits.

3.0.1 Sharable access to objects

The proposed architecture supports procedural access to objects, as expected by object-oriented programming systems. Access to the base data is automatic, and non-procedural. This permits base data to be effectively shared, since no single application or combination of applications imposes a structure on the base relations.

3.1 Growth of the system

New object types can be defined by adding new view-object generators. As is expected in databases, new data instances, types, and entire relations can be added without affecting other users' programs. Data can be reorganized without changing the object generator code since only the views will be involved. Such flexibility is essential for growth and multi-user access.

3.2 Support for a wide variety of representations

When simple, tabular results are to be obtained from the database, a view-object generator can easily generate tables for direct inspection and manipulation. A graphics object-generator can project the attributes needed for visual display.

3.3 High performance access from the database

To achieve this goal new database interfaces must be developed, which make the benefits of the set-oriented database retrieval concepts available to programming languages. Current database interfaces create a bottleneck when delivering data to programs. A common method for relational systems is to accept a query which specifies a set, but then require repeated invocations which deliver only single values or records at a time. This access mode, due to conventional programming techniques, is clearly inadequate.

The data from the databases is to be inserted into sets of objects at a time. For the object-oriented programmer such a set is best defined as a super-object. A program typically can not proceed until the set is complete. The object generators need only be invoked once for all the data needed to compose an super-object. An effective view-object generator hence needs an interface at a deeper level into the existing relational functions.

3.4 Updating from objects

The programs can freely update the contents of the objects. Some applications require that these changes be made persistent and hence be moved from its object representation to the base database.

To achieve update we envisage a third architectural component, the `view-update` generator. This component can be invoked at commit-time, when results effecting the objects are to be made persistent. We envisage that such a process will only update from objects which have changed, and only replace values which were changed.

Where views have been constructed using joins, aggregation operators and the like, automatic view update can be ambiguous. Restrictions on object update are one solution [Rossoupolous 86]. However, as shown by [Keller 86] such ambiguities can be enumerated and a choice can be made when the view-update is generated. The view update generator can also take advantage of the semantic knowledge available to an advanced view-object generator. An important source of knowledge constraining updates is authorization information.

4. Cost Tradeoffs

What costs and benefits will this architecture provide other than what we see as its structural advantage? We will now review areas where performance may be gained versus one of the two underlying approaches — database use or object-oriented programming — alone.

Set-based data access

A single invocation of a view-object generator will instantiate all specified objects. The overhead of programmed access to relations, typically requiring an intial `CALL` giving the query and then iterated `CALL`s to obtain the result piece by piece is avoided. Also no sequence of object instance generating calls, as seen in object-oriented programming is needed. Of course, one invocation of the object generator will be major operation, but only one call should be needed per object type.

The object generator may also perform the so-called macro-generator function, where instances of design objects are generated based on a general prototype and parameters. The source of such information is a library of cell descriptions, permitting, say, the generation of a series of cells making up a register.

The view-object generator will be more complex than either a database retrieval alone or an object prototype. It will provide a very clear definition of the mapping from base data to objects and do so in localized manner. A partial example was given, using the SQL approach to describe the view in Figure 5 and then using C++ to define the storage structure for the objects in Figure 7.

Binding of retrieved data

The ability of the view-object generator to bind the object will pay off in processing time. No joins are needed at execution to bind relational tuples, and no search operations are needed to assemble tuples belonging to the same object. Since the required information exists at view generation time no search cost is expended. The only requirement is that the object's internal data structure permits retention of the information.

Task allocation which matches hardware system components
Implicit, but not essential to our proposal, are the complementary concepts of database servers and design workstations. They will be linked by a powerful, but often still bandwidth limited communication network [Roussopoulos 86].

In such an architecture we expect that most of the retrieval and restriction operations will be carried out on the machine serving the database and the object generation and use will occur in the workstation.

Optimization in the file server The file server can select optimum retrieval strategies and reduce the data volume needed to convey the information. Keeping data in sorted order and removing redundant fields as shown in Figure 6 is straightforward and effective.

All operations are specified non-procedurally and can be rearranged and interleaved as needed to handle requests from the users. Concurrency control and version management are tasks best handled in the file server.

Communication minimization The data volume is reduced in the file server. Communication bandwidth between server and workstation can be used fully for information, rather than for data to be ignored in the workstation.

High performance on workstations The workstation only needs to assemble the information obtained into the desired object configuration.

It is desirable that all objects for an application task can be placed into the real memory of the working processor [Holland 85]. The objects now do not contain data fields irrelevant to the process at hand. Since these working objects are now compact, the probability that they will all fit into a modern workstation is increased.

Virtual memory management can be invoked if needed, but the capacities of modern machines are such that there should be adequate memory space for working storage of the required objects. However, we can envisage several techniques to exploit having the data independence provided by view generators to improve locality for virtual memory management.

5. Physical Organization of the Database

Performance issues are considered critical in Engineering Information Systems, and experiments with relational databases have not given us confidence that adequate performance is easy to obtain [Milton 83], [Narfelt 85]. Performance will furthermore be impacted by the extensions, such as version support, which are needed in the engineering environment [Katz 84]. Many of these extensions will also be useful for broader objectives, so that there is a motivation to share the development and maintenance costs of relational database systems extensions.

Objects are seen as a means to solve the performance problem, because data is bound in a user-sensible manner [Ketabchi 86]. We believe that such binding can indeed be significant in workstations, although our own experiments have not given proof of that assumption when external storage is used [Winslett 85]. There are obviously many more factors to be considered simultaneously when building comprehensive systems [Zara 85], but storing data in relational tables is often seen as a critical issue.

The fully normalized model of the representation has as its objective the minimization of redundancy and the avoidance of a number of anomalies that can occur. It also supports a very simpler storage structure, as seen in our example that can lead to a large number of relations. Retrieval from such an organization will require a large number of joins, as seen in the example leading to Figure 5.

Updates seem to require less work in a relational database than in our proposal. However, when the database has to obey interrelational contraints, expensive joins must also to be executed when the data are updated. For instance, it is neccessary to assure that the components exist and that the connections are valid when a net connection is to be changed.

There is, of course, no requirement for the physical storage structure to mimic literally the logical relational structure. Specifically, information from multiple tuples may be stored in one record. Denormalization has been formally considered [Schkolnick 80], but is not supported by current DBMS. It is commonly employed in practical relational databases [Hardwick 80]. Preserving correctness in a denormalized database requires often additional transformation and care: records may have repeated fields and at other times some fields may be replicated in multiple records. Sometimes here work is required as well: dependent tuples, as connection points for a component, will automatically be deleted if they were implemented as a repeating group. We expect to profit here from ongoing research on non-first-normal-form databases [Roth 84].

In our architecture, where we expect that the object generator is the primary means for accessing data, complexity issues discouraging use of a non-normalized storage structure are moot. The object generators can be adjusted, probably eventually automatically, to any storage structure chosen. It is likely that the most efficient storage structure will be similar to the dominant object structure.

With such an organization we have provided the two primary objectives favoring the concept of object-oriented programs for computer-aided design:

1 The view-objects provides the desired clean and compact user interface.

2 The flexibility of the storage structure can provide the locality needed to achieve high performance in source data access.

We can now provide these objectives for multiple users, and share selected information in a controlled fashion. We gain greatly from the flexibility obtained by interposing the object generator operating on a persistent base storage structure.

We are no longer bound to initial conceptulization of an the optimal data storage structure bound to an optimal object format. It is likely that in any design project the data retrieval loads will change over time. The object types which dominate use during initial design phases are not likely to be the object types used during the maintenance phase of the designed devices [Tichy 82]. A physical storage reorganization is now possible, as long as the view-object generators are adjusted correspondingly.

Replication, the primary tool to improve performance, can be utilized as well. A high demand subset of the database can be replicated and made available in a performance effective manner to the object generators. Object update functions can use primary copy token concepts [Minoura 82] to update all copies consistently and synchronously.

6. Conclusion

We argue that direct storage of objects, to make them persistent, is not appropriate for large, multi-user Engineering Information Systems. We propose storage using relational concepts, and an interface which generate objects. Initial versions of the interface may be coded directly, to test the validity of the concept. In the longer range we look towards knowledge-driven transformations.

Generation of objects from base data brings the advantages of sharing persistent information to an object-oriented program. It also provides a better interface from programs to databases, recognizing that access by programs, large or small, will remain essential for data analysis and complex transactions.

The separation of storage and working representation will also simplify the development of new approaches to engineering design [Brown 83], [Hartzband 85]. The object generator can be viewed as a system component utilizing knowledge to select and aggregate data for the information objectives.

Acknowledgement

This paper has benefited from interaction with a VHSIC program sponsored project to define information models for Engineering Information Systems, although it certainly does not reflect the consensus of the group. Comments by Tony Gadient of the USAF, WPAFB, Arnold Goldstein of MCC, and Prof. Alice Parker of USC were constructive and helped focus the issues. Earlier presentations of these concepts at Workshops for Integrated Design and Manufacturing Processes at the Naval Postgraduate School in Monterey CA, April 1986 and at the object-oriented Database Workshop engendered useful feedback. David Beech, from Hewlett-Packard, visiting the Stanford Center for Integrated Systems, provided motivation to consider the object-oriented approach and gave constructive criticism to earlier versions of this paper. He also provided the C++ example for Figure 7. Support for this research was provided in part by DARPA contract N39-84-C-0211 for Knowledge Based Management Systems and in part by the Air Force through a contract with UES F33615-82-C-1716.

References

We can cite here only a few of the many publications which have provided concepts leading to this paper. We include some recent global references to provide access to other work. A more complete list can be obtained from the author.

[Afsarmanesh 85] Afsarmanesh,H., Knapp,D., McLeod,D., and Parker,A.: "An Extensible Object-Oriented Approach to Databases for VLSI/CAD"; Univ.of Southern Cal., CSD, TR-85-330, Apr.1985.

[Ashok 84] Ashok,V., McKnight,W., and Ramanathan,J.: "Integrated Environment for Information Management in VLSI Design"; IEEE Data Engineering Conf. 1, Los Angeles, Apr.1984.

[Bancilhon 85] Bancilhon,F., Kim,W., and Korth,H.F.: "A Model of CAD Transactions"; VLDB 11, Aug.1985, Stockholm.

[Batory 80] Batory,D.S. and Kim,W. "Modeling Concepts for VLSI CAD Objects"; *ACM TODS*, vol.10 no.5, Sep.1985, pp.322–346.

[Beech 83]] Beech,D. and Feldman,J.S.: "The Integrated Datamodel: A Database Perspective"; Proc. VLDB 9, Florence, 1983.

[Borning 79] Borning,A.: THINGLAB, *A Constraint Simulation Laboratory*; Rep.No. CS-79-746, Stanford Univ., 1979.

[Brown 83] Brown,Harold,D., Tong,C., and Foyster,G.: "Environment for Circuit Design"; *IEEE Computer*, Vol.16,No.12, Dec.1983.

[Carter 86] Carter,Harold W.: "Computer-Aided Design of Integrated Circuits"; *IEEE Computer*, vol.19 no.4, Apr.1986, pp.19–36.

[Dayal 82] Dayal,U. and Bernstein,P.A.: "On the Correct Translation of Update Operations On the Relational View"; *ACM TODS*, vol.7 no.3, Sep.1983.

[DittrichD 86] Dittrich,K. and Dayal,U. (Eds.): "Proceedings 1986 International Workshop on Object-Oriented Database Systems"; IEEE CS Order no.734, Sep.1986

[Fischer 83] Fischer,G. and Boecker,H.D.; "The Nature of Design Processes and how a Computer System can Support Them"; in Degano and Sandewall(eds): *Integrated Interactive Computing Systems*, North-Holland 1983.

[Gray 80] Gray, Mike: "Databases for Computer-Aided Design"; in *New Applications of Databases*, Gardarin and Gelenbe(eds), Academic Press 1984.

[Hardwick 80] Hardwick,M.: "The Texas Tech Abstraction Management System"; Tech.Rep., Texas Tech Univ., Lubbock TX, Received Mar.1985.

[Hartzband 85] Hartzband,D.J. and Maryanski,F.J.: "Enhancing Knowledge Representation in Engineering Databases"; IEEE Computer, Vol.18 no.9, Sep.1985, pp.39–48.

[Hewitt 77] Hewitt,C.: "Viewing Control Structures as Patterns of Passing Messages"; *Art.Intell., vol.8, pp.323–364., 1977.*

[Holland 85] Holland,L., Korn,G., Matson,J., and Wolfe,P.: "Engineering Support System Software"; IEEE Micro, Vol.5 no.5, Oct.1985, pp.17–21.

[IDA 86] Institute for Defense Analysis: *Engineering Information System Specifications, 2 vols.*; Report prepared for the VHSIC program, Wright-Patterson AFB, OH, June 1986.

[Katz 83] Katz, Randy H.: "Managing the Chip Design Database"; IEEE Computer, Dec.1983, pp.16–36.

[Katz 84] Katz,R.H. and Lehman,T.J.: "Database Support for Versions and Alternatives for Large Files"; IEEE TSE, vol.SE10 no.2, Mar.1984, pp.191–200.

[Keller 86]
Arthur M. Keller: "Choosing a View Update Translator by Dialog at View Definition Time"; *IEEE Computer*, January 1986.

[Ketabchi 86] Ketabchi,M. and Berzins,V.: "Component Aggregation: A Mechanism for Organizing Efficient Engineering Databases"; IEEE CS Data Engineering Conf.2, Los Angeles, Feb.1986.

[Liskov 77] Liskov,B., Snyder,A., Atkinson,R., and Schaffert,C.: "Abstraction Mechanisms in CLU"; CACM, vol.20 no.8, Aug.1977, pp.564–576.

[Lochovsky 85] Lochovsky,F. (Ed.): *Object-oriented Systems*, special issue of the IEEE CS Database Engineering Bulletin, December 1985; reprinted in Database Engineering 1985, IEEE CS order no. 758, 1986.

[Lorie 83] Lorie,R.(ed): "Engineering Design Applications"; ACM-SIGMOD 83 vol.2, Proc. Database Week, San Jose, May.1983.

[Mermet 85] Mermet,J.: "CASCADE: vers un systeme integre de CAO de VLSI"; INRIA Bull., no.102, Jun.1985, pp.16–29.

[Milton 83] Milton,Jack and Wiederhold,Gio: "Network and Relational Systems in VLSI Design: Contradictory Performance?"; IEEE CS DataBase Engineering, Vol.1, 1983, pp.167–170.

[Minoura 82] Toshimi Minoura and Gio Wiederhold: "Resilient Extended True-Copy Token Scheme for a Distributed Database System"; *IEEE Transactions on Software Engineering*, vol.SE-8 no.3, May 1982, pp.173–188.

[Moreau 85] Moreau, J-P. and Vuilllemin,J.: "Le Projet SYCOMORE"; INRIA Bull., no.102, Jun.1985, pp.2–10.

[Mukherjee 86] Mukherjee,A.: *Introduction to nMOS and CMOS VLSI Systems Design*; Prentice-Hall, 1986.

[Narfelt 85] Narfelt, K-H. and Schefstrom, D.: "Extending the Scope of a Program Library" ; Ada in Use, Barnes and Fisher(eds), Ada Letters, Vol.V no.2, Sep.-Oct.1985, pp.25–40.

[Neuman 82] Neuman,T. and Hornung,C.: "Consistency and Transactions in CAD Database"; VLDB 8, McLeod and Villasenor(eds), Mexico City, 1982, pp.181–188.

[Roth 84] Roth,M.A., Korth,H.F., and Silberschatz,A.: "Theory of Non-First-Normal Form Relational Databases"; Tech.Rep.84-38, Dep.of CS, Univ.of Texas, Austin, Dec.1984.

[Roussopoulos 86] Roussopoulos,N. and Kang,H.: "Preliminary Design of ADMS ±: A Workstation-Mainframe Integrated Architecture for Database Management Systems"; VLDB 12, Kyoto 1986, pp.355-362.

[Schkolnick 80] Schkolnick,M. and Sorensen,P.: "Denormalization: A Performance Oriented Database Design Technique"; AICA '80, Bologna Italy, Oct.1980.

[SmSm 77] Smith, John and Smith, Diane C.P.: "Database Abstractions: Aggregation and Generalization"; *ACM TODS*; vol.2 no.2. June 1977, pp.105–133.

[Spooner 86] Spooner,D.L., Milicia,M.A., and Faatz,D.B.: "Modeling Mechanical CAD Data with Data Abstraction and Object-Oriented Techniques"; IEEE CS Data Engineering Conf.2, Los Angeles, Feb.1986.

[Stonebraker 83] Stonebraker,M., Rubenstein,B., and Guttman,A.: "Application of Abstract Data Types and Abstract Indices to CAD Data Bases"; ACM-SIGMOD 83, Database Week, San Jose CA, May.1983, pp.107–113.

[Stonebraker 85] Stonebraker,M. and Rowe,L.: "The Design of POSTGRES"; Tech.Report UC Berkeley, Nov.1985.

[Stroustrup 86] Stroustrup, B.: *The C++ Programming Language*; Addison-Wesley, 1986.

[Tichy 82] Tichy,Walter F.: "Design Implementation and Evaluation of a Revision Control System"; Proc.6th Int. Conf. Software Eng., Tokyo, Sept.1982.

[Udagawa 84] Udagawa,Y. and Mizoguchi,T.: "An Advanced Database System: ADAM — Towards Integrated Management of Engineering Data"; IEEE Data Engineering Conf. 1, Los Angeles, Apr.1984.

[Wilkins 85] Wilkins,M.W., Berlin,R., Payne,T., and Wiederhold,G: "Relational and Entity-Relationship Model Databases and Specialized Design Files in VLSI Design"; ACM-DA 22, 1985, pp.410–516.

[Zara 85] Zara,R.V. and Henke,D.R.: "Building A Layered Database For Design Automation"; ACM-DA 22, 1985, pp. 645-651.