Viewers: A Data-World Analogue of Procedure Calls

Kazimierz Subieta* Florian Matthes Joachim W. Schmidt Andreas Rudloff
Ingrid Wetzel

University of Hamburg
Department of Computer Science
Vogt-Kölln-Straße 30
D-2000 Hamburg 54, Germany

Abstract

A viewer is a reference-valued datum with a special meaning: a value of the data pointed by the viewer becomes a virtual part of data where the viewer is placed; the value virtually substitutes the viewer. Viewers are considered to be a data-world analogue of procedure calls. They possess a large conceptual and pragmatic potential as a result of new data semantics on which we can base a variety of well-organized data structures. Various applications of viewers, related to DB-PLs and object-oriented data modelling, are presented: importing common attributes, inheritance and multi-inheritance, stored selections, projections and joins, viewing a single relational structure as several hierarchical structures, etc. Methodological and formal aspects of the concept are discussed and a method of incorporating viewers into a query language is presented.

1 Introduction

A viewer is a reference-valued datum with a special semantics: a value of data referenced by the viewer virtually substitutes the viewer. Viewers have some similarities with the well-known concept of database views as found e.g. in POSTGRES [12], which can be used to build virtual nested data structures from relational ones. There are, however, essential conceptual and pragmatic differences between viewers and views; thus they should be considered as distinct notions. The basic differences are as follows:

- Views are properties of a particular (query) language. Viewers are not connected to any language: they are properties of data structures.
- Names of views occur explicitly in queries. Names of viewers never occur in queries (they occur in meta-statements only).
- Views are evaluated dynamically (when they are needed) which has consequences for performance. Viewers are properties of static data thus there is no performance problem, however, sometimes they share negative properties of materialized views.
- Updating through views leads to problems (at least in value-oriented frameworks). There are no problems in updating through viewers (though it may lead to other anomalies).
- View definitions are not updatable from inside the program since their external form is a source text. Viewers, as normal data, can be updated by assigning new references as their values.

Viewers play the same part for data as procedure calls for sequences of instructions. They allow one to see some data from different places in the database structure, similarly as procedures allow utilization of a piece of code in different program points. In view of the analogy we believe that viewers are of a large conceptual and pragmatic importance, allowing the building of well-organized data structures. In the paper we discuss various potential applications of the viewers in data-intensive environments.

Similarly to other data abstractions, viewers introduce a new kind of data semantics. More semantics

*Current address: Institute of Computer Science, Polish Academy of Sciences, Ordona 21, PL-01-237 Warszawa, Poland

Proceedings of the 19th VLDB Conference
Dublin, Ireland, 1993

inside data supports natural data views and simplification of application programs, which need not incorporate this semantics into their code. For example, due to path expressions queries addressing object-oriented data structures are usually shorter than relational queries. Path expressions are based on additional data semantics such as explicit hierarchical data structures, pointer links, or referential integrities. Complex objects, object sharing, classes, is-a relationships, behavioral invariants of classes (methods), active rules, etc. can be considered as methods of introducing more semantics into data.

Sophisticated methods concerning data semantics (knowledge representation) are considered in AI. The database domain have already been partly adopted AI ideas, for example, data abstractions and behavioral properties. This import, however, is constrained by engineering requirements, in particular, proper performance and easy-to-understand data views. For example, deductive databases have a potential to represent advanced data semantics; however, current research does not make evident that the acceptable performance is feasible. Therefore, in contrast to unlimited inventions of AI, database researchers must look for such features in the data world which are capable of representing attractive conceptual modelling primitives, are easy to implement, yield proper performance, and are easy from the programmer point of view.

A well-known technique covering many issues related to data semantics lies in the application of references (pointer links). References in network and object-oriented databases explicitly represent different kinds of dependencies between objects, such as sharing, cyclicity, subordination, aggregation and association. From a practical point of view the advantage of references is easy implementation and good performance.

From another aspect, pointer links in databases can be compared to *gotos*, which have been recognized as leading to impossible-to-understand programs. There is an analogy: a CODASYL or an entity-relationship schema for large databases resembles a maze. In relational databases the situation is even more difficult, since instead of explicit links carrying semantic information there is an attribute naming convention and assertions in a natural language. Integrity constraints, such as functional dependencies and referential integrities are conceptual links in the relational schema, turning it to a similar "maze". This problem also occurs in object-oriented databases.

In programming methodologies the problem of "spaghetti-like" structures has been avoided by structured programming which is based on procedures as semantic units of the programs. Procedures encapsulate semantic meaning for the designer and the reader of the program, and thereby allow the understanding of the program as a hierarchical structure. In the data world a similar notion can be achieved by hierarchically organized data repositories, scoping, encapsulation, locality of references [6], etc. Still, there is no possibility to organize different hierarchical data views assuming data sharing and proper performance. Viewers are the solution to this problem. Navigation according to a viewer is not definite (as pointers and *gotos* are), but always implies returning to data where the viewer is placed; thus the analogy with procedures.

We will show through examples that viewers are able to cover surprisingly many conceptual modelling issues that are currently the focus of considerations of DBPLs and object-oriented approaches. An advantage of viewers is their capability to simplify queries. Due to viewers the database designers and programmers receive a full control over so-called *automatic navigation*, which was the main motive for the 5-th normal form of relational databases (known also as *universal relation*) [1, 7].

The viewer concept has already some precedences. Wilkes et al. introduced the concept of "instance inheritance" in [16]. The idea is based on the observation that inheritance may concern not only methods and attribute definitions but also some values of attributes. For example, common or default values can be stored as first-class objects inside a class and then imported as virtual attributes (without copying) by class members. Ohori et al. [9] introduced in Machiavelli the concept of coercions or "views". A Machiavelli view is a set of simple records having references to complex records. In this way attributes of the complex records are virtually imported by the simple records, allowing the programmer, for example, to perform a natural join on the virtual attributes. Independent of this research, in LOQIS [13, 14] we implemented a more general concept, covering the instance inheritance and Machiavelli views.

The rest of the paper is organized as follows. In Section 2 we present various examples of potential applications of viewers. Section 3 is devoted to more general observations, and in Section 4 we briefly discuss consequences for database design methodologies, formal aspects, and modifications of query languages.

2 Viewers: Examples of Applications

The figures presented in this section show viewers as data having some name, and a pointer value depicted as an arrow. For retrieval such data structures are equivalent to structures where all viewers are textually substituted by values of data they point to; viewers are transparent for users. For example, the structure shown in Figure 1 should be understood as the following data [1]:

```
PET( NAME(Rex)   KIND(dog)
       LEGS(4) EARS(2) EYES(2) )
PET( NAME(Pussy) KIND(cat)
       LEGS(4) EARS(2) EYES(2) )
ANIMAL( LEGS(4) EARS(2) EYES(2) )
```

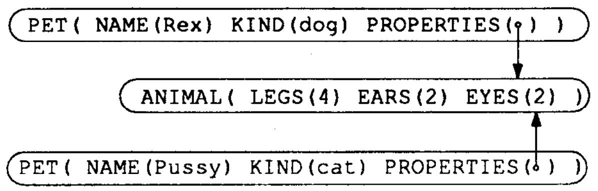

PET(NAME(Rex) KIND(dog) PROPERTIES( ))
ANIMAL(LEGS(4) EARS(2) EYES(2))
PET(NAME(Pussy) KIND(cat) PROPERTIES( ))

Figure 1: Shared attributes

The value of the datum **ANIMAL**, being the record **LEGS(4) EARS(2) EYES(2)**, virtually substitutes the data **PROPERTIES**.

2.1 Object-Oriented Issues

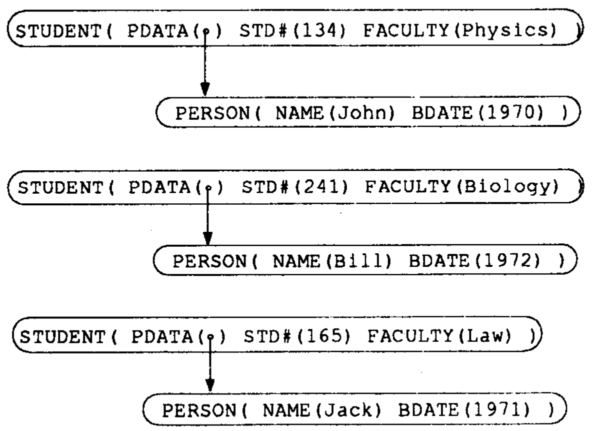

STUDENT(PDATA( ) STD#(134) FACULTY(Physics))
PERSON(NAME(John) BDATE(1970))
STUDENT(PDATA( ) STD#(241) FACULTY(Biology))
PERSON(NAME(Bill) BDATE(1972))
STUDENT(PDATA( ) STD#(165) FACULTY(Law))
PERSON(NAME(Jack) BDATE(1971))

Figure 2: Example of structural inheritance implemented by viewers

In Figure 2 we present an example of structural inheritance: a **STUDENT** object inherits basic data from a **PERSON** object. This data structure is understood as follows:

```
PERSON( NAME(John) BDATE(1970) )
PERSON( NAME(Bill) BDATE(1972) )
PERSON( NAME(Jack) BDATE(1971) )

STUDENT( NAME(John) BDATE(1970)
         STD#(134) FACULTY(Physics) )
STUDENT( NAME(Bill) BDATE(1972)
         STD#(241) FACULTY(Biology) )
STUDENT( NAME(Jack) BDATE(1971)
         STD#(165) FACULTY(Law) )
```

For class hierarchy viewers can act transitively, i.e. they import data imported by other viewers. (This feature is implemented in LOQIS.) Viewers allow multi-inheritance: any number of viewers can be stored inside a data object. Note that (similarly to views) viewers automatically propagate updates of **PERSON** objects to **STUDENT** objects.

It is observed in [10] that a quite common situation is not that STUDENT *is a* PERSON, but that PERSON *become a* STUDENT. This paraphrasing underlines dynamic nature of data views (the schema evolution): during the life of a database systems objects can gain and lose many independent roles. As we can see from the Figure 2 viewers supply a mechanism for dealing with this problem: any number of roles such as STUDENT can be dynamically created and deleted.

Figures 1 and 3 present examples of shared attributes. Attribute sharing may be useful for long attributes, for example, if they represent graphical objects or texts.

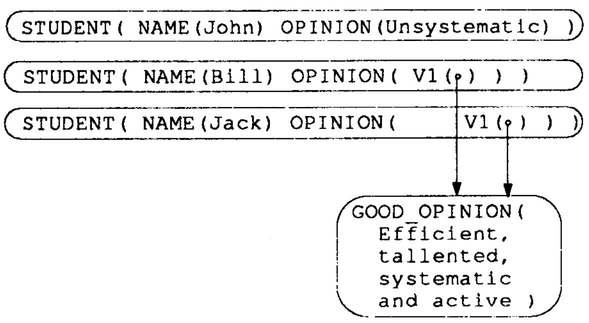

STUDENT(NAME(John) OPINION(Unsystematic))
STUDENT(NAME(Bill) OPINION(V1( )))
STUDENT(NAME(Jack) OPINION(V1( )))
GOOD_OPINION(Efficient, tallented, systematic and active)

Figure 3: Utilization of a shared long attribute

Classes in object-oriented approaches can be considered as data repositories storing invariant attributes for their members. Several kinds of such invariants can be considered, for example, common attributes, default attributes, methods, types, constraints, icons, etc. In Figure 4 we show how class invariants can be imported to particular objects by application of viewers. The figure shows cases of overriding; Bill is smoking and John

[1] In this paper we apply a syntax in which a bulk datum with name N and value $\{v_1, v_2, ..., v_n\}$ is written as $N(v_1)\ N(v_2)\ ..\ N(v_n)$

has a special firing procedure.

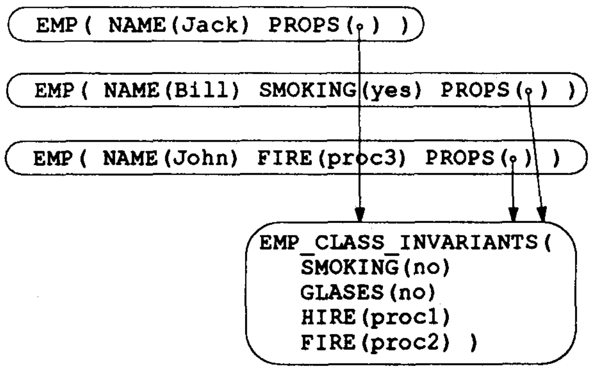

EMP (NAME(Jack) PROPS( ))
EMP (NAME(Bill) SMOKING(yes) PROPS( ))
EMP (NAME(John) FIRE(proc3) PROPS( ))
EMP_CLASS_INVARIANTS(SMOKING(no) GLASES(no) HIRE(proc1) FIRE(proc2))

Figure 4: Importing class invariants

In Figure 5 viewers organize object sharing. Note that viewers allow to introduce local aliases for objects: a PERSON accessed from inside of another object has an alias WIFE, HUSBAND, CHILD, MOTHER, etc.

Using viewers as supported by LOQIS we can easily formulate queries on such a structure, for example, "Give name of the wife of the father of Mary's husband" as follows:

```
(PERSON where NAME = "Mary").
             HUSBAND.FATHER.WIFE.NAME
```

Another query, "For each person over 30, give name, the number of children, the number of siblings, and the number of first-order cousins of the same generation", can be formulated as follows:

```
((PERSON where AGE > 30) ⋉
  (s ∈ count(unique(
       (MOTHER ∪ FATHER).CHILD.NAME)))).
(NAME × count(CHILD) × (s - 1) ×
    (count(unique((MOTHER ∪ FATHER).
                  (MOTHER ∪ FATHER).
                  CHILD.CHILD.NAME)) - s))
```

(See [15] for detailed specification of this language. ⋉ denotes a navigational join, '.' denotes projection/navigation; other operators have typical meaning.) To formulate this example in the relational model, PERSON and 5 additional relations should be defined. The last query is extremely difficult to formulate in SQL and we have doubts if SQL processors are able to optimize it. Due to viewers implemented as pointer links the query is executed in LOQIS in a reasonable time.

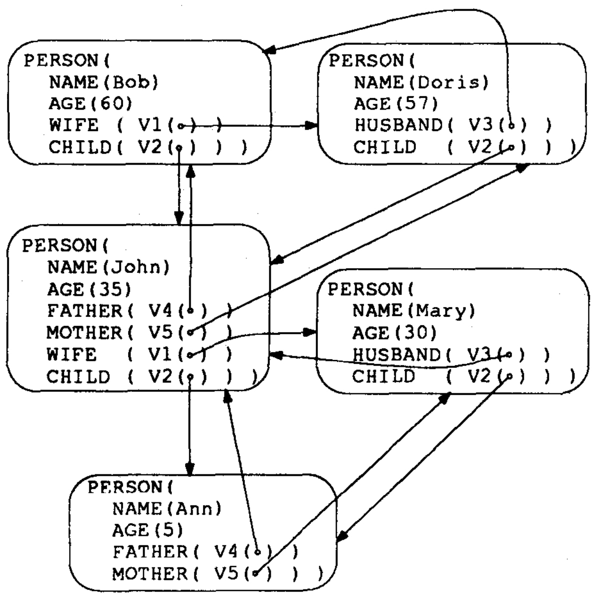

PERSON(NAME(Bob) AGE(60) WIFE (V1( )) CHILD(V2( )))
PERSON(NAME(Doris) AGE(57) HUSBAND(V3( )) CHILD (V2( )))
PERSON(NAME(John) AGE(35) FATHER(V4( )) MOTHER(V5( )) WIFE (V1( )) CHILD (V2( )))
PERSON(NAME(Mary) AGE(30) HUSBAND(V3( )) CHILD (V2( )))
PERSON(NAME(Ann) AGE(5) FATHER(V4( )) MOTHER(V5( )))

Figure 5: Implementation of Shared Objects

2.2 Stored Selections, Joins and Projections

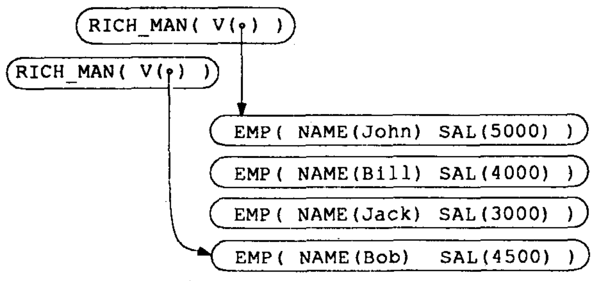

RICH_MAN(V( ))
RICH_MAN(V( ))
EMP(NAME(John) SAL(5000))
EMP(NAME(Bill) SAL(4000))
EMP(NAME(Jack) SAL(3000))
EMP(NAME(Bob) SAL(4500))

Figure 6: Implementation of a stored selection

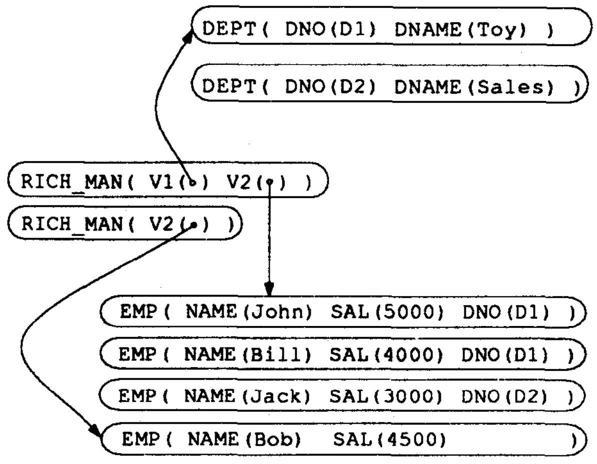
In Figure 6 we show the possibility to store in objects **RICH_MAN** the result of the selection **EMP where SAL >** **4000**. The resulting **RICH_MAN** data are seen as follows:

```
RICH_MAN( NAME(John) SAL(5000) )
RICH_MAN( NAME(Bob)  SAL(4500) )
```

The idea shown in Figure 6 allows us to store joins followed by arbitrary selections. In Figure 7 objects **RICH_MAN** store outer join between **DEPT** and **EMP**, followed by selection **SAL > 4000**; the result is equivalent to the following data [2]:

[2] Note our convention: when the value of some data is NULL, we do not write it at all.

```
RICH_MAN( DNO(D1) DNAME(Toy)
          NAME(John) SAL(5000) )
RICH_MAN( NAME(Bob)  SAL(4500) )
```



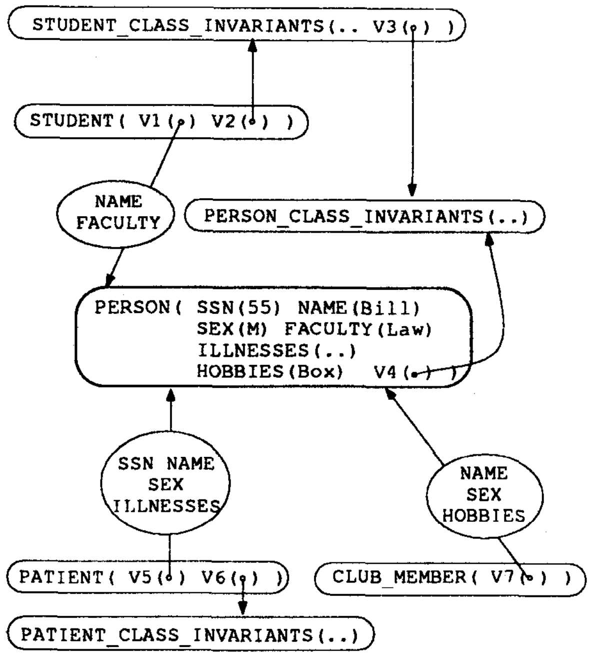
DEPT(DNO(D1) DNAME(Toy))
DEPT(DNO(D2) DNAME(Sales))
RICH_MAN(V1( ) V2( ))
RICH_MAN(V2( ))
EMP(NAME(John) SAL(5000) DNO(D1))
EMP(NAME(Bill) SAL(4000) DNO(D1))
EMP(NAME(Jack) SAL(3000) DNO(D2))
EMP(NAME(Bob) SAL(4500))

Figure 7: Stored outer join followed by selection

Stored projections require equipping the viewer with an additional feature: filtering the data that are seen through it. The simplest way to do this is associating with a viewer a set of data names; data having other names are not imported. This feature is implemented in LOQIS. For example, in Figure 7 we can equip viewers V1 with a list containing **DNAME** and **V2** with a list containg **NAME**, the result will be the projection of **RICH_MAN** onto attributes **DNAME** and **NAME**:

```
RICH_MAN( DNAME(Toy) NAME(John) )
RICH_MAN( NAME(Bob) )
```

Richardson and Schwarz [10] proposed to extend the object concept in order to support multiple independent roles for objects, preserving object identity. Figure 8 presents how viewers equipped with the filtering mechanism can support this feature. (Ovals over arrows denote data filters.) In this approach all attributes of a PERSON are collected in one variable, thus the uniqueness of identity is preserved. Different person roles are implemented as separate variables (having their own identities), but they store only viewers. Such an organization has both advantages: all attributes of a person are identified by a single identity (which may be important for administrative functions) and simultaneously, this object in a particular role has a separate identity (which is necessary for limitation of the scope of queries). If necessary, special coercion functions can be implemented in order to map e.g. a PERSON identity to a STUDENT identity and vice versa.



STUDENT_CLASS_INVARIANTS(.. V3( ))
STUDENT(V1( ) V2( ))
NAME FACULTY
PERSON_CLASS_INVARIANTS(..)
PERSON(SSN(55) NAME(Bill) SEX(M) FACULTY(Law) ILLNESSES(..) HOBBIES(Box) V4( ))
SSN NAME SEX ILLNESSES
NAME SEX HOBBIES
PATIENT(V5( ) V6( ))
CLUB_MEMBER(V7( ))
PATIENT_CLASS_INVARIANTS(..)

Figure 8: An object PERSON in several independent roles

This picture shows that viewers support richer data semantics that is typically assumed in object-oriented approaches. For example, STUDENT has less attributes than PERSON, but the STUDENT class is a subset of the PERSON class.

2.3 Network Structures Seen as Several Hierarchies

Hierarchical organizations are perhaps the most understandable for humans but have disadvantages. For many-many relationships hierarchical data views are undesirable since they lead to redundancy in representation, which in many cases is undesirable. Moreover, different users may require different hierarchical views. Considering the SUPPLIER-PART database, a clerk from the personnel department is interested in suppliers and rarely in parts, and clerk from the storage department is interested basically in parts, but sometimes his interests concern suppliers. If the database is to be organized hierarchically, the database administrator should decide which point of view is more important.

Viewers make possible implementation of a network structure which can be considered as several hierarchies. To explain this topic we introduce some notation. We enhance previously used prefixed lists (representing instances of data) by context-free grammars. A database schema is a grammar describing possible database instances; names of types are non-terminals in the grammar. As usual, {..} means iteration, [..] mean optional data, and | means alternative (exclusive variants).[3]

We refer to the SUPPLIER-PART relational database, which has the following description:

```
{ SUPPLIER( SNO(string)
            SNAME(string) ) }

{ PART(     PNO(string)
            PNAME(string)
            WEIGHT(real) ) }

{ SP(       SNO(string)
            PNO(string)
            QTY(integer) ) }
```

The data view appropriate for a personnel clerk can be represented as the following NF^2 structure:

```
{ SUPPLIER(supplier-type) }

supplier-type ← SNO(string)
                SNAME(string)
                what-supplies

what-supplies ← {SUPPLIES(part-with-qty)}

part-with-qty ← PNO(string)
                PNAME(string)
                WEIGHT(real)
                QTY(integer)
```

The view emphasizes **SUPPLIER** data; the information about parts and their quantities is hidden in the type *part-with-qty* inside a lower hierarchy level.

The NF^2 view appropriate for a storage clerk puts the **PART** data on the first plan:

```
{ PART(part-type) }

part-type      ← PNO(string)
                 PNAME(string)
                 WEIGHT(real)
                 who-supplies

who-supplies   ← {SUPPLIED_BY(supp-with-qty)}

supp-with-qty  ← SNO(string)
                 SNAME(string)
                 STATUS(integer)
                 QTY(integer)
```

[3]A data checker based on a schema understood as a context-free grammar is implemented in LOQIS.

Current DBMSs and database theories are not able to express efficiently simultaneously *both* hierarchies assuming data sharing. In Figure 9 both hierarchies are represented. Names **QTY** inside ovals denote data filters. Due to viewers we have received virtual *infinite hierarchical structures* (an interesting object for mathematics and having some flavour of recursion within viewers): each part within supplier again contains information about suppliers, and so on. This may be convenient for queries such as "Find suppliers supplying the same parts as Smith does" (in LOQIS: `(SUPPLIER where SNAME = "Smith").SUPPLIES.SUPPLIED_BY.SNAME`), or for queries requiring transitive closures.

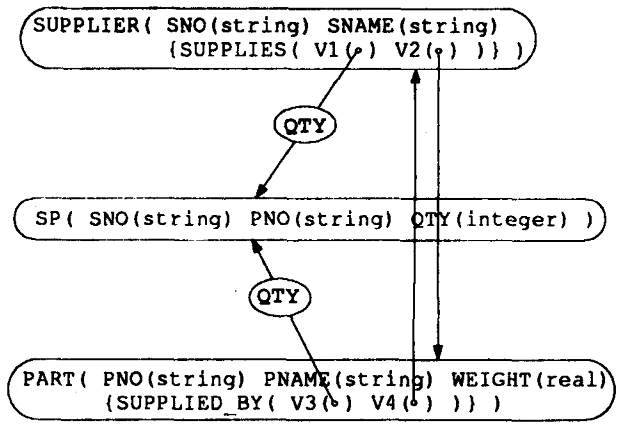

SUPPLIER(SNO(string) SNAME(string)
{SUPPLIES(V1( ) V2( ))})
QTY
SP(SNO(string) PNO(string) QTY(integer))
QTY
PART(PNO(string) PNAME(string) WEIGHT(real)
{SUPPLIED_BY(V3( ) V4( ))})

Figure 9: Implementation of two hierarchies

2.4 Data Independence and Version Management

Standard data independency problems concern how to make one record from existing two records, or how to make two records from existing one. Viewers present relevant facilities. Let A,B denote lists of attributes. Assume a database contains records R1(A) and R2(B), and we would like to substitute them by records R3(A B). Thus we make records R3(A B) and augment the database with records R1(V1(viewer to R3)), and R2(V2(viewer to R3)); V1 has a filter with names of A, and V2 has a filter with names of B. A similar method can be applied for the case of splitting one record into two. Viewers may be also useful on physical level for storing long fields, assuming a fixed format of objects. Instead of the value we can store a viewer leading to an overflow area which would allow retaining of the fixed format.

In the case of schema evolution new attributes can be introduced to existing records. For example, old records $R_1(A_1,..,A_m)$ need to be augmented by attributes $B_1,..,B_n$, and then by attributes $C_1,..,C_k$. Assuming that in each record created in the database a space for a viewer is left, extension of the record R_1 can be done as shown in Figure 10.

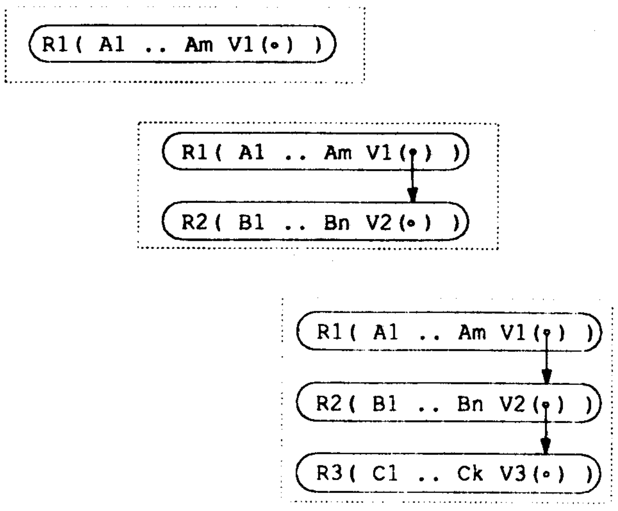


Figure 10: Extending record R_1

In CAD/CAM applications several versions of the same object can exist. Versions represent different states of some object but they may share common sub-objects and other common properties. Every common property should be a separate object and updating of it should be automatically propagated to all versions. Viewers supply a convenient mechanism for this purpose. For example, assume that **VERSION1** is described by (arbitrarily complex) attributes **ATTR1,...,ATTRx,...,ATTRn**, and **VERSION2** is made from **VERSION1** by changing attribute **ATTRx**. This situation can be described as follows:

```
VERSION1( ATTR1(..)..
          ATTRx(value of 1-st version)..
          ATTRn(..) )

VERSION2( V(viewer to VERSION1)
          ATTRx(value of 2-nd version) )
```

VERSION2 inherits all attributes of **VERSION1**, but **ATTRx**, which is overridden by own attribute **ATTRx**.

3 Updating and Typing

Viewers require a proper level of data abstraction. If data views and processing are too close to physical representation there may be no possibility to distinguish viewers from pointers; in this case viewers introduce no new quality. (By analogy, in assemblers procedure calls are simply *gotos* with some additional features.) Most relational systems have a sufficient level of abstraction and they deal with persistent pointers (known as tids). Thus implementation of viewers is possible but profits imposed by viewers may be decreased by the 1NF requirement. We believe that most of all viewers would be profitable in object-oriented database systems such as O_2 [4] and ORION [5], which support a high level of data abstraction and explicitly deal with complex hierarchical objects and persistent pointers.

So far we have assumed that the data pointed by the viewer substitute it, thus viewers are invisible at the level of user interfaces. This assumption can be true only for retrieval. When considering updating the user should be aware of differences between normal updating and updating of data imported by viewers. Hence, at the level of data types or data description we must explicitly specify viewers, and some kinds of users (dealing with updating) must be aware of their existence.

The same concerns operations which must be performed on viewers. A viewer must be initialized, i.e. a reference must be assigned as its value. The reference being a value of the viewer must be the subject of updating. The discussion concerning necessity of updating of inheritance relationships is presented in [2]. Thus, the programming language should provide special statements, which "see" viewers and enable proper operations on them. We consider that these statements belong to syntactically distinguished layer of the programming language.

Assuming static binding and strong typing we must provide capabilities for declaration and typing of viewers, and for changing their values. For example (see Figure 1), an extension of DBPL [8, 11] to deal with viewers may be the following:

```
TYPE
  AnimalProps =
    RECORD
      LEGS, EARS, EYES: integer;
    END;
  SinglePet =
    RECORD
      NAME: string;
      KIND: (dog, cat, ... );
      PROPS: IMPORT AnimalProps;
    END;
  Pet = RELATION NAME OF SinglePet;

VAR
  PET    : Pet;
  ANIMAL, DOG : AnimalProps;
  X : integer;

BEGIN
  ANIMAL := AnimalProps{4,2,2};
  PET := Pet{ {"Rex", dog, ANIMAL},
              {"Pussy", cat, ANIMAL}};
  .....
  PET["Rex"].PROPS := DOG;
  X := PET["Pussy"].EARS;
     (* Possible typing problems *)
  .....
END;
```

The type **IMPORT AnimalProps** denotes a viewer leading to a variable of type **AnimalProps**. Typing of viewers is the same as typing of pointers, and initialization and updating of viewers can be done by standard capabilities. However, typing of field selections may imply problems, since viewers can import foreign attributes into the data. If typing of viewers is static and complete, this import is determined during compilation time, thus the problem is exactly the same as with multiple inheritance [3]. If typing of the viewers' import cannot be static, e.g. because overriding by attributes with unknown types or importing random data, static strong typing is problematic.

Assuming dynamic binding, we can dynamically create, insert, modify and delete viewers; this approach is implemented in LOQIS. Actions similar to the above DBPL example are the following:

```
create permanent ANIMAL
  begin LEGS(4) EARS(2) EYES(2) end;
create permanent PET
  begin
    NAME("Rex") KIND("dog")
    PROPS( import from ANIMAL )
  end;
create permanent PET
  begin
    NAME("Pussy") KIND("cat")
    PROPS( import from ANIMAL )
  end;
```

LOQIS provides a special assignment operator for changing values of viewers. For example, if **DOG** is another datum similar to **ANIMAL**, we can use the following construct:

```
store pointer to DOG in
  (PET where NAME = "Rex").PROPS;
```

The identifier of DOG is stored as a value of PROPS. Other operators are the same as for normal data, for example, deleting the viewer for Pussy can be done by the statement:

```
delete ((PET where NAME = "Pussy").PROPS);
```

Current typing systems are not prepared for such features, thus dynamic capabilities should be somehow restricted or novel typing ideas should be developed.

Since viewers transfer control to pointed data, updating of data that are seen through viewers is feasible. However, there is a possibility of updating anomalies. Returning to the PET-ANIMAL example, assume that Rex in some dogs' battle lost an ear. If we directly update **EARS** via the Rex object, it will cause updating in the object **ANIMAL** and, in consequence, innocent Pussy will also loss an ear. Instead of updating of the **ANIMAL** object we should insert a datum **EARS(1)** to the Rex object, which will override the datum inherited from **ANIMAL**. Such semantics of the assignment operator is a novelty in programming languages and may lead to some unexpected effects.

If a class of viewers has predefined external semantics, for example, it represents the stored view **RICH_MAN: EMP where SAL > 4000**, there is a problem with propagation of updates to viewers. For example, if Bill's salary is increased, he will become **RICH_MAN**, thus a new datum **RICH_MAN** should be inserted, with a viewer pointing the Bill's object. This problem is the same as for materialized views and can be solved by two methods: by providing a manual for the programmer saying what should be additionally done when some update is performed, or more ambitiously, by active rules propagating the update automatically.

In the presented examples we recognized the necessity of filtering of data imported by viewers. This can be done by attaching a list of names to a viewer. Another kind of filtering is overriding: testing if a data having a particular name is already present in the object containing a viewer, and then do or do not the import. Sometimes filtering of data according to equality of names may be irrelevant since names equivalence may be quite casual, or we would like to import data in spite of the name conflict; see aliasing of imported methods in O_2. Thus we can assign to a viewer a rule determining renaming of data during the import. Further inventions in this respect could be based on analogy with procedures: viewers with parameters, viewers with encapsulated meaning, etc. Such ideas should be carefully tested in some real database environment.

4 Design Methodologies, Formalization and Query Languages

Since viewers are in fact pointers, we can expect that viewers are cheap from the point of view of either consumption of storage, access times, and additional maintenance functions. This creates a potential for extensive use of this facility in database systems. Implementation of viewers in a database system may change methodologies of database design because of the following factors:

- They provide a possibility of expressing simultaneously different data views required by particular applications, without the necessity of introducing one global data view and then transforming it by some external views mechanisms.

- They allow more freedom in changing the data view without destroying existing programs (data independence), thus relaxing the initial responsiblity of the database schema designers.

By analogy with procedures, viewers may have meaning in the structural database design, relying on top-down designing of data hierarchies, starting from the most important levels for particular data applications, and step-by-step refinement of necessary details down in the data hierarchy. Such a design methodology is already assumed in the entity-relationship approach, but integrating independently developed parts presents actually an essential methodological problem. As we observed in Section 2.3, viewers make possible independent development of hierarchies and then integrate them by establishing shared parts. This may present a new potential for the database design.

Several aspects of viewers call for the formal approach. One of them is formalization of *intensional* and *extensional* data, and the mappings between them. Intensional data involve viewers, while extensional data are obtained by developing all viewers according to their meaning. This leads to some substitution concept (macro-substitution), or theory of rewriting rules. We recall, however, that semantics of procedure calls based on the textual substitution has been abandoned because it does not support locality of objects with which the procedure deals. It is not excluded that some concept of the locality control for viewers will require an approach different from the textual substitution.

The second aspect is formalization of schemata (both intensional and extensional) and mapping between them. This aspect may have meaning for the database design. For example, equivalence between some constructs of intensional and extensional schema may be the basis for unification, normalization or optimization.

Formalization could concern utilization of viewers in languages, for example, in query operators or programming primitives. The goal of such investigations is establishing equivalent query constructs which is important for optimization. Another goal is recognizing a new quality introduced by viewers to the current database or knowledge-base theories.

Viewers have a direct impact on query languages. In principle each query language, e.g. SQL, can deal with viewers because from the programmer or user point of view viewers are invisible during data retrieval and manipulation. In many cases viewers can substitute views; for example, POSTQUEL [12] features extending the relational model can be equivalently based on viewers. However, there are "meta" functions, such as creation and updating of viewers, which cannot be consistently handled by value-oriented languages. In fact, the idea is based on the concept of data identifier, thus only query language that deal with this concept are appropriate.

For reasons of space we do not attempt to fully specify a query language that is relevant for viewers. (See [14, 15] for detailed description of such a language.) We adopted the stack machinery of classical programming languages to define operators of query languages. In this approach the central role is played by notions of naming, scoping and binding. Essentially, viewers do not introduce a new quality to this mechanism; it requires minor corrections of scoping and binding rules. Such a mechanism is implemented in LOQIS. For example (see Figure 9), the query "Give names of suppliers together with quantities of bolts supplied by them" can be expressed in LOQIS as

```
SUPPLIER.(SNAME × sum(
     (SUPPLIES where PNAME = "bolt").QTY) )
```

In this example we employed viewers V1 and V2 for the automatic navigation. Relational queries without automatic navigation are much more complex; for comparison see the following equivalent SQL query:

```
select SUPPLIER.SNAME, sum(SP.QTY)
from SUPPLIER, PART, SP
where SUPPLIER.SNO = SP.SNO
and SP.PNO  = PART.PNO
and PART.PNAME = "bolt"
group by SP.SNO, SUPPLIER.SNAME
```

5 Conclusion

In the paper we presented viewers, a concept which can be considered a data-world analogue of procedure calls. Viewers make possible the representation of surprisingly many conceptual modelling issues in data-intensive applications. They can be implemented efficiently, yielding good performance, since in fact they are pointers. Simultaneously they simplify database queries due to automatic navigation. These factors motivate implementation of viewers together with related language functionalities in current or prototyped database systems. This idea is implemented in a prototype database programming system LOQIS. The implementation and further experiments convinced us that the idea of viewers is worth wider popularity in scientific and practical communities.

References

[1] C. Beeri, P.A. Bernstein, N. Goodman. A Sophisticate's Introduction to Database Normalization Theory. Proc. of 4th VLDB Conf., Berlin, Germany, pp.113-124, 1978

[2] J. Banerjee, W. Kim, H.J. Kim, H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. Proc. ACM SIGMOD Conf. pp.311-322, 1987.

[3] L. Cardelli. A Semantics of Multiple Inheritance. Information and Computation, 76, pp.138-164, 1988

[4] O. Deux et al. The Story of O_2. IEEE Transactions on Knowledge and Data Engineering, 2:1, pp.91-108, 1990.

[5] W. Kim, J.F. Garza, N. Ballou, D. Woelk. Architecture of the ORION Next-Generation Database System. IEEE Transactions on Knowledge and Data Enginering, Vol.2, No.1, 1990, pp.109-124

[6] F. Matthes, A. Ohori, J.W. Schmidt. Typing Schemes for Objects with Locality. Proc.1st Intl. East/West Database Workshop on Next Generation Information System Technology, Kiew, USSR 1990 Springer Lecture Notes in Computer Science, Vol.504, pp.106-123, 1991.

[7] D. Maier, J.D. Ullman, M.Y. Vardi. On the Foundations of the Universal Relation Model. ACM Transactions on Database Systems, Vol.9, No.2, pp.283-308, 1984

[8] F. Matthes, A. Rudloff, J.W. Schmidt, K. Subieta. The Database Programming Language DBPL, User and System Manual. FIDE, ESPRIT BRA Project 3070, Technical Report Series, FIDE/92/47, 1992

[9] A. Ohori, P. Buneman, V. Breazu-Tannen. Database Programming in Machiavelli - a Polymorphic Language with Static Type Inference. Proc. of ACM SIGMOD 89 Conf., 1989, pp.46-57

[10] J. Richardson, P. Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. Proc. of ACM SIGMOD 91 Conf., 1991, pp.298-307

[11] J.W. Schmidt, F Matthes. The Database Programming Language DBPL, Rationale and Report. FIDE, ESPRIT BRA Project 3070, Technical Report Series, FIDE/92/46, 1992

[12] M. Stonebraker, L.A. Rowe, M. Hirohama. The Implementation of POSTGRES. IEEE Transactions on Knowledge and Data Engineering, 2:1, pp.125-142, 1990.

[13] K. Subieta, M. Missala, K. Anacki. The LOQIS System. Institute of Computer Science Polish Academy of Sciences Report 695, 1990.

[14] K. Subieta. LOQIS: The Object-Oriented Database Programming System Proc.1st Intl. East/West Database Workshop on Next Generation Information System Technology, Kiew, USSR 1990 Springer Lecture Notes in Computer Science, Vol.504, pp.403-421, 1991.

[15] K. Subieta, C. Beeri, F. Matthes, J.W. Schmidt,. A Stack-Based Approach to Query Languages. Hamburg University, Department of Informatics, DBIS, unpublished report, 1993

[16] W. Wilkes. Instance Inheritance Mechanism for Object Oriented Databases. Proc. of Workshop on Object-Oriented Database Systems, Bad-Münster, Oct.1988. Springer Lecture Notes in Computer Science, Vol.334, pp.274-279, 1988.