PROPERTIES AND UPDATE SEMANTICS OF CONSISTENT VIEWS

G. Gottlob

Institute for Applied Mathematics
C.N.R., Genova, Italy

Computer Science Department
Stanford University

P. Paolini

Department of Electronics
Politecnico di Milano, Milan, Italy

A.R.G., Milano, Italy

R. Zicari

Department of Electronics
Politecnico di Milano, Milan, Italy

Electrical Engineering and
Computer Science Department
University of California, Berkeley

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **SEP 1985**	2. REPORT TYPE	3. DATES COVERED **00-00-1985 to 00-00-1985**
4. TITLE AND SUBTITLE **Properties and Update Semantics of Consistent Views**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		

14. ABSTRACT

We consider the problem of the translation of view updates to database updates. Our research uses an algebraic approach in order to classify different properties of views with respect to the treatment of updates. In our classification, special attention is paid to a class of views that we call "consistent". Informally speaking, a consistent view is a view with the following property: if the effect of a view update program on a view state is determined, then the corresponding database update is unambiguously determined. Thus, in order to know how to translate a given view update into a database update, it is not necessary to know the sequence of particular operations of the view update program; it is sufficient to be aware of a functional specification of such a program. We show how conditional updates can be modeled and prove that consistent views have a number of interesting properties with respect to the concurrency of update transactions. Moreover, we show that the class of consistent views includes, as a particular subset, the class of views which translate updates under maintenance of a constant complement. However, we give examples of important realistic views that are consistent but do not translate updates under constant complement. The results of Bancillion and Spyratos [ACM-TODS 6:4, 1981] are generalized in order to capture the update semantics of the entire class of consistent views.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**67**	

ABSTRACT

We consider the problem of the translation of view updates to database updates. Our research uses an algebraic approach in order to classify different properties of views with respect to the treatment of updates. In our classification, special attention is paid to a class of views that we call "consistent". Informally speaking, a consistent view is a view with the following property: if the effect of a view update program on a view state is determined, then the corresponding database update is unambiguously determined. Thus, in order to know how to translate a given view update into a database update, it is not necessary to know the sequence of particular operations of the view update program; it is sufficient to be aware of a functional specification of such a program. We show how conditional updates can be modeled and prove that consistent views have a number of interesting properties with respect to the concurrency of update transactions. Moreover, we show that the class of consistent views includes, as a particular subset, the class of views which translate updates under maintenence of a constant complement. However, we give examples of important realistic views that are consistent but do not translate updates under constant complement. The results of Bancilhon and Spyratos [ACM-TODS 6:4, 1981] are generalized in order to capture the update semantics of the entire class of consistent views.

KEYWORDS: DATABASE SYSTEMS, LOGICAL DESIGN, VIEW DEFINITION, UPDATE SEMANTICS, DATA ABSTRACTIONS, CONSISTENCY, CONCURRENCY, THEORY.

NOTE: This research has been partially sponsored by each of the following organizations:

- Defense Advanced Research Projects Agency (N39-84-C-211).
- Consiglio Nazionale delle Ricerche (C.N.R.) Italy.

University of California, Berkeley (MICRO grant)

1. INTRODUCTION

A view facility is an important part of many existing database systems, such as Query By Example [14], System R [10], Ingres [13], DB2 [19]. In such a context, the problem of update of views which are implemented on an underlying database, has been studied with different approaches [1-3, 5-9, 11, 12, 18, 22, 23]. Updates on views must be translated into updates on the underlying database. In general, there exists more than one database update that may correspond to the same view update. The problem is how to choose a view update, avoiding that the corresponding underlying database update may create inconsistencies or have side effects on the view. Starting with the work of Paolini and Pelagatti [5], it was acknowledged that a careful analysis of views and databases also needs to account for operations on views and databases, and not just for states, as it was traditionally done. A database can be described by the set of its possible legal states and by its operations. We model databases as algebras that we call **data abstractions**, that is a set of values and a set of operations to manipulate them. A view is a particular way of looking at a database and it can also be described by its states (which are, in general, different from the database states) and by its operations (which are in general different from the database operations).

In this paper, as in [7], both databases and views are defined as data abstractions. We distinguish between the notions of **static** and **dynamic** view. A static view of a database consists of a data abstraction and a mapping which establishes the correspondence between the database states and the view states. A dynamic view consists of a static view and an **update policy** which states how to translate view updates into database updates. In the paper, when we will use the general term "view", we mean "dynamic view".* Our research uses an algebraic approach in order to classify different properties of views with respect to the treatment of updates. In our classification, special attention is given to a class of views, that we call **consistent**. Informally speaking, a consistent view is a view with the following property: if the effect of a view update on a view state is determined, then the corresponding database update is unambiguously determined. Thus, in order to know, how to translate a given view-update program into a database update program, it is not necessary to know

* Some authors use the term "view" to designate static views [1].

the sequence of the single operations of the view update program, but it is sufficient to be aware of a **functional specification** of such a program.*

Bancilhon and Spyratos [1] propose an elegant solution to the view-update problem. They show that the choice of the database update policy can be made by deciding which portion of the database should remain invariant; this invariant portion is called the **constant complement**.

In our paper we show that the class of views treated by Bancilhon and Spyratos is a subclass of the consistent views. We show, however, that there are views of highly applicative importance which are consistent, but cannot be modeled by the approach of [1]. Consequently, we have extended the theory of Bancilhon and Spyratos to capture the update semantics of the much larger class of consistent views. Our approach uses a notion of complement as defined in [1]. However, we do not require that the information contained in the complement remain invariant, but we permit that updates may cause the loss of parts of this information.

The paper is organized as follows.

In section 2, we introduce our notation and define the basic concepts that will be used throughout the rest of the paper. Database and views are defined as data abstractions (algebras). In particular, among different other types of views, we introduce the notion of consistent view by using a purely algebraic definition. We also state a theorem which shows the relationship between the different types of views.

In section 3, a comprehensive example of a database with a number of views of different type is given. The example is discussed in detail since it will be used throughout the paper.

In section 4, we introduce the concept of **conditional update**. Informally, a conditional view update is executed depending on the result of the evaluation of a predicate on a view state. Conditional update enables us to define complex and realistic view update programs.

In section 5, we explain in more detail the importance of consistent views; we show the

* Our definition of "consistency" does not coincide with the one given in [1].

main properties of these views and give some examples of views which are not consistent. In particular, we define the concept of **functional equivalence** of update programs and show that for a consistent view, a pair of functionally equivalent view update programs are translated into a pair of functionally equivalent database update programs. Moreover, we analyze the properties of consistent views with respect to the concurrent execution of view transactions.

In section 6, we define the view update problem, with particular reference to the work of Bancilhon and Spyratos. We define the notion of complement, recall and reinterpret the results of Bancilhon and Spyratos and explain the limitation of their approach in modelling the update semantics of some views of the example given in section 3.

In section 7, the extension of the theory of Bancilhon and Spyratos to the class of all consistent views is carried out.

After giving a survey of related work in section 8, we conclude our paper with an overview and some comments on our results and we state our plans for future research.

2. NOTATION AND BASIC CONCEPTS

In this section we introduce our notation and state the basic definitions that will be used throughout the paper. In particular we will use the terminology and reintroduce a number of definitions of [7], where both database and views are defined as **data abstractions**, i.e. algebras.

A **data abstraction** is defined by a pair $D = \langle \overline{D}, \Sigma_D \rangle$, where $\overline{D}$ is the set of the possible **legal states** of the abstraction and Σ_D is the set of **operations** of the abstraction.

In our general model [7], the set of operations Σ_D is composed of query operations I_D and update operations U_D : $\Sigma_D = \langle I_D, U_D \rangle$. In this paper we are interested only in **update operations**, thus, for simplicity, we will disregard the query operations I_D and assume $\Sigma_D = U_D$. Our simplified model of a data abstraction therefore consists of a pair $D = \langle \overline{D}, U_D \rangle$, where $\overline{D}$ is the set of legal states and U_D is a set of update operations. Each update operation is a function from $\overline{D}$ in $\overline{D}$.

Given a set U_D of update operations on $\overline{D}$, it is easy to define the corresponding set of **update expressions**. The set E_D of update expressions is the collection of all possible compositions of update operations:

$\forall u \in U_D, u \in E_D$

$\forall u, v \in E_D, u \cdot v \in E_D$

no other elements are in E_D

Throughout the paper we assume that for each data abstraction D, U_D contains only **total operations**. A generalization of most of our results to partial operations is straightforward. The outline of such a generalization can be found in [7].

Some of the properties of data abstractions defined in [7] that will be used in this paper are now described.

DEFINITION 2.1. Given two data abstractions A and B, A **statically includes** B **through** α denoted by $A[\alpha] \Rightarrow B$ iff there exists a surjective function α from A to B which is called **abstraction function**.

The static inclusion is a necessary condition for having a data abstraction A as a possible representation for another data abstraction B. When $A[\alpha] \Rightarrow B$, the pair (α, B) is called a **static view** of A.

Regarding the sets of operations of two data abstractions, the basic notion is that of a translation function.

DEFINITION 2.2. A **translation function** τ from B to A is a function that maps operations of B into expressions of A:

$$\tau : U_B \rightarrow E_A.$$

The translation function τ is extended to the domain E_B as follows:

$$\forall u1,\ u2 \in E_B : \tau(u1 \cdot u2) = \tau u1 \cdot \tau u2.$$

Since τ is a function which maps operations to operations (also called "functional"), we adopt the usual mathematical notation and write τu instead of $\tau(u)$, whenever parenthesis are not needed to delimit the argument of τ.

The abstraction function α and the translation function τ are the basic notions to define the concept of **implementation** of a data abstraction.

DEFINITION 2.3. An abstraction A **implements** an abstraction B through α and τ, denoted by $A[\alpha, \tau] \Rightarrow B$ if and only if α is surjective and

$$\forall u \in U_B\ \forall b \in \overline{B}\ \forall a \in \overline{A} : b = \alpha(a) \Rightarrow u(b) = \alpha(\tau u(a)).$$

That is: α is a homomorphism with respect to τ.

When $A[\alpha, \tau] \Rightarrow B$, the triple $V = (\alpha, \tau, B)$ is called a **(dynamic) view** of A, and A is said to be the **base**. The elements of $\overline{A}$ are said to be **base states**; the elements of $\overline{B}$, are called **view states**. The members of U_A are called **base operations**; the elements of U_B are called **view operations**.

DEFINITION 2.4. Let $D = \langle \overline{D}, \Sigma_D \rangle$ denote a database. For static views $D[\alpha] \Rightarrow A$, we can define an equivalence relation $\underset{\alpha}{==}$ as follows:

$$\forall d1, d2 \in \overline{D} : d1 \underset{\alpha}{==} d2 \Leftrightarrow \alpha(d1) = \alpha(d2)$$

that is, $\underset{\alpha}{==}$ groups together the states of $\overline{D}$ which correspond by α to the same state of B.

For any data abstraction A, we can define two graphs:

DEFINITION 2.5. The **natural transition graph T_A** connects states of $\overline{A}$ as follows:

$$T_A(a1, a2) \Leftrightarrow (\exists u \in U_A : u(a1) = a2)$$

that is, a1 is connected to a2 iff there exists an update operation of A, which maps the first state into the second. T_A(a1,a2) in the rest of the paper will be denoted as $a1 \rightarrow a2$.

DEFINITION 2.6. The **natural connection graph C_A** of A is the transitive closure of T_A. The following is immediately to see:

$$C_D(a1,a2) \Leftrightarrow (\exists u \in E_A : u(a1) = a2)$$

that is, a1 is connected to a2 if there exists an update **expression** of D, which maps the first state into the second. C_A(a1,a2) in the rest of the paper will be denoted as $a1 \overset{*}{\longrightarrow} a2$. We will use the notation $a1 \overset{*}{\Longrightarrow} a2$ to express the disjunction : $(a1 \overset{*}{\longrightarrow} a2 \vee a1 = a2)$.

The definition of the two graphs are useful in order to characterize some properties of data abstractions.

DEFINITION 2.7. An abstraction A is **cyclic** iff

$$\forall a1, a2 \in \overline{A} : (a1 \overset{*}{\longrightarrow} a2 \Leftrightarrow a2 \overset{*}{\longrightarrow} a1)$$

that is, if state a1 can be updated to a2 then there also exists an inverse update expression which transforms a2 to a1.

Each of the above defined graphs T_D and C_D refers to a single data abstraction.

When an abstraction D (base) is used to implement an abstraction A (view), two other graphs are defined, which are useful to represent the effects of the translation of the operations of A into operations of D.

DEFINITION 2.8. Let D be a data abstraction and let $V = (\alpha, \tau, A)$ be a view of D. The **Transition Graph induced by V on D, T_{DV}**, is defined as follows:

$$T_{DV}(d1, d2) \Leftrightarrow \exists u \in U_A : \tau u(d1) = d2$$

that is, two states of the base D are connected by T_{DV}, iff there exists an update operation of the view V, the translation of which maps the first base state into the second. In the rest of the paper, $T_{DV}(d1, d2)$ will be denoted by d1 $-$v$\rightarrow$ d2.

DEFINITION 2.9. Let D be as in the above definition. The **Connection Graph induced by V on D, C_{DV}**, is defined as follows:

$$C_{DV}(d1, d2) \Leftrightarrow \exists u \in E_A : \tau u(d1) = d2$$

that is, two states of the base D are connected by C_{DV} iff there exists an update expression of the view, the translation of which maps the first base state into the second. In the rest of the paper, C_{DV}(d1,d2) will be denoted by d1 $-*$v$\rightarrow$ d2. Whenever we want to express the disjunction (d1 $-*$v$\rightarrow$ d2 $\vee$ d1 = d2) we will write d1 $=*$v$\Rightarrow$ d2.

A view $V = (\alpha, \tau, A)$ implemented on a base D may have some interesting properties with regard to its base. We will now define some of these properties using the transition and connection graphs induced by V on D.

DEFINITION 2.10. The view V **preserves connections** iff

$$\forall d1, d2, d2' \in \overline{D} : (d2 \underset{\alpha}{=\!=} d2' \wedge (d1 -\!*v\!\rightarrow d2 \wedge d1 -\!*v\!\rightarrow d2')) \Rightarrow d2 = d2'.$$

That is, from a given state of the base, it is not possible to reach in the connection graph induced by V on D two different states which represent the same state for the view.

DEFINITION 2.11. The view V **preserves loops** iff

$$\forall \mathrm{d1}, \mathrm{d2} \in \overline{\mathrm{D}} : (\mathrm{d1} \underset{\alpha}{==} \mathrm{d2} \wedge \mathrm{d1} \; {-}{*}\mathrm{v}{\rightarrow} \; \mathrm{d2}) \Rightarrow \mathrm{d1} = \mathrm{d2}.$$

This means: if an update expression leaves unchanged the state of the view (loop), then its translation leaves unchanged the base's state.

Note that the properties of preserving connections and of preserving loops do not imply one another. In figure 2.1 we show a database (with only three states) and a view (with two states) which preserves loops but not connections. Figure 2.2 shows a database with a view preserving connections but not loops.

DEFINITION 2.12. The view V is **consistent** iff V preserves connections and preserves loops.

LEMMA 2.1. V is consistent iff

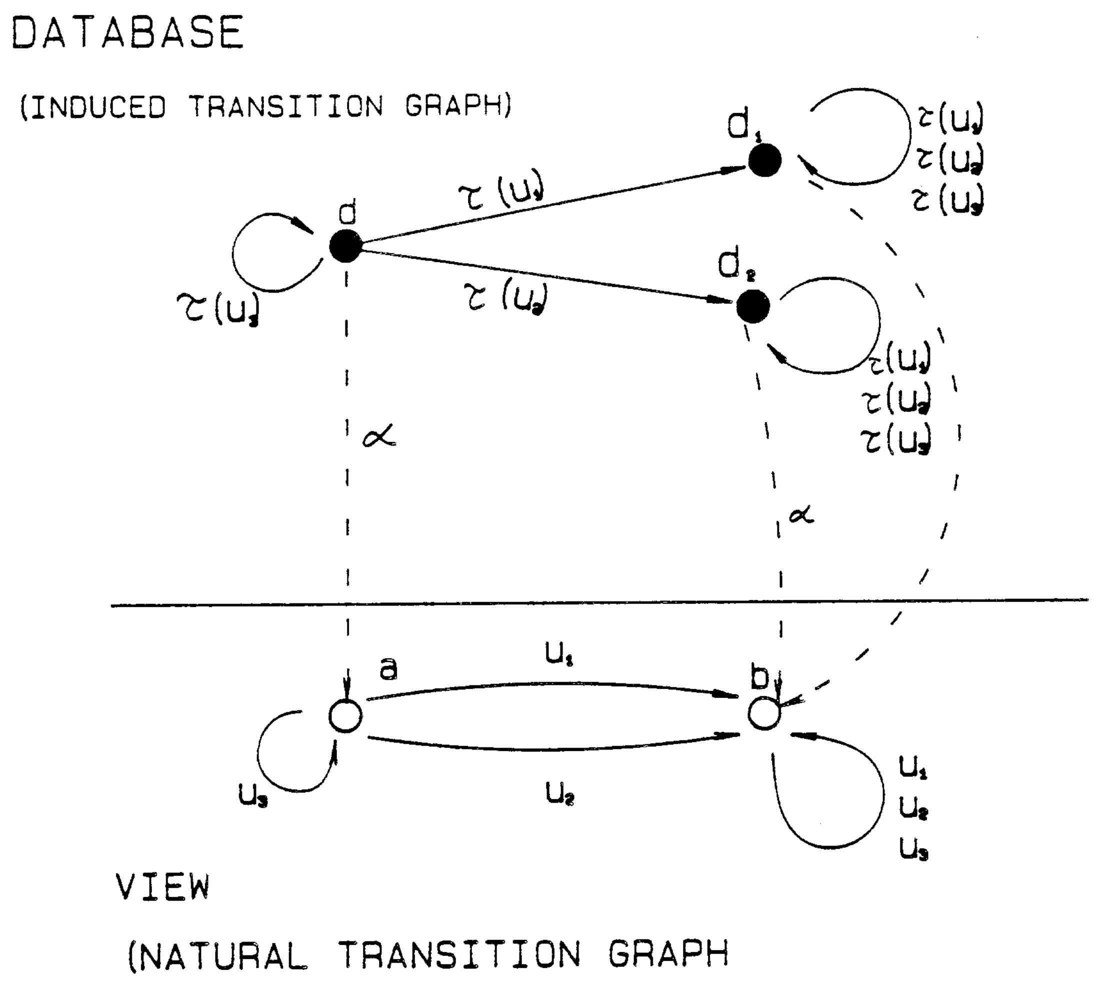
$$\forall \mathrm{d1}, \mathrm{d2}, \mathrm{d2}' \in \overline{\mathrm{D}} : (\mathrm{d2} \underset{\alpha}{==} \mathrm{d2}' \wedge \mathrm{d1} = {*}\mathrm{v} {\Rightarrow} \mathrm{d2} \wedge \mathrm{d1} = {*}\mathrm{v} {\Rightarrow} \mathrm{d2}') \Rightarrow \mathrm{d2} = \mathrm{d2}'.$$

PROOF. The easy proof of this lemma is omitted.

As we already stated in the introduction, the class of consistent views is of highly theoretic and applicative importance. Section 5 is dedicated to explain and prove the properties of consistent views. Let us, however, anticipate that if a view $\mathrm{V} = (\alpha, \tau, \mathrm{A})$, defined on a database D, is consistent then the following property holds:

$$\forall \xi_1, \xi_2 \in \mathrm{E_A}, \; \forall \mathrm{a} \in \overline{\mathrm{A}}, \; \forall \mathrm{d} \in \overline{\mathrm{D}} :$$
$$((\alpha(\mathrm{d}) = \mathrm{a} \wedge \xi_1(\mathrm{a}) = \xi_2(\mathrm{a})) \Rightarrow \tau\xi_1(\mathrm{d}) = \tau\xi_2(\mathrm{d})).$$

Informally, this property can be interpreted as follows: If we are able to give a functional specification of a complex view update, say by a function F which maps each view state



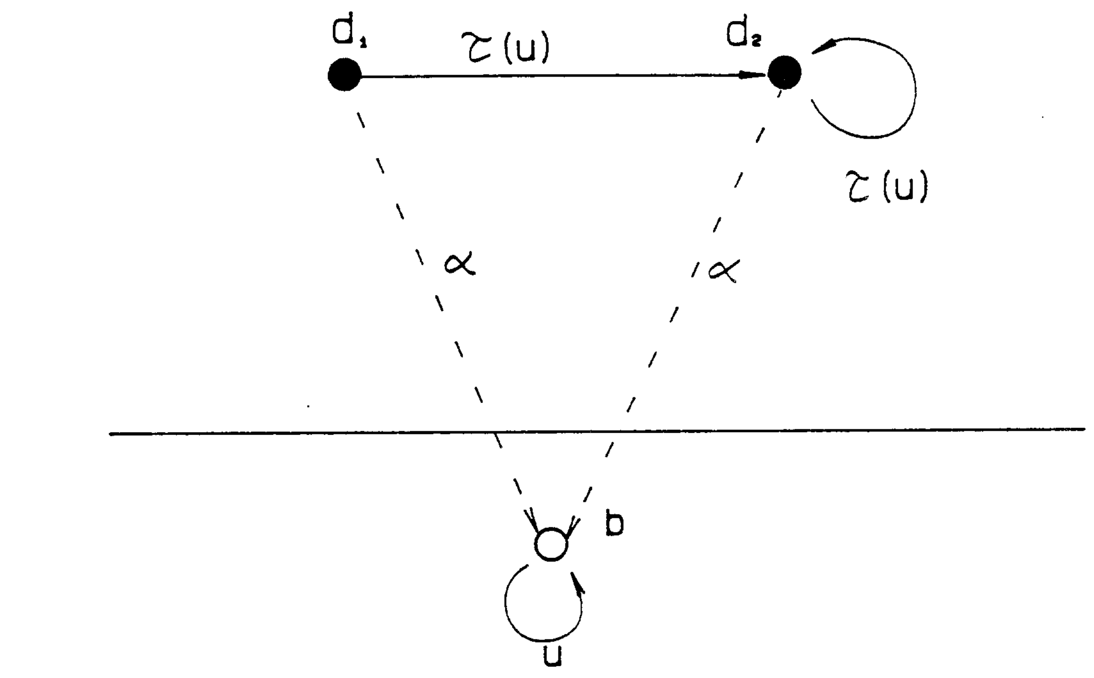
LEGEND:

●	DATABASE STATES
○	VIEW STATES
$\xrightarrow{u_i}$	UPDATE OPERATIONS ON THE VIEW
$\xrightarrow{\tau(u_i)}$	TRANSLATED UPDATE OPERATIONS
$\xrightarrow{\alpha}$	ABSTRACTION FUNCTION

<u>FIGURE 2.1.</u> A VIEW WHICH PRESERVES LOOPS BUT DOES NOT PRESERVE CONNECTIONS

DATABASE

(INDUCED CONNECTION GRAPH)



VIEW

(CONNECTION GRAPH)

LEGEND:	●	DATABASE STATES
	○	VIEW STATES
	$\xrightarrow{u_i}$	UPDATE OPERATIONS ON THE VIEW
	$\xrightarrow{\tau(u)}$	TRANSLATED UPDATE OPERATIONS
	$\xrightarrow{\alpha}$	ABSTRACTION FUNCTION

<u>FIGURE 2.2.</u> A VIEW WHICH PRESERVES CONNECTIONS BUT DOES NOT PRESERVE LOOPS

$a \in \overline{A}$ into an updated state $F(a) \in \overline{A}$, then all possible view update expressions $\xi_1, \xi_2, \xi_3, \ldots$ which implement precisely the function F have equivalent translations under τ. A functional specification of a view update is therefore sufficient to define the translation of the update. It is therefore not necessary to inspect the particular sequence of operations of the view update program.

Clearly, consistency is a very desirable property which makes a view much more accessible for theoretic analysis and more tractable for practical management.

The above quoted property will be proved in a more general form in section 5 not only for view update expressions, but for the more powerful class of "view update programs". Let us now proceed by defining another important property of views.

DEFINITION 2.13. A view V **partitions** the base D iff it is possible to decompose $\overline{D}$ into a family of nonempty sets $D1, \ldots, Dk$, with $i \neq j \Rightarrow Di \cap Dj = \phi$ and $D1 \cup D2 \cup \ldots \cup Dk = \overline{D}$, such that the following properties a) and b) hold:

a) $(i \neq j \wedge di \in Di \wedge dj \in Dj) \Rightarrow \neg(di =\!*\!v\!\Rightarrow dj)$

b) $\forall 1 \leq i \leq k : (d1, d2 \in Di \wedge d1 \neq d2) \Rightarrow \alpha(d1) \neq \alpha(d2)$.

LEMMA 2.2. If a view partitions the base then it is consistent.

PROOF. Assume the view $V = (\alpha, \tau, A)$ partitions the base D but is not consistent. Then, by Lemma 2.1, there exist $d1$, $d2$, $d2' \in \overline{D}$, such that $d1 \underset{\alpha}{=\!=} d2 \wedge d1 =\!*\!v\!\Rightarrow d2 \wedge d1 =\!*\!v\!\Rightarrow d2' \wedge d2 \not\to d2'$. From Definition 2.13.a it follows that d1, d2 and d2′ are all in the same component, say Di. But this is in contradiction to Definition 2.13.b, since there it is required $\alpha(d1) \neq \alpha(d2)$.

Q.E.D.

LEMMA 2.3. If a view is cyclic and preserves loops then it partitions the base.*

PROOF. Assume that $V = (\alpha, \tau, A)$ is a cyclic and loop preserving view of the base D. It is easy to see that the relation $=\!*v\!\Rightarrow$ then becomes an equivalence relation on $\overline{D}$. Let D1, D2, ..., Dk denote the equivalence classes with respect to $=\!*v\!\Rightarrow$, i.e. the elements of the quotient algebra $\overline{D}/ =\!*v\!\Rightarrow$. We will show, that D1, ..., Dk form precisely a partition into subsets of $\overline{D}$ as required by definition 2.13. Property a) of definition 2.13 holds trivially by the definition of D1, ..., Dk. Assume that property b) does not hold. Then there exists a Di and d1, d2 $\in$ Di such that d1 $\neq$ d2 and $\alpha(\text{d1}) = \alpha(\text{d2})$. From d1 $=\!*v\!\Rightarrow$ d2 it then follows that there exists an update expression $u \in E_A$, such that $u(\alpha(\text{d1})) = \alpha(\text{d1})$ and $\tau u(\text{d1}) = \text{d2}$. But then, since V preserves loops, it must hold d1 = d2, which is a contradiction.

Q.E.D.

Note that a view, which partitions the base is not necessarily cyclic (examples are given in the next section). Therefore we will also consider the class of views which are both cyclic and partition the base (this class is a proper subclass of the class of views which partition the base).

In figure 2.3 we give an overview of the inclusion relationship between the different classes of views defined in this section.

We can summarize our results about the classifications of views in the following theorem:

THEOREM 2.1.

a) If V partitions the base then V is consistent (Lemma 2.2).

b) If V is consistent then V preserves loops (Def. 2.12).

* In [7], a property "V slices the base" has been defined, which is more general than the property "V partitions the base". In the present paper, the slicing property will not be used.

c) If V is consistent, then V preserves connections (Def. 2.12).

d) If V is cyclic and preserves loops then V partitions the base (Lemma 2.3).

A pictorial representation of this theorem is given in figure 2.4. This figure also helps to find all transitive consequences of the assertions stated in Theorem 2.1, such as for example: If V is cyclic and preserve loops then V preserves connections.

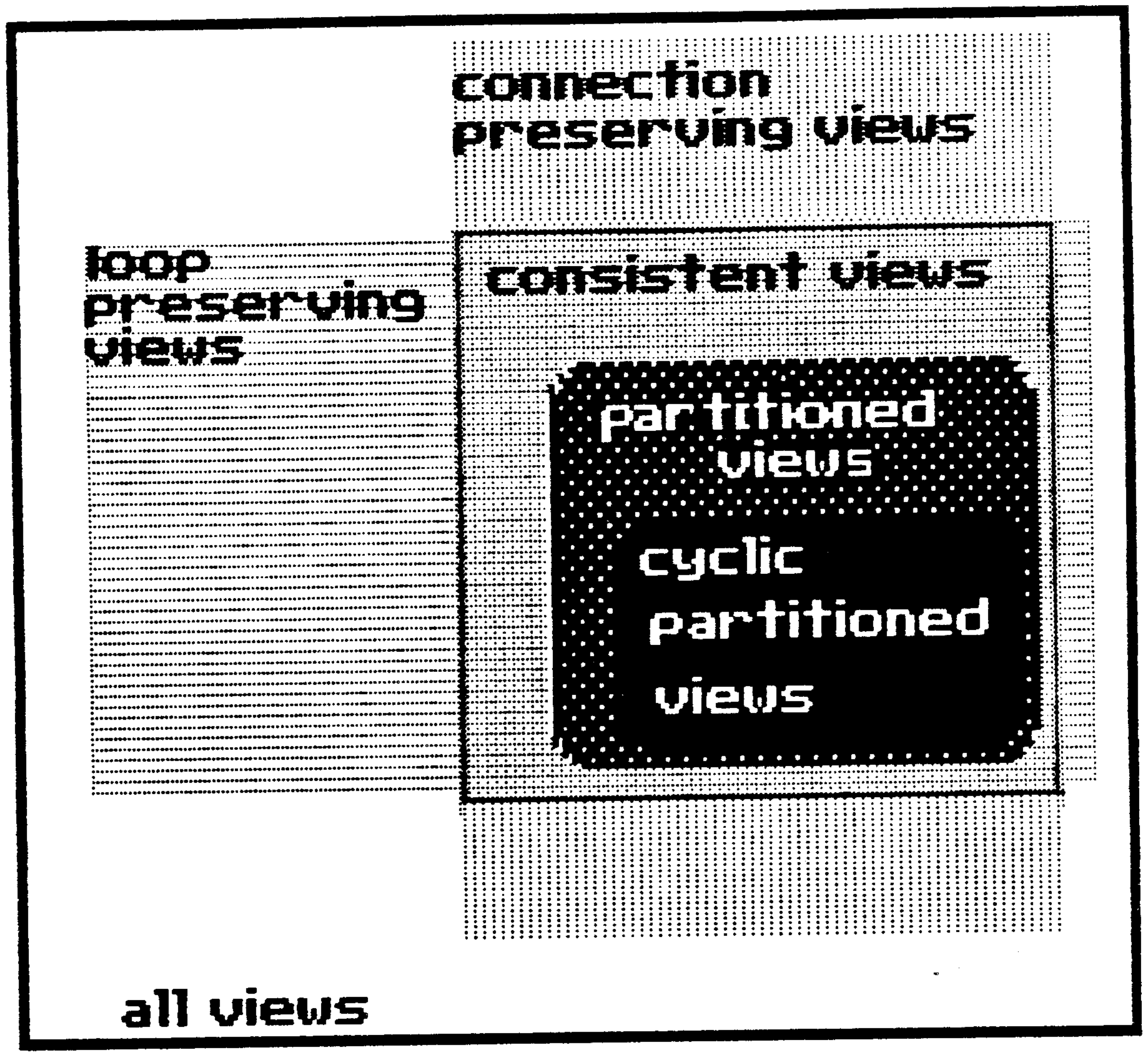

connection preserving views
loop preserving views
consistent views
partitioned views
cyclic partitioned views
all views

FIGURE 2.3

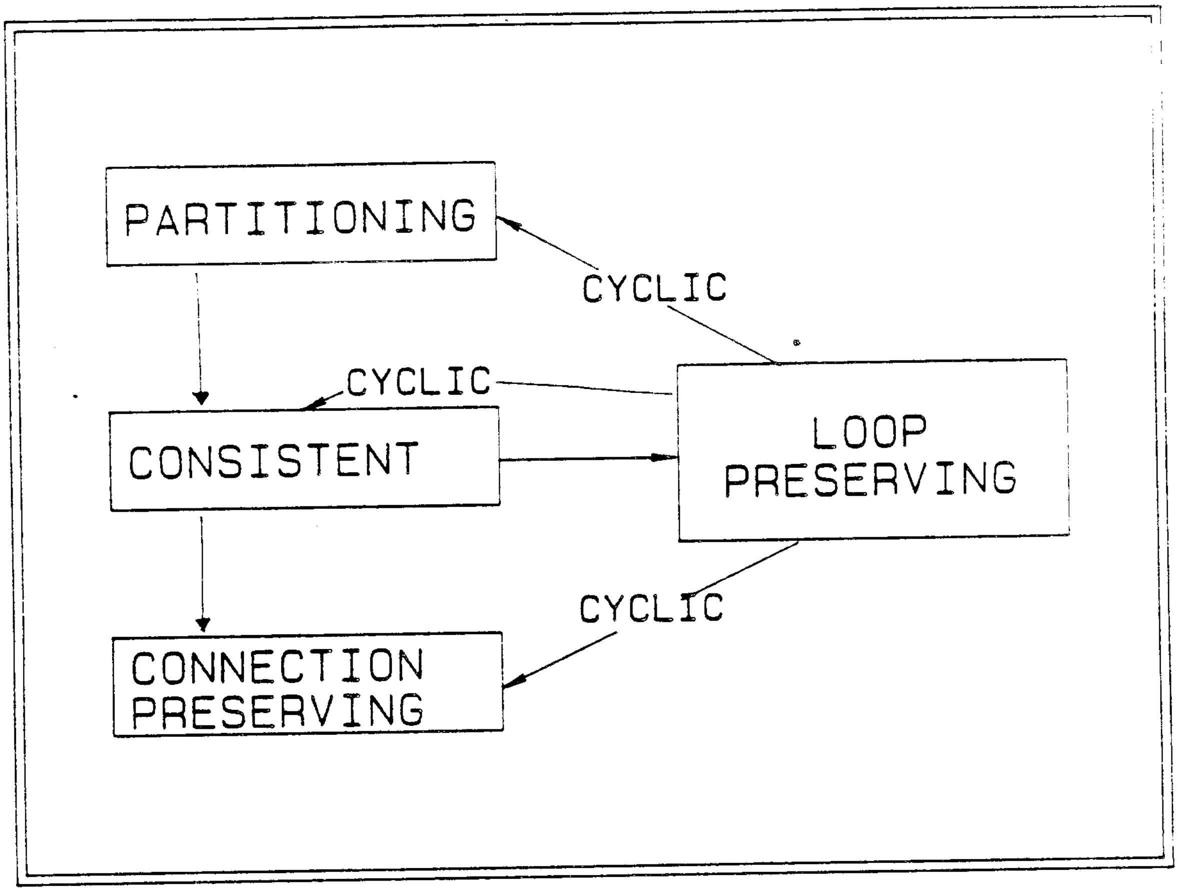

PARTITIONING
CYCLIC
CYCLIC
CONSISTENT
LOOP PRESERVING
CYCLIC
CONNECTION PRESERVING

FIGURE 2.4.

3. A COMPREHENSIVE EXAMPLE

This section is entirely dedicated to the presentation of a comprehensive example of a database on which different views are defined. In the next sections we will use this example to emphasize some problems and to illustrate our results.

In the following, we assume that the reader is familiar with the elementary concepts of the relational data model [17], [24]. In particular, we use the following notation: Let R be a relation, t ∈ R a tuple of R and A an attribute of R, then t.A denotes the value of the tuple t for the attribute A. The projection of the relation R to attribute A is denoted by R[A]. A tuple of a relation is explicitly referenced by enclosing its attribute values in angle brackets, e.g. ⟨i, j, k⟩.

Consider a relational database STORE, which maintains information about the type, name and price of different products that are for sale in a store and about the single pieces of each product which are available in the warehouse.

The database STORE consists of two relations BILL and PIECES with the following schemas:

BILL : (PRODUCT#, NAME, PRICE)

PIECES : (PRODUCT#, PIECE#, BROKEN)

The single attributes have the following meaning:

PRODUCT# :	an integer from 1 to 100 identifying a product-type.
NAME :	the name of a product.
PRICE :	the selling price for one piece of a product.
PIECE# :	a number identifying each single piece available in the magazine.
BROKEN :	a boolean value indicating whether the piece is broken or not.

We assume furthermore that the database is subject to the following integrity constraints:

a) (PRODUCT#, PIECE#) is a key for the relation PIECES.

b) PRODUCT# is a key for the relation BILL.

c) $\forall t \in \text{BILL}(1 \leq \text{t.PRODUCT\#} \leq 100) \wedge \forall 1 \leq i \leq 100\ \exists t \in \text{BILL}\ (\text{t.PRODUCT\#} = i)$. This means, that BILL always contains 100 rows, one for each product-type.

d) $\forall t \in \text{PIECES}\ \exists t' \in \text{BILL}(\text{t.PRODUCT\#} = \text{t}'\text{.PRODUCT\#})$. This means, that for each piece, the corresponding product is described in the relation BILL. Note that the converse is not necessarily true: For some product description in BILL, there might exist no piece in PIECES.

$\overline{\text{STORE}}$ is the set of all legal states of the database. A database state $d \in \overline{\text{STORE}}$ consists of a pair (b, p), where b and p are legal instances (with respect to the above integrity constraints) of BILL and PIECES. Furthermore, let $\overline{\text{PIECES}}$ denote the set of all legal states of the relation PIECES:
$\overline{\text{PIECES}} = \{b \mid (b, p) \in \text{STORE}\}$;
and let $\overline{\text{BILL}}$ denote the set of all legal states of the relation BILL :
$\overline{\text{BILL}} = \{p \mid (b, p) \in \text{STORE})$.

We assume that all computable database operations, such as insertion, deletion and updating of tuples, as well as conditional combinations thereof are allowed update operations on our database. Let Ω denote the set of all these operations, then our database is given by the data abstraction STORE = $\langle \overline{\text{STORE}}, \Omega \rangle$. Note,that since Ω is defined as the complete set of all possible operations, the natural transition graph T_{STORE} connects every pair of states of the base set STORE in both directions (this holds also for the natural connection graph C_{STORE}, which, in this case, is equal to T_{STORE}).

Let us now describe six views V1, ..., V6 of this database. As defined in the previous section, each view consists of a triple (α, τ, A), where α is the abstraction function

$$\alpha : \overline{\text{STORE}} \rightarrow \overline{\text{A}},$$

which associates to each legal state of the STORE database a state of the view; A is a data abstraction $(\overline{\text{A}}, \text{U}_\text{A})$, where $\overline{\text{A}}$ is the set of view states and U_A is the set of update operations defined on $\overline{A}$; τ is the translation of each view update operation into an update expression of the database STORE.

VIEW 1: The Acquisition view $V1 = (\alpha_1, \tau_1, (\overline{A}_1, U_1))$

description: This view "sees" at each moment the present state of the relation PIECES. The store can acquire new pieces of each product, by issuing the operation BUY[x, y, z], where x designates the product number identifying the product type, y the piece number of the new piece and z indicates whether the new piece is broken or not (the view thus also consents to acquire broken pieces). Formally, for each different triple (x, y, z), BUY[x, y, z] is a different view update operation.

Abstraction function α_1 and view state set $\overline{A}_1$:
Let $d = (b, p) \in$ STORE; $\alpha_1(d) = \alpha_1(b, p) = p$.

$\overline{A}_1 = \overline{\text{PIECES}}$

Set of update operations U_1:

$$U_1 = \{\text{BUY}[x,\ y,\ z] \mid x, y \in \text{INTEGER} \wedge 1 \leq x \leq 100 \wedge (z = \text{true} \vee z = \text{false})\},$$

such that for all relational instances $p \in \overline{\text{PIECES}}$:

$$\text{BUY}[x,\ y,\ z](p) = \textit{if } \langle x, y\rangle \in p[\text{PRODUCT\#},\ \text{PIECES\#}] \textit{ then } p \textit{ else } p \cup \{\langle x, y, z\rangle\}.$$

Note: INTEGER denotes a *finite* range of positive integers.

Translation function τ_1:

Let $d = (b, p) \in$ STORE.

$$\tau_1\text{BUY}[x,\ y,\ z](b,\ p) = (b,\ \text{BUY}[x,\ y,\ z](p)).$$

Thus the translation leaves the BILL relation invariant.

VIEW 2: The Repair Service $V2 = (\alpha_2, \tau_2, (\overline{A}_2, U_2))$

description: This view serves to alter the information about the *state* of each piece available in the magazine. A piece is broken or intact according to the boolean value of its attribute "BROKEN". Only the PIECES relation is visible to the view V2 (this is just as for V1). There are two types of operations : BREAK[x, y] is issued when piece number y of product type x breaks during the storing phase. REPAIR[x, y] is issued, if piece y of product x was broken and has been repaired.

Abstraction function α_2 and view state set $\overline{\mathrm{A}}_2$:

$$\alpha_2 = \alpha_1; \quad \overline{\mathrm{A}}_2 = \overline{\mathrm{A}}_1 = \overline{\mathrm{PIECES}}.$$

Set of update operations U_2:

$$\mathrm{U}_2 = \{\mathrm{BREAK}[x, y], \mathrm{REPAIR}[x, y] \mid x, y \in \mathrm{INTEGER} \wedge 1 \leq x \leq 100\},$$

such that for all relational instances $p \in \overline{\mathrm{PIECES}}$:

$$\mathrm{BREAK}[x, y](p) = \textit{if } \langle x, y, \mathrm{false}\rangle \in p \textit{ then } (p - \{\langle x, y, \mathrm{false}\rangle\}) \cup \{\langle x, y, \mathrm{true}\rangle\}$$
$$\textit{else } p.$$

$$\mathrm{REPAIR}[x, y](p) = \textit{if } \langle x, y, \mathrm{true}\rangle \in p \textit{ then } (p - \{\langle x, y, \mathrm{true}\rangle\}) \cup \{\langle x, y, \mathrm{false}\rangle\}$$
$$\textit{else } p.$$

Translation function τ_2:

Let $d = (b, p) \in \overline{\mathrm{STORE}}$.

$\tau_2\mathrm{BREAK}[x, y](b, p) = (b, \mathrm{BREAK}[x, y](p))$.

$\tau_2\mathrm{REPAIR}[x, y](b, p) = (b, \mathrm{REPAIR}[x, y](p))$.

Thus, as for view V1, the translation leaves the BILL relation invariant.

VIEW 3: The Inventory $\mathrm{V3} = (\alpha_3, \tau_3, (\overline{\mathrm{A}}_3, \mathrm{U}_3))$

description: This view is defined exactly in the same way as View 2, with the only difference that the REPAIR operation does not exist, thus a transition between two states due to the BREAK operation is irreversible in view V3.

Abstraction function α_3 and view state set $\overline{\mathrm{A}}_3$:

$$\alpha_3 = \alpha_2; \quad \overline{\mathrm{A}}_3 = \overline{\mathrm{A}}_2 = \overline{\mathrm{PIECES}}.$$

Set of update operations U_3:

$U_3 = \{\text{BREAK}[x, y] \mid x, y \in \text{INTEGER} \wedge 1 \leq x \leq 100\}$, where BREAK[x, y] is defined as for view V2.

Translation function τ_3:

Let $d = (b, p) \in \overline{\text{STORE}}$.

$$\tau_3\text{BREAK}[x, y](b, p) = (b, \text{BREAK}[x, y](p)).$$

VIEW 4: The vendor's view $V4 = (\alpha_4, \tau_4, (\overline{A}_4, U_4))$

description: This view describes the information visible to a vendor. We assume that a vendor has to know the description and price of each product (i.e.the complete information contained in the relation BILL), as well as the number of intact pieces available for each product. The latter is an aggregate of information contained in the relation PIECES. We assume, however, that the vendor is *not concerned* with piece numbers or with broken pieces. All this information is hidden to the vendor. The vendor therefore sees the BILL relation, extended by a field QTY_AVAILABLE which for each product reports the number of not broken pieces available.

In order to sell one piece of product i, the vendor issues the view operation SELL[i]. The number of available pieces (i.e. the value of the field QTY_AVAILABLE) then is decremented by 1. If there are no available pieces for a specific product i (i.e. QTY_AVAILABLE = 0 in row i), then the operation SELL[i] has no effect.

There are many different possibilities to translate the view update operation SELL[i] into a database update, since in general, for product i there will exist many available pieces, one of which should be eliminated from the database. For view V4 we have chosen the following update policy: from all intact pieces, eliminate the one with the *lowest* piece number.

This choice may be motivated as follows: Assume that pieces of each product are acquired by ascending piece numbers,* then, at each time a product is sold, the piece with the longest storing duration of this product is taken from the warehouse. This FIFO policy is very useful in practice, especially when a single piece's value decrements as storage time increases.

Abstraction function α_4 and view state set $\overline{A}_4$:

Let d = (b, p) be a database state. $\alpha_4(d) = \alpha_4(b, p) = s$, where s is an instance of the schema

$$S : (\text{PRODUCT\#}, \text{NAME}, \text{PRICE}, \text{QTY\_AVAILABLE})$$

such that PRODUCT# is a key of s, s[PRODUCT#, NAME, PRICE] = b and $\forall t \in s$: t.QTY_AVAILABLE = card({x | x ∈ p ∧ x.PRODUCT# = t.PRODUCT# ∧ x.BROKEN = false}).

$$\overline{A}_4 = \{s \mid \exists d \in \text{STORE} : s = \alpha_4(d)\}.$$

Set of update operations U_4:

$U_4 = \{\text{SELL}[i] \mid i \in \text{INTEGER} \land 1 \leq i \leq 100\}$, where SELL[i] is defined as follows:

Let $s \in \overline{A}_4$ and let $t_i \in s$ denote the unique tuple of s with t_i.PRODUCT# = i, then

$$\begin{aligned} \text{SELL}[i](s) = {} & \textit{if } t_i.\text{QTY\_AVAILABLE} > 0 \textit{ then} \\ & (s - \{t_i\}) \cup \{\langle i, t_i.\text{NAME}, t_i.\text{PRICE}, t_i.\text{QTY\_AVAILABLE} - 1\rangle\} \\ & \textit{else } s. \end{aligned}$$

Translation function τ_4:

Let $d = (b, p) \in \overline{\text{STORE}}$; Define for each i a set Min[i](p) by:

$$\text{Min}[i](p) = \{t \in p \mid t.\text{PRODUCT\#} = i \land t.\text{BROKEN} = \text{false} \land t.\text{PIECE\#} = \text{minimum}\}.$$

* This means that for two elements of the same product type, the one with higher piece number has been acquired after the one with the lower piece number.

Note that Min[i](p) is either a singleton or the empty set.

We define the translation function τ_4 by:

$$\tau_4\text{SELL}[i](b,\ p) = (b,\ p\text{–Min}[i](p)).$$

VIEW 5: The alternative vendor's view V5 = $(\alpha_5, \tau_5, (\overline{A}_5, U_5))$

description: This view is equal to view V4 with the only difference that another update policy (translation function τ_5) is used: When the operation SELL is issued, instead of deleting the piece with the lowest piece number, this time, the piece with the highest piece number is deleted (i.e. sold). This policy corresponds to a LIFO strategy.

Abstraction function α_5 and view state set $\overline{\mathbf{A}}_5$:

$$\alpha_5(d) = \alpha_4(d) \quad \text{and} \quad \overline{A}_5 = \overline{A}_4.$$

Set of update operations U_5:

$$U_5 = U_4.$$

Translation function τ_5:

Let $d = (b, p) \in \overline{\text{STORE}}$; Define for each i a set Max[i](p) by:

$$\text{Max}[i](p) = \{t \in p \mid t.\text{PRODUCT\#} = i \wedge t.\text{BROKEN} = \text{false} \wedge t.\text{PIECE\#} = \text{maximum}\}.$$

Note that Max[i](p) is either a singleton or the empty set.

The translation function τ_5 is defined by:

$$\tau_5\text{SELL}[i](b,\ p) = (b,\ p\text{–Max}[i](p)).$$

VIEW 6: The Price Setting view V6 = $(\alpha_6, \tau_6, (\overline{A}_6, U_6))$

description: Through this view, the store's manager has the possibility to modify the selling price of each single product. Only the relation BILL is visible to the user of V6.

For each product x and for each (representable) integer y, an operation SETPRICE[x, y] is defined, which sets the PRICE component of the record describing product x to value y.

Abstraction function α_6 and view state set $\overline{A}_6$:

Let d = (b, p) ∈ $\overline{\text{STORE}}$.

$$\alpha_6(d) = \alpha_6(b,p) = b; \quad \overline{A}_6 = \text{BILL}.$$

Set of update operations U_6:

$$U_6 = \{\text{SETPRICE}[x,\ y] \mid x, y \in \text{INTEGER} \land 1 \leq x \leq 100\},$$

where SETPRICE[x, y] is defined as follows: let b ∈ $\overline{A}_6$, and let t denote the tuple of b with t.PRODUCT# = x; then

$$\text{SETPRICE}[x,\ y](b) = (b - \{t\}) \cup \{\langle x, t.\text{NAME}, y\rangle\}.$$

Translation function τ_6:

Let d = (b, p) ∈ $\overline{\text{STORE}}$.

$$\tau_6\text{SETPRICE}[x,\ y](b,\ p) = (\text{SETPRICE}[x,\ y](b),\ p).$$

Note that among all views that we have defined, V6 is the only one which allows to modify the BILL part of the database STORE.

Let us now classify the different views of this example according to the properties defined in the previous section.

1. V1 is **not cyclic**, since there exists no operation (or expression) which could act as an inverse of the BUY operations. The BUY operations irreversibly add tuples to the view. The view V1 **partitions the base**: each class D_i corresponds to a different state b_i of the relation BILL. For fixed b_i, let

   $$D_i = \{\langle b_i, p\rangle \mid \langle b_i, p\rangle \in \overline{\text{STORE}}\}.$$

   For this partition, it is easy to verify, that the properties required by definition 2.13 hold.

2. V2 is **cyclic** and **partitions the base**. The classes D_i of the partition are as for the view V1.

3. V3 is **not cyclic**, but **partitions the base**. Also for this view, the classes D_i are the same as for the view V1.

4. V4 is **not cyclic** and **does not partition the base**. Nevertheless V4 is **consistent**.

   The fact that V4 does not partition the base can be shown by a counterexample. Figure 3.1 shows two states d1 = (b1, p1) and d2 = (b2, p2) of STORE, as well as a third database state d. It clearly holds α_4(d1) = α_4(d2). Consider the operation τ_4SELL[1]. It holds: τ_4SELL[1](d1) = τ_4SELL[1](d2) = d. Assume, that V4 be partitioned. Since d1 $\underset{\alpha_4}{==}$ d2, these two states must belong to two different classes Di and Dj of the partition (d1 ∈ Di and d2 ∈ Dj with i ≠ j). Since τ_4SELL[1](d1) = d, it holds d1 =*V4⇒ d, therefore, by property a) of definition 2.13, it must hold: d ∈ Di. On the other hand, from τ_4SELL[1](d2) = d, we immediately deduce d2 =*V4⇒ d, and therefore, d ∈ Dj. This is in contradiction to the disjointness of the sets Di and Dj (Def. 2.13). Therefore V4 is not partitioned.

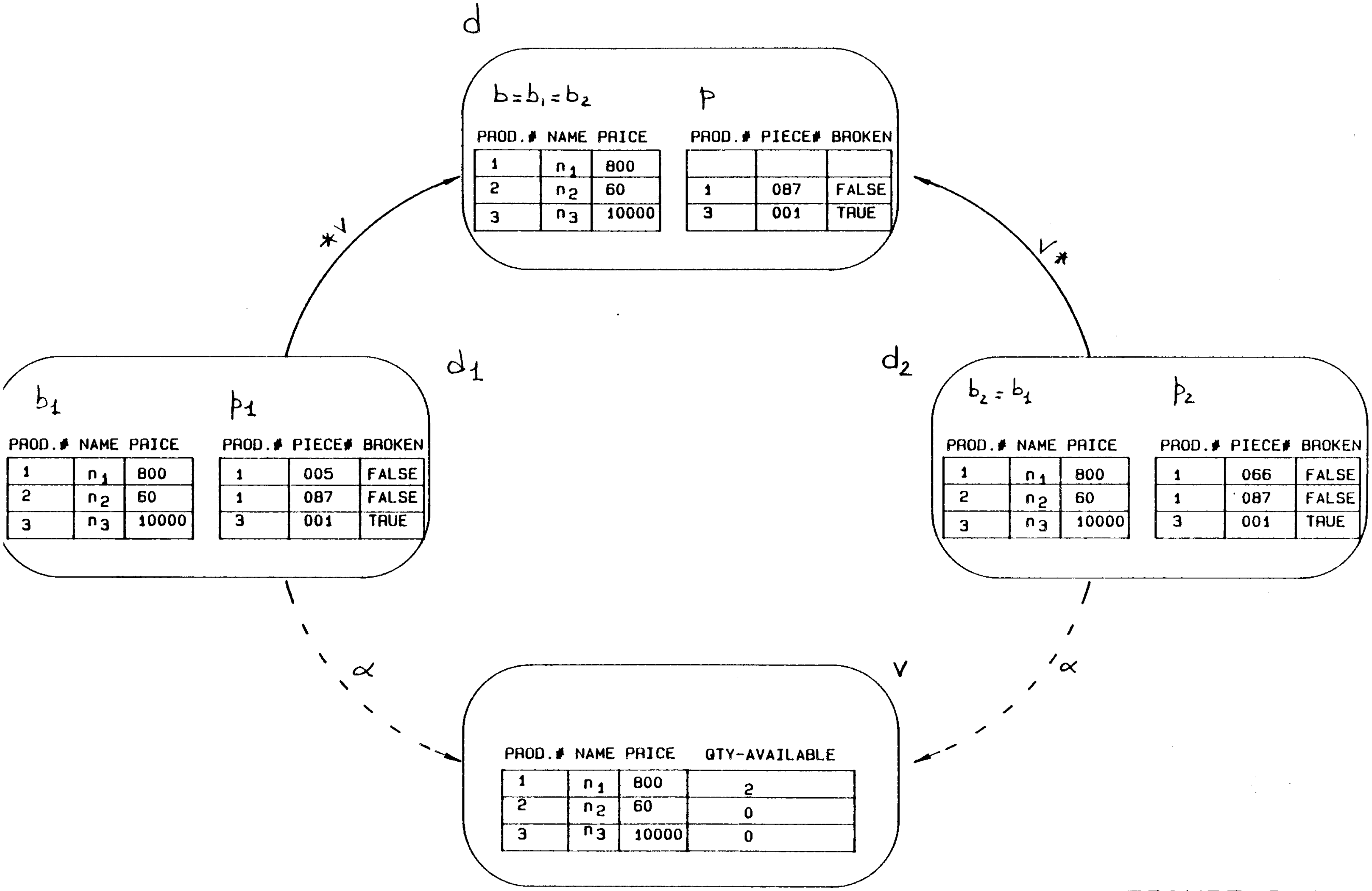
   The **consistency** of V4 will be proved formally later in this paper. However, let us explain informally, why V4 is consistent. If we know the effect of a view update program on a particular view state, than we know **how many** pieces of each product have been sold. By the particular update policy V4 (translation function τ_4) we then also know exactly **which** pieces of each product have been sold: namely those with the lowest PIECE# (provided they are not broken). Thus any functional specification of a view update program (expression) unambiguously determines the translation of this program into a database update program. As we have already remarked, this property is equivalent to consistency.

5. V5, for similar reasons as V4 is **consistent**, but **does not partition the base** and **is not cyclic**.

6. V6 is clearly **cyclic** and **partitions the base**. The single classes of the partition are

correspond to different instances of the PIECES relation . Every different instance of this relation determines a different class Di.

The properties of the views V1, . . . , V6 are summarized in figure 3.2.



$b = b_1 = b_2$

PROD.#	NAME	PRICE
1	n_1	800
2	n_2	60
3	n_3	10000

p

PROD.#	PIECE#	BROKEN
1	087	FALSE
3	001	TRUE

b_1

PROD.#	NAME	PRICE
1	n_1	800
2	n_2	60
3	n_3	10000

p_1

PROD.#	PIECE#	BROKEN
1	005	FALSE
1	087	FALSE
3	001	TRUE

$b_2 = b_1$

PROD.#	NAME	PRICE
1	n_1	800
2	n_2	60
3	n_3	10000

p_2

PROD.#	PIECE#	BROKEN
1	066	FALSE
1	087	FALSE
3	001	TRUE

v

PROD.#	NAME	PRICE	QTY-AVAILABLE
1	n_1	800	2
2	n_2	60	0
3	n_3	10000	0

FIGURE 3.1.

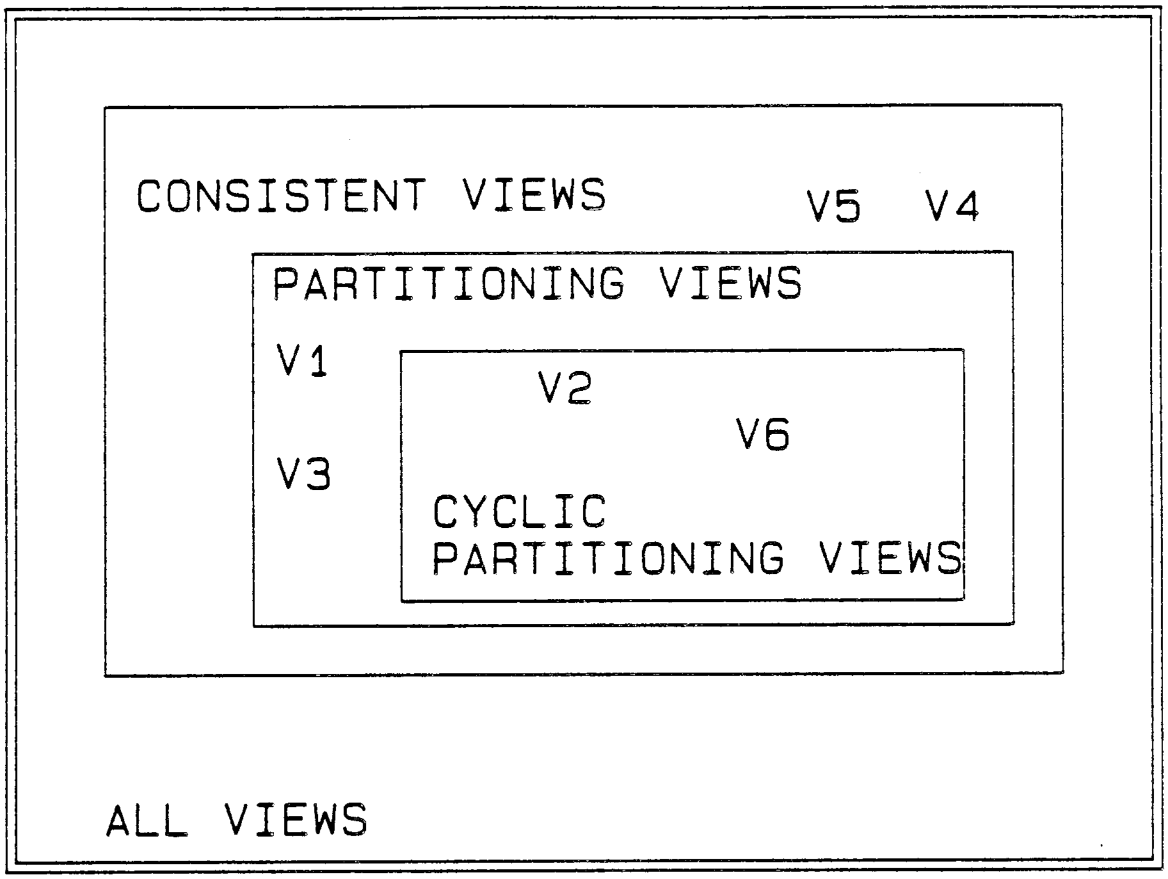

CONSISTENT VIEWS
V5
V4
PARTITIONING VIEWS
V1
V3
V2
V6
CYCLIC PARTITIONING VIEWS
ALL VIEWS

FIGURE 3.2.

4. CONDITIONAL VIEW UPDATES

We have shown in the last section that our algebraic model of databases and views can be used to represent quite realistic situations. However, it is not difficult to see, that the choice of the set U_A of update operations of view $V = (\alpha, \tau, (\overline{A}, U_A))$ is critical in the sense that it may impose a serious limit on the representability of view update programs (also view transactions).

Consider for example the view V_4 of section 3. Suppose that a client wants to buy three pieces of product 5, but if there are less pieces available, the client prefers not to make the deal and to look for another company. It is clear what the vendor has to do, in order to satisfy such a "conditional" request. He first must inspect the present state s of his view and read the number of available items for product 5. If the number is not less than 3 he will issue the expression

$$\mathrm{SELL}[5](\mathrm{SELL}[5](\mathrm{SELL}[5](s))).$$

If the number of available items is less than 3, then the vendor leaves the view unchanged. The overall transaction can be called a **conditional update**. We are not able to represent a conditional update through the update operations or update expressions of our view V_4. Of course, we could add a set of new operations to U_4, say $\{\mathrm{SELL}[n,m]\}$ which perform exactly the task of our conditional update, namely to "sell" n pieces of product m, only if n pieces of this product are available. However, this would be a very application dependent modification of the operation set U_4.

Remember that for each update operation set U, we have a corresponding set of update **expressions** E (Section 2). We prefer to extend the set of update expressions E to a set $E^{\dagger} \supset E$, such that $E^{\dagger}$ contains **all** possible conditional updates that can be built from the "primitive" update expressions of E.

A necessary concept for the formalization of conditional updates is the notion of **predicate**. A conditional update is executed depending on the result of the evaluation of a predicate on a view state. If $A = (\overline{A}, U_A)$ is a data abstraction, then P_A denotes the set of decidable predicates over $\overline{A}$. We do not undertake the tedious task of defining formally what a decidable predicate on elements of A is. The syntactical form of such predicates would depend heavily on the formalism chosen for representing or manipulating the elements of $\overline{A}$ (for example the relational model). Semantically speaking, P_A consists of all predicates p that allow to distinguish a particular subset $S \subseteq \overline{A}$ among others, such that for all elements $s \in S$, $p(s)$ is true, but for all elements $s' \in \overline{A} - S$, $p(s')$ is false. The number of semantically different predicates in P_A is thus essentially $2^{|\overline{A}|}$ (we suppose that $\overline{A}$ is a finite set). Clearly, in most formalisms the same predicate can be syntactically represented in many (possibly infinite) different ways. Rather than prescribing a specific formalism, we leave the syntax of predicates to the designer of a particular database system. When we use predicates without naming a particular state, to which predicates are applied, we will use Church's Lambda Notation for representing the predicate. For example the predicate which is true iff the state to which it is applied is equal to state "a" can be denoted by

$$\lambda x(x = a).$$

In this expression, x is a **placeholder** for the argument. If this predicate is applied to a specific state c, the expression "$\lambda x(x = a)$ (c)" is equivalent to "$c = a$".

The next definition introduces more formally the basic concepts needed in order to deal with conditional updates.

DEFINITION 4.1. Let $A = (\overline{A}, U_A)$ be a data abstraction.

a) id_A denotes the identity on $\overline{A}$, $P_A$denotes the set of all evaluable predicates on $\overline{A}$

b) Let us define the new compound operation "if $[p, e_1, e_2]$" with the following semantics:

$$\forall a \in A : \text{if } [p, e_1, e_2](a) = \text{ if } p(a) \text{ then } e_1(a) \text{ else } e_2(a).$$

The set $Prog_A$ of **Update Programs** on $\overline{A}$ is recursively defined as follows:

- $id_A \in Prog_A$
- $\forall e \in E_A$: $e \in Prog_A$

- $\forall e_1, e_2 \in \mathrm{Prog}_A$, $\forall p \in P_A$: if $[p,e_1,e_2] \in \mathrm{Prog}_A$.
- $\forall\, e_1, e_2 \in \mathrm{Prog}_A$: $e_1 \cdot e_2 \in \mathrm{Prog}_A$ (composition of operations)
- Prog_A does not contain any other operations

c) A **Conditional View Update** is a view update program which contains a subexpression of the form if $[p, e_1, e_2]$.

EXAMPLE. Consider the view V_4 of Section 3. Assume a vendor wants to sell three pieces of product 5 if the number of available pieces of product 5 is equal to 3 or greater than 3. Let avail(s,n) be a function which returns for each view state s the number of available pieces for product n. The conditional view update then can be formulated as follows:

$$\text{if } [\lambda x(\text{avail}(x,5) \geq 3), \text{SELL}[5]\text{SELL}[5]\text{SELL}[5], \text{id}_{A_4}].$$

Since we extended the set E_A of view update expressions to the set Prog_A of view update programs, we have to extend the domain of the translation function τ to Prog_A. In order to do this, it is sufficient to specify, how id_A and how conditional view updates are translated.

DEFINITION 4.2. (extension of τ to Prog_A)

Let $V = (\alpha, \tau, A)$ be a view of D and let $A = (\overline{A}, U_A)$. We extend $\tau : E_A \rightarrow E_D$ to a function $\tau : \mathrm{Prog}_A \rightarrow \mathrm{Prog}_D$ as follows:

a) $\tau(\mathrm{id}_A) = \mathrm{id}_D$

b) $\forall p \in P_A$, $\forall e_1, e_2 \in \mathrm{Prog}_A$: $\tau(\text{if } [\lambda x p(x), e_1, e_2]) = \text{if } [\lambda x p(\alpha(x)), \tau(e_1), \tau(e_2)]$.
Note that $\lambda x p(\alpha(x))$ is a predicate on $\overline{D}$.

c) $\forall e_1, e_2 \in \mathrm{Prog}_A$: $\tau(e_1 \cdot e_2) = \tau(e_1) \cdot \tau(e_2)$.

It is easy to see, that this is the most natural and probably the only "reasonable" extension of τ to Prog_A.

For a database designer who wants to specify a translation function τ, it is thus sufficient that he specifies the translation of each primitive update in U_A. By definition 4.2 this function τ is unambiguously extended to the set Prog_A of all possible view update programs.

DEFINITION 4.3. Let $V = (\alpha, \tau, A)$ be a view of D with $A = (\overline{A}, U_A)$.

We define the **conditional closure** V^+ of V as follows:

V^+ is a view of D;

$V^+ = (\alpha^+, \tau^+, A^+)$ where

$A^+ = (\overline{A}, \mathrm{Prog}_A)$

$\alpha^+ = \alpha$

$\tau^+ = \tau$ extended to Prog_A as in definition 4.2

Note that it holds $V^{++} = V^+$.

V^+ is the view which has as primitive operations all possible update programs of V.

In the following we will show that V^+ is consistent iff V is consistent. This is a useful result. It can be interpreted as follows: The addition of conditional updates does not alter the consistency state of view. It is therefore always sufficient to consider a view with primitive (non conditional) operations for a consistency proof. View update programs can be used without the danger of introducing some hidden inconsistency.

Before showing our result, let us prove a lemma.

LEMMA 4.1. Let $V = (\alpha, \tau, A)$ be a view of D, then the two relations $=*v\Rightarrow$ and $=*v^+\Rightarrow$ defined on $\overline{D}$ are the same.

PROOF. We have to show, that for all d, d' $\in \overline{D}$:

$$(d =*v\Rightarrow d') \Leftrightarrow (d =*v^+\Rightarrow d')$$

"$\Rightarrow$" this part of the proof is trivial, since $E_A \subseteq \mathrm{Prog}_A$.

"$\Leftarrow$" this part of the proof can be done by induction on the number of "if's" that occur in an update program of V. We only state the main idea of the proof: Let d $=*v^+\Rightarrow$ d'; then there exists an update program g $\in$ Prog_A such that $\tau g(d) = d'$. If g does not contain any conditional subexpression, then $g = id_A$ or $g \in E_A$. If $g = id_A$ then $\tau g = id_D$ and $d = d'$; in this case d $=*v\Rightarrow$ d' holds trivially. If $g \in E_A$ then d $=*v\Rightarrow$ d' by definition of " $=*v\Rightarrow$ ". If g contains some conditional subexpressions, then consider $g(\alpha(d))$. It is easy to show that $g(\alpha(d)) = \alpha(d')$. Now consider the series of update expressions ei that occur in the program g and that are effectively applied during the evaluation of $g(\alpha(d))$. Combine those expressions in the right order into one single expression e. Clearly $e \in E_A$ and it holds $e(\alpha(d)) = \alpha(d')$. By how τ is extended from E_A to Prog_A (def. 4.2) we also know that $\tau g(d) = \tau e(d) = d'$. From $\tau e(d) = d'$ we immediately conclude d $=*v\Rightarrow$ d'. □

THEOREM 4.1. Let V be a view of D. V is consistent iff V^+ is consistent.

PROOF. V and V^+ have the same abstraction function α (def. 4.3). The relations $=*v\Rightarrow$ and $=*v^+\Rightarrow$ on $\overline{D}$ coincide. In Lemma 2.1 we showed that the consistency of a view V depends only on α and on $=*v\Rightarrow$. The theorem follows immediately. □

Before concluding this section, let us state a lemma, which will be used in section 5.

LEMMA 4.2. Let $V = (\alpha, \tau, A)$ be a view of D. V^+ is consistent iff V^+ is connection preserving.

PROOF. It suffices to show that if V^+ preserves connections then V^+ also preserves loops. Remember that V^+ contains id_A as operation and that $\tau(id_A) = id_D$. Suppose that V^+ is connection preserving but not loop preserving. It follows by definition 2.11 that there exist elements $d_1, d_2 \in \overline{D}$, such that $\alpha(d_1) = \alpha(d_2)$ and $d_1 -*v^+\rightarrow d_2$ and $d_1 \neq d_2$. Since $id_D(d_1) = d_1$ it also holds $d_1 -*v^+\rightarrow d_1$. This is in contradiction with the assumption that V^+ is connection preserving. □

5. PRESERVATION OF PROGRAM EQUIVALENCE AND CONCURRENCY

The concept of view update program which was introduced in the last section is at the base of some fundamental properties of consistent views. In this section we show that consistent views have a number of important preservation properties. We will first consider the **equivalence** of update programs. We show that a view is consistent if and only if its translation function τ translates any pair of functionally equivalent view update programs into a pair of functionally equivalent database update programs. Later we shall draw our attention to the preservation of **concurrency** properties. We will define the concept of **transaction** and we will show that the translation function τ preserves commutativity of transactions, non-interference and serializability.

5.1 PROGRAM EQUIVALENCE

Let us now give a formal definition of functional equivalence of update programs.

DEFINITION 5.1. Let e_1 and e_2 be two update programs of a data abstraction A $= (\overline{A}, U_A)$. e_1 and e_2 are **functionally equivalent** iff

$$\forall a \in A \quad e_1(a) = e_2(a).$$

If e_1 and e_2 are functionally equivalent, we write $e_1 \equiv e_2$.

DEFINITION 5.2. Let $V = (\alpha, \tau, A)$ be a view of D. τ **preserves equivalence** iff

$$\forall e_1, e_2 \in \mathrm{Prog}_A : (e_1 \equiv e_2) \Rightarrow (\tau(e_1) \equiv \tau(e_2)).$$

If for a view $V = (\alpha, \tau, A)$, τ preserves equivalence, then we also say: V preserves equivalence.

THEOREM 5.1. A view is equivalence preserving iff it is consistent.

PROOF.

"if" This part of the theorem follows easily from the definition of consistency.

"only if" Assume $V = (\alpha, \tau, A)$ is equivalence preserving but not consistent. By lemma 5.2, V^+ is not connection preserving. Therefore there exist elements d, d_1, $d_2 \in \overline{D}$, such that

$$\begin{aligned} & d \ {-*}\mathrm{v}^+{\rightarrow}\ d_1 \\ \text{and} \quad & d \ {-*}\mathrm{v}^+{\rightarrow}\ d_2 \\ \text{and} \quad & \alpha(d_1) = \alpha(d_2) \\ \text{and} \quad & d_1 \neq d_2 \end{aligned}$$

From this we infer that there exist two different database update programs h_1 and h_2, $h_1, h_2 \in \mathrm{Prog}_D$, such that $h_1(d) = d_1$ and $h_2(d) = d_2$ and such that h_1 and h_2 are the translations by τ of two view update programs, respectively e_1 and e_2:

$$h_1 = \tau(e_1)$$
$$h_2 = \tau(e_2).$$

Clearly h_1 and h_2 are not functionally equivalent, as they have different images in d.

Notice that for e_1 and e_2 it holds:

$$e_1(\alpha(d)) = \alpha(d_1) = \alpha(d_2) = e_2(\alpha(d)).$$

(This equation follows from the fact that τ is a translation function).

Although e_1 and e_2 are equivalent with respect to the single view state $\alpha(d)$, it does not necessarily hold that e_1 and e_2 are functionally equivalent. They might differ with respect to other view states. We therefore construct the following view update programs for f_1 and f_2:

$$f_1 = \text{if } [\lambda x\,(x = \alpha(d)), e_1, id_A]$$
$$f_2 = \text{if } [\lambda x\,(x = \alpha(d)), e_2, id_A]$$

It is clear that f_1 and f_2 are functionally equivalent view update programs.

Let us examine their translations:

$$\tau f_1 = \text{if}\left[\lambda x\left(\alpha(x) = \alpha(d)\right), h_1, id_D\right]$$

$$\tau f_2 = \text{if}\left[\lambda x\left(\alpha(x) = \alpha(d)\right), h_2, id_D\right]$$

Clearly τf_1 and τf_2 are not functionally equivalent since their images in d differ. Therefore V is not equivalence preserving. □

Theorem 5.1 states that the class of consistent views is exactly the class of views for which a functional specification of a view update is sufficient in order to functionally determine its translation. This property is of highly practical relevance. It ensures that a translation of a view update program is essentially independent of the particular sequence of the single operations of the program, and that it only depends on the **semantics** of the view update.

As long as update programs are atomic actions, functionally equivalent update programs might be interchanged without any problem. Therefore a view programmer who operates on a consistent view might change the structure of his view update program, for example in order to perform some optimization. As long as he does not change the semantics, the view update program needs not to be retranslated into a new database update program.

This high level of independence between view and database makes a database system much more accessible for theoretical analysis. Correctness proofs and verification of global transactions (possibly involving different views) can be done by using formal specification techniques.

At the end of this section, we will give an example of a view which is not consistent and we will show that in this case equivalent view update programs are translated into database updates which are not equivalent.

5.2 CONCURRENCY

Let us now draw our attention to the concurrency problem. Control over concurrency is one of the most important features of advanced database management systems [28,29,30], but the interaction between views and concurrency has hardly ever been analyzed.

There are three main questions arising when the interaction between views and concurrency control is considered:

1) How can a set of concurrent transactions, operating on the same view, be correctly translated to a set of (hopefully) concurrent transactions on the base?

2) How can a set of concurrent transactions on different views be correctly translated to a set of (hopefully) concurrent transactions on the base?

3) What is the interaction (or side effect) between transactions of different views implemented on the same base?

In this paper we limit ourselves to consider the first question. We will show that the transaction function of consistent views preserves commutativity, non-interference and serializability of transactions.

Before we can proceed, we have to define our concept of **transaction** and the above mentioned properties of pairs of transactions.

Since we are mainly interested in translating **updates**, we define the concurrency properties of two transactions with respect to the effect that these updates have on their data abstraction (either view or database) and not with respect to the outputs that the transactions may issue to the view user.

DEFINITION 5.3. Let $A = (\overline{A}, U_A)$ be a data **abstraction.**

a) A **transaction** T on A is a finite series of update programs p_i: $T = [p_1, p_2, \cdots, p_n]$, where $p_i \in Prog_A$, for $1 \leq i \leq n$. The single programs p_i are also called **atomic actions** of T and it is assumed that each p_i be performed "atomically" and without the parallel interaction of any other programs.

If T is a transaction then T° denotes the view update program which is the result of the composition of all atomic actions of T:

$$T^\circ = p_1 \cdot p_2 \cdots p_n$$

If $T_1 = [p_1, \dots, p_n]$ and $T_2 = [p'_1, \dots, p'_m]$, then $T_1 \cdot T_2$ is defined as:

$$T_1 \cdot T_2 = [p_1, p_2, \dots, p_n, p'_1, \dots, p'_m]$$

Note that it holds $(T_1 \cdot T_2)^\circ \equiv T_1^\circ \cdot T_2^\circ$.

b) Two transactions T_1 and T_2 are **commutative** iff it holds:

$$(T_1 \cdot T_2)^\circ \equiv (T_2 \cdot T_1)^\circ.$$

A **mix** (also "schedule") of a pair of transactions $T_1 = [p_1, \dots, p_n]$, $T_2 = [p'_1, \dots, p'_m]$ is a transaction $T = [p''_1, \dots, p''_{n+m}]$ such that each p''_i belongs either to T_1 or to T_2 and

$$\forall 1 \leq i < j \leq n+m, \quad \forall 1 \leq h, k \leq n:$$
$$(p''_i = p_h \in T_1 \wedge p''_j = p_k \in T_1) \Rightarrow h < k$$
$$\forall 1 \leq i < j \leq n+m, \quad \forall 1 \leq h, k \leq m:$$
$$(p''_i = p_h \in T_2 \wedge p''_j = p_k \in T_2) \Rightarrow h < k.$$

$\mathrm{Mix}(T_1, T_2)$ denotes the set of all transaction mixes of T_1 and T_2.

Two transactions T_1 and T_2 are **non-interfearing** (or **completely concurrently executable**) if the following property holds:

$$\forall T \in \mathrm{Mix}(T_1, T_2) : T^\circ \equiv (T_1 \cdot T_2)^\circ.$$

Note that if two transactions are non interfearing, they are also commutative, since $T_2 \cdot T_1$ represents a particular mix of T_1 and T_2.

Let T be a mix of T_1 and T_2. T is said to be **serializable** if it holds

$$T^\circ \equiv (T_1 \cdot T_2)^\circ \vee T^\circ \equiv (T_2 \cdot T_1)^\circ.$$

Note that the concepts of non interference and serizability are easily generalizable to sets of more than two transactions. For the sake of simplicity, we just consider pairs of transactions in this paper.

c) Let $V = (\alpha, \tau, (\overline{A}, U_A))$ be a view of D and let $T = [p_1, \dots, p_n]$ be a transaction on

A. We then define the translation τT of T to D as follows:

$$\tau(T) = [\tau(p_1), \ldots, \tau(p_n)].$$

Note that it holds $\tau(T^\circ) = (\tau(T))^\circ$.

d) The translation function τ of V is

- **commutativity preserving** iff any pair of commutative transactions T_1, T_2 is translated by τ into a pair of commutative transactions $\tau(T_1)$, $\tau(T_2)$ on D.

- **concurrency preserving** iff any pair T_1, T_2 of non-interfearing transactions is translated by τ into a pair of non-interfearing transactions $\tau(T_1)$, $\tau(T_2)$ on D.

- **serializability preserving** iff any serializable mix T of transactions T_1, T_2 is translated by τ into a serializable mix $\tau(T)$ of $\tau(T_1)$ and $\tau(T_2)$. Whenever τ has a preservation property, we also say that V has this property, for instance if τ is commutativity preserving, then we say that V is a commutativity preserving view.

THEOREM 5.3. Any consistent view is

a) commutativity preserving

b) concurrency preserving

c) serializability preserving.

PROOF. Let $V = (\alpha, \tau, A)$ be a consistent view of D. By theorem 5.1, we know that V is equivalence preserving.

a) Assume that T_1 and T_2 are two commutative view transactions: $(T_1 \cdot T_2)^\circ \equiv (T_2 \cdot T_1)^\circ$. Therefore it must hold

$$\tau((T_1 \cdot T_2)^\circ) \equiv \tau((T_2 \cdot T_1))^\circ.$$

Thus

$$\tau(T_1^\circ \cdot T_2^\circ) \equiv \tau(T_2^\circ \cdot T_1^\circ)$$

and hence

$$\tau(T_1^\circ) \cdot \tau(T_2^\circ) \equiv \tau(T_2{}^\circ) \cdot \tau(T_1^\circ)$$

and

$$\tau(T_1)^\circ \cdot \tau(T_2)^\circ \equiv \tau(T_2)^\circ \cdot \tau(T_1)^\circ$$

and finally

$$(\tau(T_1) \cdot \tau(T_2))^\circ \equiv (\tau(T_2) \cdot \tau(T_1))^\circ$$

thus $\tau(T_1)$ and $\tau(T_2)$ are commutative transactions on D.

Q.E.D

b) Assume that $T_1 = [p_1, \ldots, p_n]$ and $T_2 = [p'_1, \ldots, p'_m]$ are two non-interfearing transactions on V. We have to show that τT_1 and τT_2 are non interfearing transactions on D. Let T′ be an arbitrary mix of τT_1 and τT_2. T′ is of the form $[\tau p''_1, \ldots, p''_{n+m}]$ where $p''i \in T_1$ or $p''i \in T_2$ for $1 \leq i \leq n+m$. Let us define a transaction T on A as follows:

$$T = [p''_1, \ldots, p''_{n+m}].$$

Clearly T is a mix of T_1 and T_2 and it holds $\tau T = T'$. Since T_1 and T_2 are not interfearing, we have $T \equiv T_1 \cdot T_2$. Since V is equivalence preserving, it immediately follows: $\tau T \equiv \tau(T_1 \cdot T_2)$ which can be rewritten as: $T' \equiv \tau(T_1 \cdot T_2)$. Since $\tau(T_1 \cdot T_2) \equiv \tau(T_1) \cdot \tau(T_2)$ we finally get $T' \equiv \tau(T_1) \cdot \tau(T_2)$.

Thus $\tau(T_1)$ and $\tau(T_2)$ are non-interfearing.

Q.E.D.

c) Let T_1, T_2 be transactions on V and let T be a serializable mix of T_1 and T_2. Assume without loss of generality that $T \equiv T_1 \cdot T_2$. It then follows

$$\tau(T) \equiv \tau(T_1 \cdot T_2) \equiv \tau(T_1) \cdot \tau(T_2).$$

Therefore $\tau(T)$ is serializable with respect to $\tau(T_1)$ and $\tau(T_2)$.

Q.E.D

□

Notice that the properties of commutativity, non-interference and serializability are defined independently of particular operations, such as, for example, lock and release operations of differerent granularities.

Lock and release primitives can be modeled explicitely as particular elements of the set of operations U_A of a data abstraction A. Clearly the static data model (the set $\overline{A}$) has to be predisposed for locking. For example, if the possibility of write-locks at the granularity of relations is desired, then to each datarelation of the static model should be attached a boolean value with the function of "locking flag". Locking and releasing then can be modeled by setting and resetting this flag. Update operations in such a model are programs which are conditional on the value of locking flags.

Note that the preservation of commutativity, concurrency and serializability are necessary but not sufficient conditions for the consistency of a view. It is possible to construct "pathological views", which are for instance commutativity preserving but not consistent.

We conclude this section with a brief example of a non consistent view.

Consider the database of section 3. Let view V7 be as view V5 with additional operations BUY[i] defined as follows:

View operations:	BUY[i] increments the QTY-AVAILABLE value for product i by 1.
Translation:	τ(BUY[i]) = $\overline{\text{BUY}}$[i] adds a new tuple for product i to the PIECES relation. Let d be a database state. Let $\max(i, x)$ be a function which for a database state x returns the highest value of PIECE# that exists in state x for product i (if there is no piece for product i then $\max(i, x) = 0$). The $\overline{\text{BUY}}$[i] operation applied to a database state d then adds the tuple (i, max(i,d) + 1, false) to the PIECES relation.

Consider the database and view state depicked in figure 3.1 and the following two view update programs p_1 and p_2 for V7:

$$p_1 : \text{if } [\lambda x(\text{avail}(1,x) \geq 1), \text{SELL}[1]\text{BUY}[1], \text{id}_{A7}]$$

$$p_2 : \text{if } [\lambda x(\text{avail}(1,x) \geq 1), \text{BUY}[1]\text{SELL}[1], \text{id}_{A7}]$$

where avail(n,x) is a function which returns the number of available pieces for product n and for view state x..

Clearly p_1 and p_2 are functionally equivalent view updates. Unfortunately this does not hold for the translation: Consider the database state d_2 of figure 3.1. $\tau(p_1)$ will first delete the piece with number 087 from the database and then introduce a new piece of product1 with piece number 067. $\tau(p_2)$ first introduces a new piece with number 088 for product number 1 and then deletes this piece. Therefore our view V7 is not equivalence preserving and thus not consistent.

It is easy to see that V7 does not preserve commutativity; consider for example the two transactions

$$T_1 = [\text{SELL}[1]]$$

$$T_2 = [\text{BUY}[1]].$$

6. THE TRANSLATION OF VIEW UPDATES

In this section we will discuss some aspects of the well known view update problem. Let us first reformulate this problem, using the notation and formalism introduced in section 2.

Let $D = (\overline{D}, U_D)$ be a database, let $A = (\overline{A}, U_A)$ be a data abstraction such that there exists a surjective function $\alpha: \overline{D} \rightarrow \overline{A}$. The pair (α, A) then constitutes a **static view** of D (def. 2.1). The view update problem can be stated by means of the following three questions:

1. Is it possible to find a translation function τ, such that the triple (α, τ, A) is a (dynamic) view ?

2. Which conditions on α and on U_A must be satisfied in order to guarantee that a translation τ exists?

3. If there are different possibilities to choose a translation τ, how can we characterize the different possible choices?

The pioneer work in solving the view update problem has been carried out by F. Bancilhon and N. Spyratos. In their paper [1] it is shown how the choice of an update policy τ can be made by deciding which portions of the database should remain constant (complement). Choosing a complement that remains invariant under all translated operations assigns unambiguous semantics to a view update. Formally, a complement of the static view (α, A) can be defined as a mapping β as follows:

DEFINITION 6.1. Let (α, A) be a static view of a database $D = (\overline{D}, U_D)$. Let β be a surjective function from $\overline{D}$ onto a set $\overline{B}$, $\beta: \overline{D} \rightarrow \overline{B}$. β is a complement of α iff

$$\forall d, d' \in \overline{D} : \alpha(d) = \alpha(d') \Rightarrow \beta(d) \neq \beta(d').$$

It is easy to see that, if β is a complement of α, then the knowledge of $\alpha(d)$ and of $\beta(d)$ uniquely determines the database state d. Thus β is indeed a mapping "complementary" to α, and the knowledge of both, α and β, is sufficient for computing the database.

In [1], the property of a set of update operations $U \subseteq U_A$ to be **translatable under constant complement** β is referred to as the β-translatability of U.

DEFINITION 6.2. U is **β-translatable** iff $\forall d \in \overline{D}, \forall u \in U \; \exists v \in E_D \; \exists d' \in \overline{D}$:

$$(v(d) = d' \wedge \alpha(d') = u(\alpha(d)) \wedge \beta(d') = \beta(d)).$$

Our definition of β-translatability differs slightly from the original one [1], because in our model the database D is an algebra with a well defined set U_D of possible operations, while in [1] a database is a set of states on which all computable operations may be performed; therefore we must require the existence of v, such that $v(d) = d'$. This difference does not essentially change the concept of β-translatability. Our algebraic definition of a database is more general and has the advantage that it may be used in "cascades", i.e., a view of a database may act as a base for a deeper level view to constitute a layered system of views.

We can, more generally, define the **β-translatability of a static view** (α, A) by :

DEFINITION 6.3. (α, A) is β-translatable iff β is a complement of α and the entire set of view update operations U_A is β-translatable.

DEFINITION 6.4. If (α, τ, A) is a view of D, and β is a complement of α, such that $\forall u \in U_A \; \forall d \in \overline{D} : \beta(\tau u(d)) = \beta(d)$, then we say that the complement β is **constant under translation** τ or τ **translates under constant complement** β.

If (α, A) is β-translatable for some complement β, we also say that it is **translatable under constant complement.**

Among the results of Bancilhon and Spyratos, we present the two most important ones, translated into our terminology.

PROPOSITION 6.1. If A is cyclic and (α, A) is β-translatable, then there exists exactly one translation τ such that $V = (\alpha, \tau, A)$ is a loop preserving view of D and β is constant under translation τ. τ is uniquely determined by β through: $\forall u \in U_A : \tau u = (\alpha \times \beta)^{-1}$ $(u\alpha \times \beta)$. (for more details see [1]).

PROPOSITION 6.2. If A is cyclic, then for each translation τ, such that $V = (\alpha, \tau, A)$ is a loop preserving view of D, there exists a complement β of α, such that

a) (α, A) is β-translatable

b) the relation " $=*v\Rightarrow$ " is an equivalence relation

c) the complement β of α is given by: $\beta(d) = \bar{d}$ where $\bar{d}$ is the equivalence class of d with respect to $=*v\Rightarrow$.

Consider view V2 of the database-example defined in section 3. It is easy to see, that the complement β_2 of α_2 which corresponds to the translation τ_2 is as follows:

$$\forall d = (b, p) \in \overline{D} : \beta_2(b, p) = b.$$

Clearly β_2 is constant under τ_2, since breaking or repairing pieces does not alter the BILL relation.

In a similar way, a complement β_6 of α_6 which determines τ_6 is defined by

$$\forall d = (b, p) \in \overline{D} : \beta_6(b, p) = p.$$

Clearly β_6 is constant under τ_6, since changing the price of some products does not affect the information contained in the relation PIECES.

Note that the static views considered by Bancilhon and Spyratos must be *cyclic*. Note furthermore that their theory applies only for the characterization of *loop preserving* views. By theorem 2.1, we therefore conclude that the views treated by Bancilhon and Spyratos are exactly the ***cyclic partitioning views***. This is the class of views corresponding to the innermost area of figure 2.4 (or respectively figure 3.2).

However, the requirement of cyclicity is not strictly necessary for translation under constant complement. The following observations will lead to a first generalization of the theory of Bancilhon and Spyratos.

OBSERVATION 6.1. The cyclicity condition of proposition 6.1 can be omitted. This is justified by the fact that the proofs in [1] for the original formulation of proposition 6.1 do not use the cyclicity property.

OBSERVATION 6.2. In proposition 6.2.a, instead of requiring the cyclicity of A, the more general requirement that (α, τ, A) be base partitioning is sufficient. In the context of proposition 6.2 this requirement is indeed more general, since we are considering only loop preserving but not necessarily cyclic views. Any loop preserving and cyclic view is partitioning, but there exist many interesting views which are partitioning (and therefore also loop preserving) but not cyclic (for example View 1 and View 3 of section 3). Note that the parts b and c of Proposition 6.2 are not necessarily valid if we replace the cyclicity requirement by the more general requirement of a *partitioning view*: the relation $=*v\Rightarrow$ then is not necessarily an equivalence relation. We can overcome this difficulty by considering the equivalence relation induced by the partition of the view, instead of considering $=*v\Rightarrow$. Let Φ denote the equivalence relation on $\overline{D}$ induced by the partition. Parts b and c of Proposition 6.2 become true if we substitute Φ for $=*v\Rightarrow$.

The above observations suggest that there exist a strong connection between the concept of "partitioning view" and the concept of "translatability under constant complement". In facts, these two concepts are equivalent. This is formally stated in the following theorem.

THEOREM 6.1. A view (α, τ, A) of base D partitiones the base iff there exists a complement β of α, such that β is constant under translation τ.

PROOF.

"if" Let β be a complement which is constant under τ. Let $\overline{B} = \beta(\overline{D})$. Let D_x denote $\beta^{-1}(x)$ for $x \in \overline{B}$. Thus, D_x denotes the set of all elements of $\overline{D}$ which are mapped to x by β. It is trivial to verify that the family of sets $\{D_x \mid x \in \overline{B}\}$ has exactly the properties of partitioning classes required by Definition 2.13.

"only if" Assume (α, τ, A) is partitioned. Consider the equivalence classes $D_1, \ldots, D_k$ which constitute the partition as defined in def. 2.13. Let β be defined as: $\forall d \in D : \beta(d) = \overline{d}$ where $\overline{d}$ denotes the unique equivalence class Di, such that $d \in Di$. It follows trivially from property b of Definition 2.13 that β is a complement. From property a of the same definition it follows that β is constant under τ.

Q.E.D.

By this simple generalization of Bancilhon and Spyratos' theory, we are now able to state the update semantics of our view V1 by defining the complement β_1 as follows:

$$\beta_1 : \overline{\text{STORE}} \rightarrow \overline{\text{BILL}}; \quad \forall (b,p) \in \overline{D} : \beta_1(b,p) = b.$$

For the complement β_3 of the view V3 it holds: $\beta_3 = \beta_1$.

Note that, the views V1 and V3 are kind of views which are not considered in [1] because they are not cycle.

We thus have characterized the class of partitioning views as the class of views which are updatable under constant complement. The equivalence of these two notions was quite obvious. A more difficult problem is the characterization of **consistent but not partitioning views**. By theorem 6.1 we know that such views do not translate updates under constant complement.

For example, let us consider view V4 defined in section 3. It is easy to verify that the mapping β_4 such that $\forall (b,p) \in \overline{D} : \beta_4(b,p) = p$ is a complement of α_4.

β_4 is in a sense the most "reasonable" complement we can find, in order to express the information hidden to the user by α_4. However, this complement does not remain constant. It **looses information** whenever a nontrivial update operation is issued. More generally, it is easy to show that *every* complement of α_4 looses information when the view V4 is updated, using the strategy τ_4: when a piece of a product is sold, the tuple associated with the piece is eliminated from the database. This tuple contains also the piece number, which is invisible to the view user. Therefore we can say that updating V4 causes a loss of information not visible to the view. Since the lost information was not visible to the view, it did belong to the complement.

Since we have shown that the class of consistent views is of practical relevance, it is necessary to define exactly what is the characterization of their update semantics. In the next section, we show how it is possible to extend the approach of Bancilhon and Spyratos in order to capture the update semantics of the entire class of consistent views.

7. UPDATE SEMANTICS OF CONSISTENT VIEWS

The basic idea of the theory developed in this section can be stated as follows.

Let (α, A) be a static view of a database D and let $\beta: \overline{D} \rightarrow \overline{B}$ be a complement of (α, A). We will structure the set $\overline{B}$ of complement values by a partial ordering "$\leq$", such that it becomes a partially ordered set (poset). Intuitively, $\beta(d') \leq \beta(d)$ has the following meaning: All information contained in $\beta(d')$ is also contained in $\beta(d)$, but $\beta(d')$ may contain less information than $\beta(d)$.

We will furthermore require that the partial ordering $\leq$ determines unambiguously an update policy. For this reason we will introduce the notion of **α-β-decisiveness** (Def. 7.2). We will restrict the number of possible orderings $\leq$ on $\beta(\overline{D})$ by considering only α-β-decisive orderings. It will become clear later, that these orderings are sufficient to determine all consistent update policies.

DEFINITION 7.1. Let $(X, \leq)$ denote a finite poset. Let a and b be two elements of X. An element $c \in X$ is an **upper bound** of a and b with respect to $\leq$ iff $a \leq c$ and $b \leq c$. If there exists an upper bound for a and b then we write $a \uparrow b$ else we write $a \not\uparrow b$.

DEFINITION 7.2. Let (α, A) be a static view of D, let β be a complement of α and let "$\leq$" be a partial ordering defined on $\beta(\overline{D})$. The partial ordering $\leq$ is called α-β-decisive iff $\forall d, d' \in \overline{D}$: $((d \neq d' \wedge \alpha(d) = \alpha(d')) \Rightarrow \beta(d) \not\uparrow \beta(d'))$.

We are now able to introduce our generalization of the notion of β-translatability of Bancilhon and Spyratos.

DEFINITION 7.3. Let (α, A) be a static view of D, let β be a complement of α and let $\leq$ be an α-β-decisive ordering on $\beta(\overline{D})$. A view update operation $u \in U_A$ is **β-$\leq$-translatable** iff $\forall d \in \overline{D}$, $\exists d' \in \overline{D}$:

$$d \overset{*}{\Longrightarrow} d' \wedge \alpha(d') = u(\alpha(d)) \wedge \beta(d') \leq \beta(d).$$

If each update operation of the static view (α, A) is β-$\leq$-translatable, then we say that **the static view (α, A) is β-$\leq$-translatable.**

The notion of β-$\leq$-translatability is the basic definition which is necessary for introducing the concept of "translation under loss of information." Informally, the condition $\beta(d') \leq \beta(d)$ states that the complement of an updated database state can never contain more information than the complement of the original state, if the database update is a translation of a view update.

It is easy to see, that β-$\leq$-translatability is indeed a generalization of β-translatability as defined in [1]. Observe that "=" constitutes an α-β-decisive partial ordering on $\beta(\overline{D})$. Bancilhon's and Spyratos' concept of β-translatability therefore coincides with β-=-translatability.

In the following, let us assume that (α, A) is a static view of D and that β denotes a complement of α, such that (α, A) is β-$\leq$-translatable for a given partial ordering $\leq$ on $\beta(\overline{D})$. [This assumption is made for the following Definition 7.4, for Lemma 7.1 and for Definition 7.5].

DEFINITION 7.4. For each update expression $u \in E_A$ and for each database state $d \in \overline{D}$, we define a set $S_{u,d}$ as follows:

$$S_{u,d} = \{d' \mid d' \in \overline{D} \wedge d \overset{*}{\Longrightarrow} d' \wedge \alpha(d') = u(\alpha(d)) \wedge \beta(d') \leq \beta(d)\}.$$

$S_{u,d}$ consists of all database states in which d is allowed to be transformed whenever $\alpha(d)$ is updated by u.

LEMMA 7.1. $\text{card}(S_{u,d}) = 1$.

PROOF. Since α is β-$\leq$-translatable, $S_{u,d}$ is not empty. Assume, $S_{u,d}$ has two different elements d1 and d2. By the definition of $S_{u,d}$ it holds: $\alpha(d1) = \alpha(d2)$; since $d1 \neq d2$ it therefore must hold $\beta(d1) \neq \beta(d2)$. Furthermore it is $\beta(d1) \leq \beta(d)$ and $\beta(d2) \leq \beta(d)$ and thus $\beta(\mathrm{d1}) \uparrow \beta(\mathrm{d1})$. This is a contradiction to the α-β-decisiveness of $\leq$. Therefore, $\mathrm{S}_{\mathrm{u,d}}$ is a singleton.

Q.E.D.

DEFINITION 7.5. For each update operation $\mathrm{u} \in \mathrm{U_A}$, we define a function σ_u on the set $\beta(\overline{\mathrm{D}})$ as follows:

$$\sigma_\mathrm{u}: \beta(\overline{\mathrm{D}}) \to \beta(\overline{\mathrm{D}});$$

$$\sigma_\mathrm{u}(\beta(\mathrm{d})) = \beta(\mathrm{d}') \text{ such that } \mathrm{d}' \in \mathrm{S}_{\mathrm{u,d}}.$$

By Lemma 7.1 σ_u is well defined for each β and $\leq$. Since σ_u depends on β and on the particular partial ordering $\leq$ we should correctly write $\sigma_{\beta,\leq,\mathrm{u}}$ instead of σ_u. For simplicity we just write σ_u, when β and $\leq$ are determined by the context.

Informally speaking, σ_u is the update that must be applied to the complement, when the view is updated by an update operation u.

We are finally able to state the first important theorem of this section, which establishes the correspondence between ordered complements of a static view on one hand and update policies which extend the static view to a dynamic view on the other hand.

THEOREM 7.1. Assume that (α, A) is a static view of D and that β denotes a complement of α, such that (α, A) is β-$\leq$-translatable for a given partial ordering $\leq$ on $\beta(\overline{\mathrm{D}})$. The translation function τ, defined below extends the static view (α, A) to a dynamic view $(\alpha, \tau, \mathrm{A})$ of D:

$$\tau: \mathrm{U_A} \to \mathrm{E_D} \text{ such that}$$

$$\forall \mathrm{u} \in \mathrm{U_a} : \tau\mathrm{u} = (\alpha \times \beta)^{-1}(\mathrm{u}\alpha \times \sigma_\mathrm{u}\beta),$$

where σ_u is defined as in Definition 7.5 from β and $\leq$.

τ is called the **β-$\leq$-translator** for $\mathrm{U_A}$.

PROOF. Let us first observe that τ is well defined, since $(\alpha \times \beta)^{-1}$ is a *function*. This because β is a complement of α and therefore $(\alpha \times \beta)$ is one-to-one. Let d′ denote the only element of the singleton $\mathrm{S}_{\mathrm{u,d}}$. It then holds by Definition 7.5: $\sigma_\mathrm{u}(\beta(\mathrm{d})) = \beta(\mathrm{d}')$ and by definition 7.4: $u(\alpha(d)) = \alpha(d')$. Therefore we have

$$\begin{aligned} \tau u(d) &= (\alpha \times \beta)^{-1}(u\alpha \times \sigma_u\beta(d)) \\ &= (\alpha \times \beta)^{-1}(u\alpha(d) \times \sigma_u\beta(d)) \\ &= (\alpha \times \beta)^{-1}(\alpha b(d') \times \beta(d')) \\ &= d' \end{aligned}$$

Thus $\forall d \in \overline{D}$: $\tau u(d) = d'$ such that $d' \in S_{u,d}$.

We must now show that α is a homomorphism with respect to τ (see def. 2.3). We have to show that $\forall d \in \overline{D}\ \forall u \in U_A$: $(a = \alpha(d) \Rightarrow ua = \tau u(d))$. Since $\tau u(d) = d' \in S_{u,d}$, it holds $\alpha(\tau u(d)) = \alpha(d') = u(\alpha(d))$. But $u(\alpha(d)) = u(a)$. Therefore $\alpha(\tau u(d)) = \alpha(d') = u(a)$.

Q.E.D.

COROLLARY 7.1. Let (α, A) be a static view of D and τ be a β-$\leq$-translator for U_A. It holds: $\forall u \in E_A, \forall d \in \overline{D}$: $\tau u(d) = d'$ such that $d' \in S_{u,d}$.

PROOF. For $u \in U_A$ the assertion of this corollary was already proved in the proof of Theorem 7.1. The generalization to E_A follows easily by induction.

THEOREM 7.2. If (α, A) is a static view of D and τ is a β-$\leq$-translator for U_A then the view (α, τ, α) is **consistent**.

PROOF. Assume (α, τ, A) is not consistent: there exist d, d', d'' $\in \overline{D}$ with $d' \neq d''$, such that $d =*v\Rightarrow d'$ and $d =*v\Rightarrow d''$ and $\alpha(d') \neq \alpha(d'')$. Then there exist v, w $\in E_A$, such that $d' = \tau\ v(d)$ and $d'' = \tau\ w(d)$. By Corollary 7.1. it follows $d' \in S_{v,d}$ and $d'' \in S_{w,d}$. Since τ is a translator and α is a homomorphism with respect to τ it holds: $v(\alpha(d)) = w(\alpha(d))$; therefore $S_{v,d} = S_{w,d}$ and thus $d' = d''$. Contradiction.

Q.E.D.

We have thus shown that all views (α, τ, A), where τ is a β-$\leq$-translator for U_A are consistent. If we succeed in showing the converse, i.e. that for all consistent views (α, τ, A) there exist

β and $\leq$ such that τ is a β-$\leq$-translator, then we have completely characterized the variety of consistent views.

DEFINITION 7.6. Let $V = (\alpha, \tau, A)$ be a view of D. Let β_{can} be a function defined as follows:

$$\beta_{can}: \overline{D} \rightarrow 2^{\overline{D}} \quad (\text{powerset of } \overline{D})$$

$$\forall d \in \overline{D} : \beta_{can}(d) = \{d' \mid d =\!*\!V\!\Rightarrow d'\}.$$

We call β_{can} the **canonical complement** of β for the view V. (This term is justified by the following Lemma).

LEMMA 7.2. If V is consistent then β_{can} is a complement of α.

PROOF. Let $\alpha(d) = \alpha(d')$ and $d \neq d'$. Then it cannot hold $d =\!*\!V\!\Rightarrow d'$, since V is consistent (and therefore loop preserving). Thus $d' \notin \beta_{can}(d)$. On the other hand, by definition of β_{can} and of " $=\!*\!V\!\Rightarrow$ " it holds $d' \in \beta_{can}(d')$. Therefore $\beta_{can}(d') \neq \beta_{can}(d)$. Thus β_{can} is a complement of α.

Q.E.D.

We now state the second important theorem of this section.*

THEOREM 7.3. Let $V = (\alpha, \tau, A)$ be a consistent view of D. Let β_{can} denote the canonical complement of α for V. Let "$\subseteq$" denote the set-theoretic inclusion. It then holds: U_A is β_{can}-$\subseteq$-translatable and τ is the β_{can}-$\subseteq$-translator for U_A.

PROOF.

a) We first show that $\subseteq$ is α-β-decisive, i.e.

$$\forall d', d'' \in \overline{D} : (d' \neq d'' \wedge \alpha(d') = \alpha(d'')) \Rightarrow \beta_{can}(d') \dagger \beta_{can}(d'').$$

* The set theoretical induction $\subseteq$ is a partial ordering which satisfies the criteria for α-β decisiveness.

Assume $d' \neq d''$ and $\alpha(d') = \alpha(d'')$ but $\beta_{can}(d') \uparrow \beta_{can}(d'')$. Let $d \in \overline{D}$ such that $\beta_{can}(d)$ is an upper bound for both $\beta_{can}(d')$ and$\beta_{can}(d'')$. This means $\beta_{can}(d') \subseteq \beta_{can}(d)$ and $\beta_{can}(d'') \subseteq \beta_{can}(d)$. Since $d' \in \beta_{can}(d')$ and $d'' \in \beta_{can}(d'')$ it follows $d', d'' \in \beta_{can}(d)$, and therefore $d =*v\Rightarrow d'$ and $d =*v\Rightarrow d''$. This is in contradiction with the consistency of V. Thus $\subseteq$ is α-β-decisive.

b) We show that U_A is β_{can}-$\subseteq$-translatable. Let $\tau u \in E_D$, and $d, d' \in \overline{D}$ such that $d' = \tau u(d)$. It then holds $d =*v\Rightarrow d'$ and therefore $d' \in \beta_{can}(d)$. From the definition of β_{can} and from the transitivity of " $=*v\Rightarrow$ " it then follows easily $\beta_{can}(d') \subseteq \beta_{can}(d)$, which proves the β_{can}-$\subseteq$-translatability of U_A.

c) That τ is the β_{can}-$\subseteq$-translator for U_A can easily be verified by considering an update $u \in U_A$, the corresponding mapping σ_u as introduced in Definition 7.5 and by showing $\tau u = (\alpha \times \beta)^{-1}(u\alpha \times \sigma_u\beta)$.

Q.E.D.

This theorem completes the formal framework, which enables us to define the update semantics of any consistent view by specifying a complement and a partial ordering on the set of complement values. Let us now reconsider the views V4 and V5 of our example-database. We choose the same complement for both views:

$$\forall(b,p) \in \overline{D}: \quad \beta_4(b,p) = \beta_5(b,p) = p.$$

Since both views V4 and V5 have the same complement, we must express their different update semantics through different partial orderings on the set of complement values. Before doing so, let us define two particular partial ordering relations between sets of integers, which will help to simplify our notation.

DEFINITION 7.7. Let X and Y denote two sets of integers. We say that **X is a postfix of Y**, denoted by **X$Y** iff $X \subseteq Y$ and $\forall x \in X, \forall y \in Y - X : y < x$. (Here "<" has its conventional meaning). We say that **X is a prefix of Y**, denoted by **X@Y** iff $X \subseteq Y$ and $\forall x \in X, \forall y \in Y - X : y > x$.

EXAMPLE. Consider the example of section 2. Let $Y = \{1,2,5,7,8,10,14,15\}$, then $\{1,2,5\}@Y$ but $\{14,15\}\$Y$. Note that $\beta_4(\overline{D}) = \beta_5(\overline{D}) = \overline{\text{PIECES}}$, i.e. the set of legal instances of the relation PIECES. If p is a relational instance and A an attribute of p, let p[A] denote the projection of p over the attribute A; if c is a predicate on tuples of p, let p⟨c⟩ denote the set of all tuples of p which satisfy c (selection). Combinations of selections and projections are written without parenthesis, e.g. p⟨c⟩[A].

We are now ready for specifying the partial orderings $\leq_4$ and $\leq_5$ for respectively the view V4 and V5.

$\leq_4$: $\forall p, p' \in \text{PIECES} : p \leq_4 p' \Leftrightarrow$

$p' \subseteq p \wedge p'\langle \text{BROKEN} = \text{true}\rangle = p\langle \text{BROKEN} = \text{true}\rangle \wedge$

$\forall 1 \leq n \leq 100 :$

$p'\langle \text{BROKEN} = \text{false} \wedge \text{PRODUCT\#} = n\rangle[\text{PIECE\#}]\$p\langle \text{BROKEN} =$
$\text{false} \wedge \text{PRODUCT\#} = n\rangle[\text{PIECE\#}].$

$\leq_5$: $\forall p, p' \in \text{PIECES} : p \leq_5 p' \Leftrightarrow$

$p' \subseteq p \wedge p'\langle \text{BROKEN} = \text{true}\rangle = p\langle \text{BROKEN} = \text{true}\rangle \wedge$

$\forall 1 \leq n \leq 100 :$

$p'\langle \text{BROKEN} = \text{false} \wedge \text{PRODUCT\#} = n\rangle[\text{PIECE\#}]@p\langle \text{BROKEN} =$
$\text{false} \wedge \text{PRODUCT\#} = n\rangle[\text{PIECE\#}].$

It is easy to see that $\leq_4$ and $\leq_5$ are indeed partial orderings. It is possible to prove that $\leq_4$ is α_4-β_4-decisive and that $\leq_5$ is α_5-β_5-decisive. A formal proof thereof is given in the appendix.

That U_{A4} is β-$\leq_4$-translatable and that U_{A5} is β-$\leq_5$-translatable is easily verified, as well as the fact that $\leq_4$ corresponds to τ_4 and $\leq_5$ corresponds to τ_5.

We thus have specified the update semantics of V4 and V5 by supplying for each of these views a complement and a partial ordering on the complement values.

From the properties of these complements and partial orderings we furthermore deduce that V4 and V5 are *consistent views*.

8. RELATED WORK

Paolini and Pelagatti [5] considered a database as an abstract object upon which one may operate with a given set of operations. This approach is further developed in [6] and [7], where databases and views are modeled as data abstractions. Also Rowe and Schoens [20], Claybrook et al. [16], Lockemann et al. [21] and Weber [25] all use an abstract data type approach to model database views.

Most of the authors who have been studying the view update problem concentrate their attention on finding ways for deriving translations automatically or semi-automatically by using particular update policies or by restricting the set of allowed update policies. Their derivation rules usually are based upon notions of "natural translation" (typically minimality of side-effects) and upon constraints on the data model and on the database instances (typically functional dependencies for relational databases).

Examples of this approach are Dayal and Bernstein [3] for automatic translation within the context of the relational model and Dayal and Bernstein [22] for automatic translation of updates on network views. In [3] only views which are combinations of projections and selections and joins of relations are considered. A careful analysis of different types of update operations, such as insertions, deletions and replacements is given. For these types of operations, Dayal and Bernstein consider translators which do not necessarily lead to consistent views in the sense of our definition. On the other hand, their model of view definition does not include aggregate functions, thus they are not able to handle such important views as our examples V4 and V5 (section 3). A primary objective of [3] is the preservation of integrity constraints (functional dependencies).

Siklossy [23] assumes, as a prerequisite, that views preserve loops, and calls this property "minimal admissibility".

Furtado et al. [12] provide rules on permissivity of various types of updates. They restrict the number of allowed update operations and conclude that "some operation must be prohibited in order to assure harmonious interaction among database users".

Fagin et al. [11] provide a framework for the interpretation of updates in the context of logical databases.

The work of Bancilhon and Spyratos [1], outlined in section 6 of our paper, has stimulated many further investigations by different researchers. Cosmadakis and Papadimitriou [9] show that finding a minimal complement of a given view is NP-complete. They adopt the Universal Relation Assumption [26] and their views are essentially projections of a given universal relation. Keller and Ullman [8] define the notion of independent views (i.e. views whose ranges of values may be achieved independently) and consider the relationship between complementary views and independent views. Our own work is partially based on Bancilhon and Spyratos' ideas. Sections 6 and 7 of the present paper generalize the notion of "constant complement".

In [18] and [27] Keller analyzes the possible translations of of particular classes of update operations for relational views (The considered updates are, as in [3], insertions, deletions and replacements.) Keller gives five criteria that all candidate update translations must satisfy. The satisfaction of these criteria implies restrictions on the view definition function α and on the form of view update expressions that our approach does not require. In Keller's model, for instance, the key of each data relation that may be affected by updates must appear in the view. Some combinations of view update operations are not allowed, such as the replacement of a tuple A by a tuple B followed by the deletion of tuple B. This is due to the fact that Keller's translations depend on the particular sequence of the view update operations, and not solely on the functional semantics of the view update (as in our approach). As in [3], aggregate functions are not covered. On the other hand, Keller's model includes some interesting views that are not covered by our approach (for example, some non-loop-preserving views). Keller also shows how the choice of a translator can be done semi-automatically by a program which conducts a dialog with the data base administrator.

9. CONCLUSIONS AND FUTURE RESEARCH

The overview of related work given in the previous section shows that no approach undertaken so far to solve the view-update problem is complete in the sense that it covers all possible views of practical relevance. This criticism is also valid for our approach. We do not claim that the classes of views studied in the present paper capture the complete spectrum of all "reasonable" views. We believe, however, that the class of consistent views covers a large number of important and interesting applications, some of which are not covered by other approaches. In particular, we have shown that the class of consistent views is a superset of the class of views studied by Bancilhon and Spyratos in [1]. We have given examples of nontrivial applications that can be modeled by our approach, but not by the approach described in [1].

We have shown that consistent views are characterized by extremely useful properties. In particular, we proved that the consistency of a view is not affected when the original set of update expressions is augmented by the possibility of conditional execution of view updates. We have shown that the translation function of a consistent view preserves functional equivalence of update programs, as well as a number of important concurrency properties. We have shown that the update semantics of consistent views can be determined by imposing a partial order on the values of the view complement.

We believe that the class of consistent views merits attention for its good properties. However, more research is needed in order to render our results more applicable. Let us conclude this paper by giving some outlines of the research we plan to carry out in the near future:

- Study the properties of different types of relational views (projective views, selective views, join views ...) for different types of update operations (insertion, deletion, replacement). Derive necessary and sufficient conditions for the consistency of such views.

- Extend our model of view update programs to cover recursively defined update programs. Note that such an extension would require the capability of handling view and database updates which are partial operations.

- Study the interaction between different views.

- Find algorithms to compute all possible translation functions for a given static view in order to obtain a consistent dynamic view. Our ongoing research shows that there exists a strong relationship between the set of different consistent translation functions and the set of all spanning trees of a directed graph. (This result has not yet been proved in full generality but only for some particular classes of static views.) Therefore we think that it could be useful to take profit of existing graph theoretic algorithms in order to generate translation functions.

- Given an inconsistent view V, find methodologies for splitting V into two or more consistent views whose sets of operations are subsets of the set of operations of V. With such a splitting it is possible to replace a program P that originally operates on the inconsistent view V by a program P' which switches between different consistent views (for example through explicit switching primitives). The parts of P' that are executed completely within one consistent view can be modified as long as their functional semantics are not affected. All advantages provided by consistent views apply to these program parts.

10. REFERENCES

[1] Bancilhon F., Spyratos N., "Update Semantics of Relational Views", ACM TODS, vol. 6, n. 4, Dec. 1981.

[2] Dayal U., Bernstein P.A., "On the Updatability of Relational Views", Proc. 4th VLDB, West Berlin,1978.

[3] Dayal U., Bernstein P.A., "On the Correct Translation of Update Operations on Relational Views", ACM TODS, vol. 8, n. 3, Sept. 1982.

[4] Goguen J.A., Thatcher J.W., Wagner E.G., Wright J.B., "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", in Data Structuring (Current Trends in Programming Methodology, vol. 4), R.Yeh, Editor, Englewood Cliffs, NJ, Prentice Hall, 1978.

[5] Paolini P., Pelagatti G., "Formal Definition of Mappings in a Database", Proc. ACM SIGMOD, Toronto, 1977.

[6] Paolini P., "Verification of Views and Applications Programs", Workshop Formal Bases for Databases, Toulouse, France, Dec. 1979.

[7] Paolini P., Zicari R., "Properties of Views and their Implementation", in Advances in Database Theory, vol. 2, Jack Minker et al. editors, Plenum Press, New York, 1984.

[8] Keller A.M., Ullman J.D., "On Complementary and Independent Mappings on Databases", Proc. ACM SIGMOD, Boston, June 1984.

[9] Cosmadakis S.S., Papadimitriou C.H., "Update of Relational Views", JACM, vol. 31, n. 4, Oct. 1984.

[10] Astrahan, M.M. et al., "System R: Relational Approach to Database Management", ACM TODS, vol. 1, n. 2,June 1976.

[11] Fagin R., Ullman J.D., Vardi M.Y., "On the Semantics of Updates in Databases", Proc. 2nd ACM SIGACT-SIGMOD Symp., 1983.

[12] Furtado A., Sevcik K.C., Dos Santos C.S., "Permitting Updates through Views of Database", Information System, vol. 4, n. 4, 1979.

[13] Stonebraker M., et al., "The Design and Implementation of INGRES", ACM TODS, vol. 1, n. 3, Sept. 1976.

[14] Zloof M.M., "Query-by-example: A database language", IBM Syst. J., 16, 4, 1977.

[15] Gottlob G., Paolini P., Zicari R., "Proving Properties of Programs on Database Views", Dipartimento di Elettronica, Politecnico di Milano (in preparation).

[16] Claybrook B.G., Claybrook A.M., Williams J., "Defining Database Views as Data Abstractions", IEEE-TSE, vol. SE-11, n. 1,Jan. 1985.

[17] Ullman J.D., "Principles of Database System", Computer Science Press, Potomac, MD, second edition, 1982.

[18] Keller A.M., "Algorithms for Translating View Updates to Database Updates for Views involving Selections, Projections and Joins", Proc. ACM SIGACT-SIGMOD Symp., March 1985.

[19] Date C.J., "A Guide to DB2", Addison Wesley, Reading, MA., 1984.

[20] Rowe L., Schoens K.A., "Data Abstractions, Views and Updates in RIGEL", Proc. ACM SIGMOD, Boston, MA, May 1979.

[21] Lockemann P.C., et al., "Data Abstractions for Database Systems", ACM TODS, vol. 4, March 1979.

[22] Dayal U., Bernstein P.A., "On the Updatability of Network Views - Extending Relational View Theory to the Network Model", Information Systems, vol. 7, n. 1, 1982.

[23] Siklossy L., "Updating Views: a Constructive Approach", Proc. Workshop Logical Bases for Databases, Toulouse, Dec. 1982.

[24] Maier D., "Theory of Relational Databases", Computer Science Press, Rockville, MD, 1983.

[25] Weber H., "A Software Engineering View of Database Systems", Proc. VLDB, Sept. 1978.

[26] Ullman J.D., "The U.R. Strikes Back", Proc. ACM Principles of Database Systems, Los Angeles, March 1982.

[27] Keller A.M., "Updating Relational Databases Through Views", Stanford University, Computer Science Dept., Ph.D. dissertation, 1985.

[28] Papadimitriou C.H., "Serializability of Concurrent Updates", Journal of the ACM, 26:4, 1979.

[29] Bernstein P.A., Goodman N., Lay M-Y., "On Analysing Concurrency Control Algorithms When User and System Operations Differ", IEEE-TSE, SE 9:3, 1983.

[30] Bernstein P.A., Shipman D.W. and Rothnie J.B., "Concurrency Control in a System for Distributed Databases (SDD-1)", ACM-TODS, 5:1, 1980.

APPENDIX

Before giving the proof that $\leq_4$ and $\leq_5$ are α-β-decisive, let us state a simple lemma.

LEMMA.

a) $(X\$Z \wedge Y\$Z \wedge \mathrm{card}(X) = \mathrm{card}(Y)) \Rightarrow X = Y$.

b) $(X@Z \wedge Y\$Z \wedge \mathrm{card}(X) = \mathrm{card}(Y)) \Rightarrow X = Y$.

The trivial proof of this lemma is omitted.

PROPOSITION.

a) $\leq_4$ is α_4-β_4-decisive.

b) $\leq_5$ is α_5-β_5-decisive.

PROOF. a) Assume $\leq_4$ is not α_4-β_4-decisive. Then there exist two database states $d = (b, p)$ and $d' = (b', p')$ such that

$$(b, p) \neq (b', p') \tag{a}$$

and

$$\alpha_4(b, p) = \alpha_4(b', p') \tag{b}$$

and

$$\beta(b, p) \upharpoonleft_4 \beta(b', p') \tag{c}$$

From (a) and (b) it follows:

$$b = b' \tag{d}$$

From (c) it follows that there exists a p'' with:

$$p'' \subseteq p \wedge p'' \subseteq p' \tag{e}$$

and

$$p\langle \mathrm{BROKEN} = \mathrm{true}\rangle = p'\langle \mathrm{BROKEN} = \mathrm{true}\rangle = p''\langle \mathrm{BROKEN} = \mathrm{true}\rangle \tag{f}$$

and

$$\begin{aligned}&\forall 1 \leq \mathrm{n} \leq 100:\\&\mathrm{p}\langle \mathrm{BROKEN} = \mathrm{false} \wedge \mathrm{PRODUCT\#} = \mathrm{n}\rangle[\mathrm{PIECE\#}]\ \$\\&\mathrm{p}''\langle \mathrm{BROKEN} = \mathrm{false} \wedge \mathrm{PRODUCT\#} = \mathrm{n}\rangle[\mathrm{PIECE\#}].\end{aligned} \tag{g}$$

and

$$\begin{aligned}&\forall 1 \leq \mathrm{n} \leq 100:\\&\mathrm{p}'\langle \mathrm{BROKEN} = \mathrm{false} \wedge \mathrm{PRODUCT\#} = \mathrm{n}\rangle[\mathrm{PIECE\#}]\ \$\\&\mathrm{p}''\langle \mathrm{BROKEN} = \mathrm{false} \wedge \mathrm{PRODUCT\#} = \mathrm{n}\rangle[\mathrm{PIECE\#}].\end{aligned} \tag{g}$$

Furthermore it follows from (b) by definition of α_4:

$$\begin{aligned}\forall 1 \leq n:\quad &\mathrm{card}(\mathrm{p}\langle \mathrm{BROKEN} = \mathrm{false} \wedge \mathrm{PIECE\#} = \mathrm{n}\rangle) =\\&\quad \mathrm{card}(\mathrm{p}'\langle \mathrm{BROKEN} = \mathrm{false} \wedge \mathrm{PIECE\#} = \mathrm{n}\rangle).\end{aligned} \tag{h}$$

It follows immediately:

$$\begin{aligned}\forall 1 \leq n:\quad &\mathrm{card}(\mathrm{p}\langle \mathrm{BROKEN} = \mathrm{false} \wedge \mathrm{PIECE\#} = \mathrm{n}\rangle[\mathrm{PIECE\#}]) =\\&\quad \mathrm{card}(\mathrm{p}'\langle \mathrm{BROKEN} = \mathrm{false} \wedge \mathrm{PIECE\#} = \mathrm{n}\rangle[\mathrm{PIECE\#}]).\end{aligned} \tag{i}$$

From (g) and (i), by our Lemma, it follows:

$$\begin{aligned}&\forall 1 \leq n \leq 100:\\&\mathrm{p}'\langle \mathrm{BROKEN} = \mathrm{false} \wedge \mathrm{PRODUCT\#} = \mathrm{n}\rangle[\mathrm{PIECE\#}] =\\&\mathrm{p}\langle \mathrm{BROKEN} = \mathrm{false} \wedge \mathrm{PRODUCT\#} = \mathrm{n}\rangle[\mathrm{PIECE\#}].\end{aligned} \tag{j}$$

By the integrity constraints c and d given in section 3 we know that $1 \leq \mathrm{p.PRODUCT\#} \leq 100$ and $1 \leq \mathrm{p'.PRODUCT\#} \leq 100$.

Therefore from (j) we conclude

$$\mathrm{p}'\langle \mathrm{BROKEN} = \mathrm{false}\rangle = \mathrm{p}\langle \mathrm{BROKEN} = \mathrm{false}\rangle \tag{k}$$

Putting together (k) and (f) we immediately get:

$$p' = p$$

but this, together with (d) is in contradiction with (a). Q.E.D.

b) The proof is similar to the one of case a).