Flexible View Update

YUNG-CHIA LEE*

and

CHENG-HONG CHO

Department of Electrical Engineering and Computer Science,
The University of Michigan, Ann Arbor, Michigan 48109-2122

ABSTRACT

As uninstantiated windows onto a relational database, views are modified only when the intended update can be realized by updates against the underlying database. Such a conventional restriction can, and must, be relaxed when some relations, as collections of predicates, are composed of both basic and derived facts. Therefore, this paper presents an extended formulation for the problem of view updating based on two notions: the internal state and the perceptible state of a database. Through a clear distinction between these two database states, a mechanism is proposed to facilitate those legitimate view updates that are not necessarily translatable. The proposed mechanism also relies heavily on the normalization theory through functional dependencies.

1. INTRODUCTION

A view is a query definition named and stored to represent a dynamic picture of the query. It enables a user to reconstruct some portion of a database into a format appropriate for specific applications. When a view is called for, a set of tuples is derived from the relations of the database according to its definition [12, 16]. Since this set of tuples resembles a relation, it is called a *derived relation* or a *virtual relation*. To distinguish views from relations of the database, the latter are usually called *base relations* or *real relations*.

Views have been supported in most database management systems. As *personalized* pictures of a database, views have offered convenient ways for the user to perceive the database and to retreive information. However,

*Telephone: (908) 957-3319; E-mail:yungchia@speedy.att.com facilities for view updating are hardly available; database updates can be made against base relations only.

1.1. TRANSLATABLE VIEW UPDATE

View updating is the process of modifying a relational database through views. Yet, there is a question whether the user or the database system should perform the translation from a view update to base-relation updates. It seems more desirable to allow the user to update a view directly, while leaving to the database system its translation onto base relations. Such a translation, if done by the user, may be ambiguous or ill defined, and consequently create inconsistencies in the database or cause adverse side effects on the view. During the last decade, many researchers have worked on the problem of view update [1-3, 7, 9, 11, 14]. Most of them have focused on translating view updates into base-relation updates. If a view update has a valid translation, the request is considered admissible and the corresponding updates to base relations are subsequently performed. Otherwise, without a valid translation, the view update is rejected.

Dayal and Bernstein [3] provide a good formulation for this problem. They define the *correct translation* for a view update as well as the conditions for the existence of such a translation. They also present the translation algorithms for the class of view updates with correct translations. Bancilhon and Spyratos [1] further define a *complete set* of updates, a *complementary view* of a view update, and the sufficient and necessary condition for determining whether a view update is translatable. As defined, a view update is translatable if it corresponds to a *valid* and *returnable* database update. An update is valid if the database remains consistent after the change. An update is returnable if there exists an inverse update that brings the view back to the original state. The work was later extended by Cosmadakis and Papadimitriou [2]. Meanwhile, Keller [10] proposed an approach which requires the database administrator (DBA) to answer a sequence of questions when defining a view. These answers are used to choose a valid translator for each view update. The definition of the translator is stored along with the view definition, which involves select, project, and join operations.

The above approaches focus on determining whether a view update is translatable and, if so, how it can be translated. The underlying philosophy is that "*Since in the common model of relational databases the view is only a uninstantiated window onto the database, any update specified against the database view must be translated into updates against the underlying database*" [10]. The corresponding database updates then produce a new view state as if the update were performed on the view directly. This process is

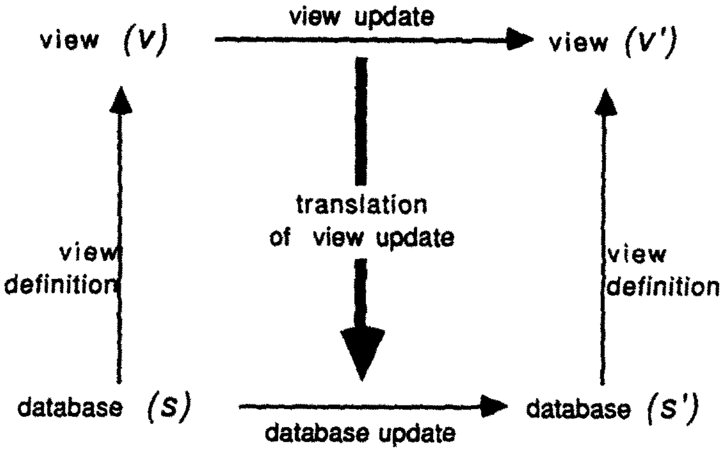

Fig. 1. Translation of view update.

briefly illustrated in Figure 1. View state v is derived from the database in state s, according to its definition. The transition of view state from v to v' is performed by moving the database from state s to a state s' such that v' can be derived from s'.

1.2. NONTRANSLATABLE BUT LEGITIMATE VIEW UPDATES

Unfortunately, there are view updates that are legitimate but not translatable. These view updates would be rejected by the approaches just described. The main reason is that the information associated with each view update may not be sufficient to update all corresponding base relations, for example, missing the key value for some base relation. However, some of these updates are logically acceptable, as they never result in any database inconsistency; theoretically, an update request should be rejected only if it results in an inconsistent database state. In reality, there are indeed nontranslatable updates which are essential and of interest to some applications. Let us consider *legitimate view updates* as logically acceptable view updates. The set of translatable view updates is therefore only a subset of legitimate view updates. Specifically, a translatable view update is both legitimate and functionally acceptable, where the function is the translation onto base relations. An important question can then be raised: "Should a database management system reject an update request as a result of its own limitations?" If the answer is yes, the state of art being so, the database management system will remain incapable of fulfilling more advanced demands. On the other hand, if the answer is no, one must develop a more sophisticated update mechanism which allows users to retrieve as well as modify the databases through the same scenes, i.e. views.

We have thus far discussed the problem of view update from a traditional perspective. There is at least one different way to see the need of accepting nontranslatable but legitimate view updates. From the logic point of view, a tuple of a relation in the relational database is a *ground atomic formula* in first-order logic, and then the database can be viewed as a collection of formulas belonging to two classes [9]: *database intension* (IDB) and *database extension* (EDB). IDB corresponds to time-invariant properties of database and usually consists of integrity constraints and view definitions. EDB reflects the current state of knowledge about the part of the world being modeled and is subject to frequent updates.

Models such as Datalog, described in [17], consider that a predicate whose relation is stored in the database is an EDB relation, while one defined by logical rules is an IDB relation. It can be shown that an IDB relation is equivalent to a view in the relation model as long as neither any of its rules nor any of the relations from which it is derived is recursively defined. What concerns this paper is the assumption usually made that each predicate symbol denotes either an EDB relation or an IDB relation, but not both. In many cases this assumption would be fine, but there are situations where a *naturally named* predicate simply denotes both types of relations. As far as end users are concerned, a missing fact must be inserted if not shown in the collection of formulas corresponding to that particular predicate name. Thus, it is the responsibility of the system or database administrators to, for example, set up two relations, one in IDB and the other in EDB, that are *artificially* named differently in terms of the internal state of the database.

1.3. OBJECTIVES

There appears to be a clear distinction between end users and the database system, as far as naming predicates in a logic-based model (or naming relations/views in a relational model) is concerned. When the facility for view update is discussed at the system level, it can be assumed that a predicate symbol will never denote both an IDB relation and an EDB relation, and a view is only an uninstantiated window onto the database. On the other hand, when it is discussed at the end-user level, these assumptions are too inflexible to follow. Considering the fact that updates are realized by the system but are initiated by end users, it seems more appropriate to deal with the problem of views updates by separating the two levels.

This paper hence suggests for the problem of view updates a more comprehensive formulation, which recognizes the distinction between the system level and the end-user level. Based on this formulation, a more flexible update mechanism is proposed to accommodate view updates that are not necessarily translatable. In the following, Section 2 presents the proposed problem formulation. The more flexible update mechanism is then described in Section 3. Section 4 further elaborates specific problems that have been identified and solved under this formulation.

2. PROBLEM FORMULATION

In this section, two different database states are identified. The *internal* state denotes the database at the system level, while the *perceptible* state denotes the database perceived at the end-user level. Described first are the internal states and the consistent transitions among them. The perceptible states are subsequently introduced along with their consistent transitions. Lastly, the problem of view updates is generalized and presented as a problem of update realizations.

2.1. *INTERNAL STATES OF A DATABASE*

The logical structure of a relational database is generally defined by its *conceptual schema*. The conceptual schema of a relational database consists of a set of relation schemes. Each relation scheme is formed by a set of attributes. Corresponding to each relation scheme, there is a relation extension, usually represented by a set of tuples. The content of the database defined on the conceptual schema is called the *extension of the conceptual schema* or *database extension* The database extension and the conceptual schema together form the foundation of the database, called *base relations*.

A database can be extended by employing *view definitions* to derive new facts as *virtual relations*. Each view definition can be applied as an inference rule to construct a view. View definitions can thus be defined on both base relations and virtual relations. The scheme, i.e., the set of attributes, of a view is defined by the view definition. At any instant of time, the content of a view, the *view extension*, can be obtained by applying its view definition to the database extension and other view extensions. Although views defined on other views cannot be derived directly from base relations, the closure of the set of view definitions can derive all views from base relations.

In addition, certain integrity constraints can be employed to prevent the database from being inconsistent. If some fact must not occur in the world being modeled, there should be a rule to prevent the database from stating that fact. Rules of this type are called *integrity constraints*. An integrity constraint is a rule that states a specific restriction on the content of database, including the database extension and view extensions.

An internal state of a database can be defined as follows:

DEFINITION 2.1. An internal state of a database is a quadruple (S, E, V, I). S is the conceptual schema of the database; E is a database extension represented by sets of tuples defined on S; V is a set of view definitions that are applicable to S and E; I is the set of integrity constraints of the database. The set of all internal states is denoted by $\sum$. Let the sets of all conceptual schemata, database extensions, view definition sets, and integrity constraint sets be denoted by $\sum_S$, $\sum_E$, $\sum_V$, and $\sum_I$, respectively. For every $(S, E, V, I) \in \sum$, one has $S \in \sum_S$, $E \in \sum_E$, $V \in \sum_V$, and $I \in \sum_I$.

As mentioned earlier, a view scheme is defined by the view definition. In contrast to conceptual schema, we define view schema as the following:

DEFINITION 2.2. Let $(S, E, V, I) \in \sum$. The *view schema* of the database, $V(S)$, is obtained by applying V to S. The *view extension* of the database, $V(E)$, is obtained by applying V to E.

Let us define a partially ordered set for all conceptual schemata so as to elaborate the relationships among them. A conceptual schema is said to be *contained* in another conceptual schema if and only if all the schemes appear in the former are also in the latter.

DEFINITION 2.3. Let S_1 and S_2 be conceptual schemata. S_1 is said to be *contained* in S_2, written $S_1 \leqslant S_2$, if and only if every relation scheme in S_1 is also in S_2.

Isomorphic to containment among sets, this relation is a partial order relation and implies the following theorem.

THEOREM 2.1. *Let S and S' be conceptual schemata, E be sets of tuples, V be a set of view definitions, and I be a set of integrity constraints. If $(S, E, V, I) \in \sum$ and $S \leqslant S'$ then $(S', E, V, I) \in \sum$.*

Proof. S' is known as a conceptual schema. $(S, E, V, I) \in \sum$ concludes that E is a database extension defined on S, V is a set of view definitions that are applicable to S and E, and I is the set of integrity constraints of the database. Since $S \leqslant S'$, every relation scheme in S is also in S'. This implies that E is a database extension defined on S', and V is a set of view definitions that are applicable to S' and E. By Definition 2.1, $(S', E, V, I) \in \sum$. ■

We can now define what constitutes a consistent internal state of a database.

DEFINITION 2.4. An internal state (S, E, V, I) is said to be *consistent* if and only if

(1) $(S, E, V, I) \in \Sigma$.
(2) the set of integrity constraints I itself is consistent, and
(3) the facts represented by E and $V(E)$ follow the constraints stated in I.

The set of all consistent states can be denoted by Σ_C.

The two theorems below immediately follow from the definition of consistent internal state.

THEOREM 2.2. *Let S and S' be conceptual schemata, where $S \leqslant S'$. If $(S, E, V, I) \in \Sigma_C$ then $(S', E, V, I) \in \Sigma_C$.*

Proof. $(S, E, V, I) \in \Sigma_C$ implies that $(S, E, V, I) \in \Sigma$. Since $S \leqslant S'$, by Theorem 2.1, we have $(S', E, V, I) \in \Sigma$. Since $(S, E, V, I) \in \Sigma_C$, the set of integrity constraints I itself is consistent, and the facts represented by E and $V(E)$ follow the constraints stated in I. By definition, $(S', E, V, I) \in \Sigma$. ■

THEOREM 2.3. *If $(S, E, V, I) \in \Sigma$ then $(S, E, V, \phi) \in \Sigma_C$.*

Proof. $(S, E, V, I) \in \Sigma$ implies that E is a database extension defined on S, and V is a set of view definition that are applicable to S and E. The empty set of integrity constraints itself is certainly consistent, and the facts represented by E and $V(E)$ satisfy the constraints stated in $\varnothing$. (In fact, there is no constraint to satisfy.) By Definition 2.4, $(S, E, V, \varnothing) \in \Sigma_C$. ■

2.2. *TRANSITIONS BETWEEN INTERNAL STATES OF A DATABASE*

Database transitions at the system level are referred to as transitions between internal states. Discussed first in the following are the transitions of an internal state in general. We then define what constitutes a consistent transition. In particular, an important conclusion will be drawn stating that every consistent transition can be represented by a series of consistent primitive transitions.

Database transitions are initiated by requests for modifications. Corresponding to each component of the internal state, we can first define a function that only changes the specific component.

DEFINITION 2.5. The functions for the basic changes are:

$$\lambda_S : \Sigma_S \times \Psi_S \to \Sigma_S,$$
$$\lambda_E : \Sigma_E \times \Psi_e \to \Sigma_E,$$
$$\lambda_V : \Sigma_V \times \Psi_V \to \Sigma_V,$$
$$\lambda_I : \Sigma_I \times \Psi_I \to \Sigma_I,$$

where Ψ_S, Ψ_E, Ψ_V, and Ψ_I are the four sets of basic functions that can be applied to the four components, respectively. In the table below, "$\surd$" indicates the component to be modified:

	S	E	V	I	Modification request
Ψ_S	$\surd$				Add/delete relation schemes
Ψ_E		$\surd$			Add/delete tuples of relations
Ψ_V			$\surd$		Add/delete view definitions
Ψ_I				$\surd$	Add/delete integrity constraints

In general, a modification produces simultaneous changes in an arbitrary number of components of the internal state. It has the total effect of a combination of several basic changes. Thus, we can define a modification as the following:

DEFINITION 2.6. A *modification* is a quadruple $(\psi_S, \psi_E, \psi_V, \psi_I)$, where $\psi_S \in \Psi_S$, $\psi_E \in \Psi_E$, $\psi_V \in \Psi_V$, and $\psi_i \in \Psi_I$. The set of all modifications is denoted by $\boldsymbol{M}$.

DEFINITION 2.6a. A modification $m = (\psi_S, \psi_E, \psi_V, \psi_I)$ is said to be *contained* in another modification $m' = (\psi'_S, \psi'_E, \psi'_V, \psi'_I)$ if and only if $\psi_S \subseteq \psi'_S$, $\psi_E \subseteq \psi'_E$, $\psi_V \subseteq \psi'_V$, and $\psi_I \subseteq \psi'_I$.

DEFINITION 2.7. A *transition* of an internal state is a transformation of a database from an internal state to another. The *transition function* $\delta : \Sigma \times \boldsymbol{M} \to \Sigma$ is defined as follows:

$$\delta[(S, E, V, I), (\psi_S, \psi_E, \psi_V, \psi_I)] = (S', E', V', I'),$$

where

$$S' = \lambda_S(S, \psi_S),\ E' = \lambda_E(E, \psi_E),\ V' = \lambda_V(V, \psi_V),\ \text{and } I' = \lambda_I(I, \psi_I).$$

An internal transition is thus formulated by applying this transition function to an internal state and a modification. Of particular interest are the kind of modifications below that are associated with special transition functions.

DEFINITION 2.8. A modification $m = (\psi_S, \psi_E, \psi_V, \psi_I)$ is said to be a *primitive modification* if and only if

(1) $\psi_E = \psi_V = \psi_I = \varnothing$,
(2) $\psi_S = \psi_V = \psi_I = \varnothing$,

(3) $\psi_S = \psi_E = \psi_I = \emptyset$, or
(4) $\psi_S = \psi_E = \psi_V = \emptyset$.

The set of all primitive modifications is denoted by M_ρ. A *primitive internal transition* is thus an internal transition through a primitive modification.

We are now ready to define a consistent (internal) transition and relate it to primitive consistent transitions.

DEFINITION 2.9. An internal transition is said to be *consistent* if and only if it transforms a database from a consistent state to another consistent state.

Let us first define an operator to concatenate primitive modifications.

DEFINITION 2.10. A *concatenate operator*, $\circ$, is defined as follows: Let m_0, m_1, $m_2 \in M$ and $s \in \Sigma$. Then $m_0 = m_1 \circ m_2$ if and only if $\delta(s, m_0) = \delta(\delta(s, m_1), m_2)$.

The theorem below follows immediately:

THEOREM 2.4. *Let m_0, m_1, $m_2, \cdots, m_n \in M$ and $s \in \Sigma$. Then $m_0 = m_1 \circ m_2 \circ \cdots \circ m_n$ if and only if $\delta(s, m_0) = \delta(\delta(\cdots(\delta(\delta(s, m_1), m_2), m_3)\cdots), m_n)$.*

Proof. Let $n = 2$. By Definition 2.10, $m_0 = m_1 \circ m_2$ iff $\delta(s, m_0) = \delta(\delta(s, m_1), m_2)$. Assume it is true for $n = k$. Prove that it is also true for $n = k+1$: By Definition 2.10, $m_0 = (m_1 \circ m_2 \circ \cdots \circ m_k) \circ m_{k+1}$ iff $\delta(s, m_0) = \delta(\delta(s, (m_1 \circ m_2 \circ \cdots \circ m_k)), m_{k+1})$. By the induction hypothesis, $\delta(s, (m_1 \circ m_2 \circ \cdots \circ m_k)) = \delta(\delta(\cdots(\delta(\delta(s, m_1), m_2), m_3)\cdots), m_k)$. Therefore, $m_0 = (m_1 \circ m_2 \circ \cdots \circ m_k) \circ m_{k+1}$ iff $\delta(s, m_0) = \delta(\delta(\cdots(\delta(\delta(s, m_1), m_2), m_3)\cdots), m_{k+1})$. By induction, $\delta(s, m_0) = \delta(\delta(\cdots(\delta(\delta(s, m_1), m_2), m_3)\cdots), m_n)$ ■

The intent of introducing the above operator is to describe a complete set of modifications that is small but sufficient to characterize all state transitions.

DEFINITION 2.11. An arbitrary set of modifications M^* is said to be a *complete modification set* for a set of internal states Σ^* if and only if for every pair of states s, $s' \in \Sigma^*$ there is a series of modifications $m_1, m_2, \cdots, m_n \in M^*$ $(n \geqslant 0)$ which transforms the database from state s to state s'. That is,

$$\delta(s, m_1 \circ m_2 \circ \cdots \circ m_n) = \delta\Big(\delta\Big(\cdots\Big(\delta(\delta(s, m_1), m_2), m_3\Big)\cdots\Big), m_n\Big) = s'.$$

With the complete modification set so defined, we are ready to assert the following:

LEMMA 2.1. *Every consistent (internal) transition can be represented by a series of consistent primitive transitions.*

Proof. Let $\delta(s, m) = s'$ be a consistent transition from state $s = (S, E, V, I)$ to state $s' = (S', E', V', I')$. Let $s_1 = (S, E, V, \varnothing)$, $s_2 = (S, E, \varnothing, \varnothing)$, $s_3 = (S, \varnothing, \varnothing, \varnothing)$, $s'_1 = (S', E', V', \varnothing)$, $s'_2 = (S', E', \varnothing, \varnothing)$, and $s'_3 = (S', \varnothing, \varnothing, \varnothing)$. Since $\delta(s, m) = s'$ is a consistent transition, $s, s' \in \sum_C$. This implies s, $s' \in \sum$. By Theorem 2.3, s_1, $s'_1 \in \sum_C$. The fact that s, $s' \in \sum$ also implies that E is a database extension defined on S, and E' is a database extension defined on S'. Evidently, an empty set of view definitions is applicable to S and E, and an empty set of database extensions is considered as being defined on any conceptual schema. Therefore, $s_2, s'_2, s_3, s'_3 \in \sum$. By Theorem 2.3, $s_2, s'_2, s_3, s'_3 \in \sum_C$. By Definitions 2.8 and 2.9, $\delta(s, m_1) = s_1$, $\delta(s_1, m_2) = s_2$, $\delta(s_2, m_3) = s_3$, $\delta(s_3, m_4) = s'_3$, $\delta(s'_3, m_5) = s'_2$, $\delta(s'_2, m_6) = s'_1$, and $\delta(s'_1, \mathrm{m}_7) = s'$ are consistent primitive database transitions. Therefore, $m = m_1 \circ m_2 \circ \cdots \circ m_7$. This implies that every consistent transition can be decomposed into a series of consistent primitive transitions. ■

In other words, the set of primitive modifications is the only set we have to deal with when ensuring consistent transitions.

THEOREM 2.5. *The set of all consistent primitive modifications, M_p, is a complete modification set for $\sum_C$.*

Proof. By Lemma 2.1, every consistent transition can be decomposed into a series of consistent primitive transitions. By Definition 2.9 the set of all consistent primitive modifications, M_p, is a complete modification set for $\sum_C$. ■

We have thus far formulated the database state at the system level and defined the consistent transition for internal states. In particular, we have identified the set of consistent primitive modifications and proved that it is the only set we have to be concerned with while dealing with database transitions at the system level. As mentioned in Section 1, database updates are initiated by end users and realized at the system level. Of great importance now is how the database state perceived by the end user should be formulated so that there exists a clear picture of what components of the internal state should be modified when realizing a database update at the end-user level.

2.3. *PERCEPTIBLE STATES OF A DATABASE*

A relational database is usually perceived by its end users as nothing but a set of tables. Each table, called a relation, is associated with a relation scheme. A perceptible state is therefore defined as the composite of a set of table structures (relation schemes) and a set of table contents (relation contents) defined accordingly.

Definition 2.12. A *perceptible state* of a database is denoted by (S_ρ, C_ρ) where S_ρ is the set of relation schemes of the database and C_ρ is the set of relation contents. The set of all perceptible states is denoted by $\wp$.

The purpose of the above definition is to describe the perceptible state in a simple but, perhaps, more adequate way, as far as the end user is concerned. Most database updates would be conceived and issued against it. However, before discussing database updates and their realization at the system level, let us clarify how the perceptible state is formed from the internal state. First, assume that there is no view ever defined. The set of relation schemes and that of relation contents will then be exactly the same as the conceptual schema and the database extension (at the system level), respectively. Now, with a number of views defined, additional tables are introduced into the perceptible state. In other words, in addition to base relations that are formed by the conceptual schema S and the database extension E, there are also relations in the perceptible state that are derived by the set of view definitions V.

Furthermore, it is possible that the access to some information in the database might be prohibited for unauthorized users. Therefore, we can define a filter ξ for each group of end users. This filter masks or coalesces[1] all the relations and views that should be invisible to end users. The components in the perceptible state (S_ρ, C_ρ) can be formulated as follows: $S_\rho = \xi(S \cup V(S))$ and $C_\rho = \xi(E \cup V(E))$. Mathematically speaking, every relation perceived by the end user is in fact a view. Let us suppose that the filter is rarely changed and that the set of integrity constraints I in the internal state is also rather static.

Definition 2.13. Let ξ and I be the given filter and the integrity constraint set, respectively. An internal state (S, E, V, I) is said to *model* a perceptible state (S_ρ, C_ρ), written $(S, E, V, I) \models (S_\rho, C_\rho)$, if and only if $S_\rho = \xi(S \cup V(S))$ and $C_\rho = \xi(E \cup V(E))$.

Note that, given ξ and I, there is a many-to-one relationship between internal states and perceptible states. Given an internal state $s = (S, E, V, I)$, the corresponding perceptible state can be determined as $\text{perc}(s) = (\xi(S \cup V(S)), \xi(E \cup V(E)))$. The reverse is not true, however.

Transitions of perceptible states are due to the change of C_ρ, the perceptible content, as most end users are to modify the content of the database only. Let

[1]Obviously, a filter will mask information which is not to be seen. The situation where ξ is required to coalesce relations will be discussed in Sections 3.1 and 3.3.

Σ_ρ be the set of all perceptible states, and Γ be all the updates against perceptible states, namely, those directly intended by the end user. The transition function for perceptible states is thus a mapping $\delta_\rho : \Sigma_\rho \times \Gamma \to \Sigma_\rho$. The transition function must be defined to portray the change precisely. Let $\delta_\rho((S_\rho, C_\rho), \gamma_\rho) = (S_\rho, C'_\rho)$, where (S_ρ, C_ρ), $(S_\rho, C'_\rho) \in \Sigma_\rho$ and $\gamma_\rho \in \Gamma$. Then C'_ρ must not only completely demonstrate γ_ρ, the intention of the end user against C_ρ, but also reflect the necessary changes associated with γ_ρ. An obvious example is the change of the view extension propagated from that of the base relation on which the view is defined.

2.4. *UPDATE REALIZATION*

From the user's perspective, a database update is to modify the contents C_ρ of the perceptible state. Update realization is therefore the process through which database updates intended at the user level are realized at the system level. Let us first examine the existing approaches for view update in terms of our formulation established so far. Recall that C_ρ is composed of two parts: $\xi(E)$ and $\xi(V(E))$. Updates against $\xi(E)$ are usually performed by modifying E directly. Updates against $\xi(V(E))$, namely, view updates, are translated into underlying database updates, and thus are realized by changing E. This practice, as a whole, only amounts to the primitive modification ($\psi_S = \emptyset$, $\psi_E, \psi_V = \emptyset$, $\psi_I = \emptyset$) we have defined earlier. So, if a database update is not translatable, it must have been either illegitimate or so much more involved that other primitive modifications are required.

In order to accommodate more flexible updates, i.e., to also accept non-translatable but legitimate view updates, we propose to formulate the problem of view update as in Figure 2. A database update, in terms of perceptible states, is a transition $\delta_\rho((S_\rho, C_\rho), \gamma_\rho) = (S_\rho, C'_\rho)$ such that there exist internal states $s \models (S_\rho, C_\rho)$, $s' \models (S_\rho, C'_\rho)$ with $s, s' \in \Sigma_C$. Of course, as mentioned earlier, it is not necessarily true that C'_ρ is exactly what the user has intended or foreseen. The change of view extension propagated from that of base relations is an example. Nevertheless, it is clear that, to realize a database update which is specified at the end-user level, a database system must interpret such intention by the internal modifications defined earlier.

DEFINITION 2.14. Let Γ_ρ be a given set of (perceptible) database updates, M be the set of all modifications, and $M^+ = M \cup \{\text{null}\}$. An *update realization algorithm* $\mathscr{A}$ is a mapping $\mathscr{A} : \Gamma_\rho \to M^+$. Let δ be the internal transition function, $s \in \Sigma$ be a given internal state, and $\gamma_\rho \in \Gamma_\rho$ be a perceptible database update. A realization $\mathrm{A}(\gamma_\rho)$ *properly reflects* the intention of γ_ρ if and only if

(1) $\delta_\rho(\text{perc}(s), \gamma_\rho) = \text{perc}(\delta(s, \mathscr{A}(\gamma_\rho)))$.

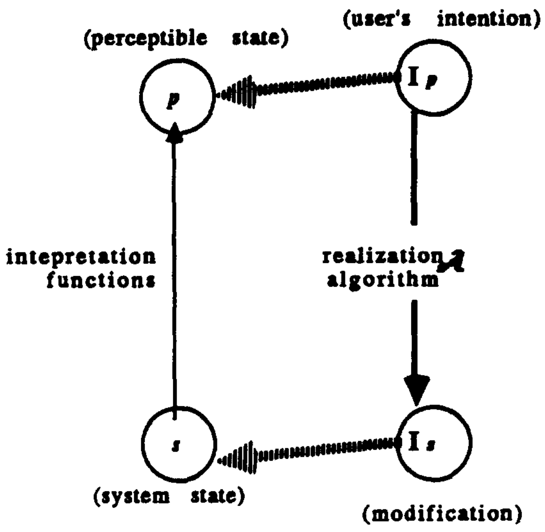

Fig. 2. Update realization.

(2) $\delta(s, \mathscr{A}(\gamma_p)) \in \Sigma_C$, and

(3) no other modification in M contained in $\mathscr{A}(\gamma_p)$ satisfied (1) and (2).

γ_p is said to be realized by $\mathscr{A}$ on s if $\mathscr{A}(\gamma_p) \in M$ and $\mathscr{A}(\gamma_p)$ properly reflects γ_p. γ_p is not realized by $\mathscr{A}$ on s if $\mathscr{A}(\gamma_p) = \text{null}$. For every $\gamma_p \in \Gamma_p$, $\mathscr{A}(\gamma_p)$ must either realize or not realize γ_p.

DEFINITION 2.15. For any subset N of M^+, the *preimage* of N for an update realization algorithm $\mathscr{A}$, denoted $\mathscr{A}^{\leftarrow}(N)$, is the set of (perceptible) database updates $\{\gamma_p \in \Gamma_p - \mathscr{A}(\gamma_p) \in N\}$.

DEFINITION 2.16. An update realization algorithm A *covers* another update realization algorithm $\mathscr{A}'$, denoted $\mathscr{A} \sqsupseteq \mathscr{A}'$, if and only if $\mathscr{A}^{\leftarrow}(\{\text{null}\}) \subseteq \mathscr{A}'^{\leftarrow}(\{\text{null}\})$.

So an ideal update realization algorithm is an update realization algorithm which is not covered by others.

3. A MECHANISM FOR FLEXIBLE VIEW UPDATES

The formulation established so far allows us to first describe the notion of a *generalized* relation, which leads to a straightforward but rather naive approach. We then discuss the update anomalies associated with this approach and set the tone for the next section on view normalization.

3.1. *GENERALIZED RELATIONS FOR NATURAL NAMING*

Our formulation implies that there is a perceptible state on which most user updates are specified, while such updates will be realized on the internal state. It also implies that a realization algorithm does not have to be limited by the particular primitive modification which modifies only the database extension E of the internal state.

For naming ease, as described in Section 1, the natural name for each relation should be the only name an end user is concerned with; he/she retrieves as well as updates a relation through this single name, be it a base relation, a view, or both. In other words, a relation name in the perceptible state should be general enough to stand for a real relation, a derived relation, or both if needed. Let r_{real} and r_{virtual} denote the real relation and the virtual relation, respectively. The relation name r now refers, in general, to

$$r_{\text{real}} \cup r_{\text{virtual}_1} \cup r_{\text{virtual}_2} \cup \cdots \cup r_{\text{virtual}_n},$$

where n is the number of distinct definitions for the virtual part of this relation; $r_{\text{virtual}_1}, r_{\text{virtual}_2}, \cdots$, and r_{virtual_n} are derived relations; and r_{real} is the collection of facts stated directly. These parts are referred to by different names internally and are coalesced by the filter ξ into a single relation when their common external name is queried. This filter also hides all the constituent relations, if any, so that only the generalized relation is perceived by database users. It thus relieves users of the burden in discriminating various parts of a relation.

Now, in terms of database transitions, this notion of generalized relation could imply the following. An update against the base relation r will be performed, on r_{real}, directly. Updates against any of the views, r_{virtual_i}, if translatable, will be realized by updates on the constituent base relations. And a nontranslatable but legitimate view update against relation r will be handled by modifying r_{real}. That is, even if r starts as a view, it is now a generalized relation with the new r_{real} component added.

3.2. *DIRECT VIEW UPDATES*

What we are about to suggest is a straightforward approach as shown in Figure 3. If the view update is translatable, it is realized by updates onto the constituent base relations. For the very first nontranslatable view update, a nonprimitive (internal) modification $(\psi_S, \psi_E, \emptyset, \emptyset)$ is performed: first, to

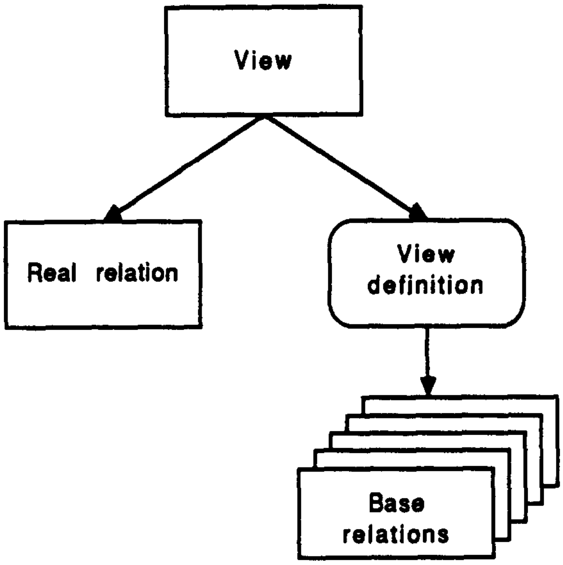

Fig. 3. View flock (a single real relation).

create a relation r_{real}, and second, to insert the new view tuple into it.[2] The creation of this additional relation is an internal transition through the primitive modification $(\psi_S, \emptyset, \emptyset, \emptyset)$, while the insertion of the view tuple becomes a tranditional transition through $(\emptyset, \psi_E, \emptyset, \emptyset)$. In general, the possibility of a modification $(\emptyset, \psi_E, \emptyset, \emptyset)$ for each update request should be examined first and, if needed, followed by the nonprimitive modification $(\psi_S, \psi_E, \emptyset, \emptyset)$.

The following example shows how this approach works.

EXAMPLE 3.1. Let p and q be the *father-child* relation and the ***husband-wife*** relation, respectively. Let the instances of these two relations be

p	*father*	*child*
	"Carl"	*"David"*
	"Frank"	*"George"*
	"Irvine"	*"John"*

q	*husband*	*wife*
	"David"	*"Eva"*
	"George"	*"Helen"*
	"John"	*"Karen"*

Let the *father_in_law-daughter_in_law* relation r be defined as

$$r = \delta_{father,\, wife \leftarrow father\_in\_law,\, daughter\_in\_law} \Pi_{father,\, wife}(p[child = husband]\, q).$$

[2]The issues of deletion and modification will be discussed later.

where δ is the operator that renames attributes. Thus, r can be derived as

r	*father_in_law*	*daughter_in_law*
	"*Carl*"	"*Eva*"
	"*Frank*"	"*Helen*"
	"*Irvine*"	"*Karen*"

Assuming that we know Adam is Barbara's father_in_law, but we fail to know the name of Adam's son who is Barbara's husband, the internal state can then keep this information at

r_{real}	*father_in_law*	*daughter_in_law*
	"*Adam*"	"*Barbara*"

Notice that the subscript "real" is to indicate that it is the additional real relation defined by the primitive modification $(\psi_S, \emptyset, \emptyset, \emptyset)$. When relation r is queried, the union of tuples in r_{real} and tuples derived from p and q will be referred to. In this example, it is

r	*father_in_law*	*daughter_in_law*
	"*Adam*"	"*Barbara*"
	"*Carl*"	"*Eva*"
	"*Frank*"	"*Helen*"
	"*Irvine*"	"*Karen*"

From now on, whenever a nontranslatable *father_in_law-daughter_in_law* tuple must be inserted, only a primitive modification $(\emptyset, \psi_E, \emptyset, \emptyset)$ needs to be performed, since r_{real} has become available.

As shown in the above example, in addition to translatable view updates, this approach provides new facilities to accommodate nontranslatable view updates. It is therefore a "better" realization algorithm, denoted $\mathscr{A}_L$, which *covers* the realization algorithm $\mathscr{A}_T$ for translatable view updates. Let the realization algorithm for simple relational databases be $\mathscr{A}_R$. Then the relationship among these three algorithms is $\mathscr{A}_R \subseteq \mathscr{A}_T \subseteq \mathscr{A}_L$, as illustrated in Figure 4.

3.3. *VIEW-UPDATE ANOMALIES*

There are problems with the approach just described. Its being straightforward makes it look somehow naive, but is not where the real problems are.

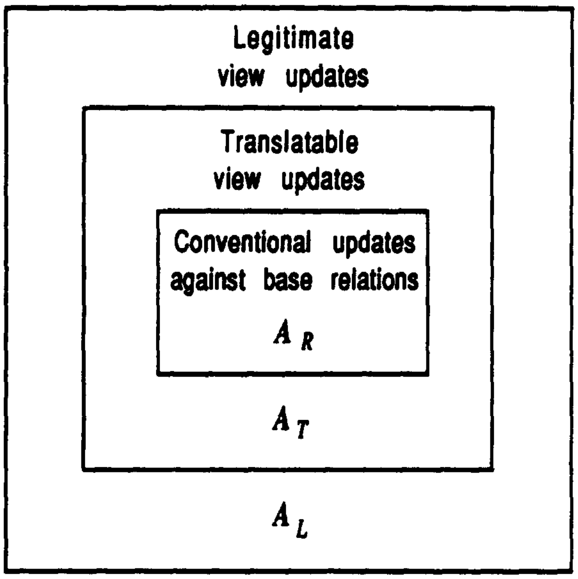
Legitimate view updates
Translatable view updates
Conventional updates against base relations
A_R
A_T
A_L

Fig. 4. Relationship of realization power among updating paradigms.

One of them is due to the transitivity of functional dependencies; there might be inconsistencies between tuples derived from view definitions and those directly stored. For example:

EXAMPLE 3.2. Let $R_1[\underline{A}B]$, $R_2[\underline{B}C]$, $R_3[\underline{C}D]$, and $R_4[\underline{D}E]$ be schemes of base relations that are in 3NF with respect to the set of FDs $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow E\}$. A view is defined as $V[A\ C\ E] = \pi_{ACE}(R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4)$. Let $r_1 = \{\langle a: A\, b: B\rangle\}$, $r_2 = \{\langle b: B c: C\rangle\}$, $r_3 = \{\langle c: C d: D\rangle\}$, and $r_4 = \{\langle d: D e: E\rangle\}$ be single-tuple relations over R_1, R_2, R_3, and R_4, respectively. Note that the view $v(V) = \{\langle a: A c: C e: E\rangle\}$ can be derived. Now insert a new tuple $\langle a,: A c: C e,: E\rangle$ into view $v(V)$. Since it cannot be properly realized by inserting corresponding tuples to r_1, r_2, r_3, and r_4, it is instead stored as a real tuple in $v(V)$. However, $F \models \{A \rightarrow CE, C \rightarrow E\}$, and $\{A \rightarrow CE, C \rightarrow E\}$ is a set of FDs which must be satisfied by all tuples in the database, including $v(V)$. Accordingly, the request to insert the tuple $\langle a,: A c: C e,: E\rangle$ to view $v(V)$ should have been rejected, because $C \rightarrow E$ would otherwise be violated.

The example above resembles a common problem in database design, *update anomaly* [4–6, 12, 15, 16], which is usually handled by normalization. The difference here is that, instead of the database scheme, it is the view scheme that needs to be normalized.

So the approach above must be modified to store information in a cluster of normalized relations, called the *view flock* in Figure 5, rather than a single *view* relation. Let us again examine this modified mechanism in terms of internal state transitions. Originally, without the view flock, the nontranslatable but legitimate view update was realized by introducing an additional real relation through the internal transition $(S, E, V, I) \rightarrow ((S \cup R_v), (E \cup r_v), V, I)$. Assuming that S is in 3NF (the third normal form), R_v is, however, not necessarily in 3NF. As a view to begin with, R_v is of course not a scheme on which the entire set of FDs in I should, or could, be enforced.

It is meaningful to decompose R_v into a 3NF view flock with respect to only those FDs that are enforceable on R_v. Thereby, the relations in the 3NF view flock can preclude most of the update anomalies stated above. Once decomposed, this "view relation" R_v is no longer a real relation. Instead, it is a virtual relation derived from those decomposed relations in the view flock. In other words, not only will S and E be modified to include relations in the view flock, V will also be changed by adding a view definition for R_v. Such transitions are associated with internal modifications of type $(\psi_S, \psi_E, \psi_V, \emptyset)$, where ψ_S, ψ_E, and ψ_V are nonempty. Meanwhile, the filter ξ will be modified to hide from the perceptible state all the real relations in the view flock. As indicated earlier, the filter ξ is indeed used to coalesce all the constituents of a relation and to hide all of them. Since the view flock in its entirety is referred

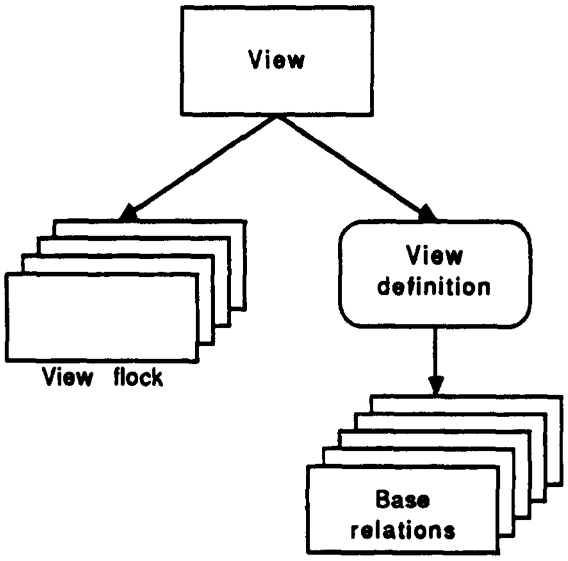

Fig. 5. View flock decomposed into 3NF subschemes.

to as an r_{virtual} rather than r_{real}, additional masks should be used to hide all the decomposed relations in the view flock. In the next section, we shall discuss how to form a view flock as well as other related issues.

4. VIEW FLOCKS AND NORMALIZATION

To decompose a view scheme into a view flock is an interesting problem. Two key algorithms are required: first, an efficient algorithm to extract the FD set enforceable by the view scheme, and second, an efficient algorithm to synthesize the view flock. We will first discuss the second algorithm briefly, because it is already available. We will then discuss the first algorithm which is the main focus of this section. The algorithm proposed here is in fact quite subtle; it finds a set of FDs that is equivalent to the set of all FDs applicable to the view scheme. A couple of related issues will also be discussed.

Given a relation scheme R and an FD set F over R, the *Synthesis* algorithm[3] [13] produces a set of subschemes $S = \{R_1, R_2, \cdots, R_p\}$ over R and a set of *designated keys* K_i for each subscheme $R_i \in S$. This set of subschemes has the following four properties:

(1) $F \equiv \{K_{ij} \rightarrow R_i \mid K_{ij} \in K_i \wedge R_i \in S\}$.

(2) Every subscheme $R_i \in S$ is in 3NF with respect to F.

(3) There is no set of subschemes satisfying properties (1) and (2) with fewer subschemes than S.

(4) For any relation $r(R)$ that satisfies F, $r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \cdots \bowtie \pi_{R_p}(r)$.

The first property ensures that F is enforceable on S. Since K_i is the set of designated keys of subscheme R_i, the instance of the subscheme must always satisfy the key dependencies $K_{i,j} \rightarrow R_i$ for all $K_{i,j} \in K_i$. That is, all the functional dependencies can be enforced by simply checking the designated key dependencies. The last property guarantees the *lossless decomposition* [13].

4.1. EXTRACTING AN ENFORCEABLE FD SET

Normalizing a view into 3NF requires first the *enforceable set* of FDs. The FD set enforceable by a view is the set of FDs among the attributes of the view scheme. The definition of the enforceable set is as follows:

DEFINITION 4.1. Let U be the universal scheme of a database. Given a set of FDs F over U and a scheme $R \subseteq U$, a set of FDs, F_R, is said to be an

[3]Throughout this paper, we refer to the *refined Synthesis* algorithm as the *Synthesis* algorithm.

enforceable set of FDs, with respect to F, on the scheme R if and only if the following condition is satisfied:

For every FD, $X \rightarrow Y$, $F \models X \rightarrow Y$, and $XY \subset R$ if and only if $F_R \models X \rightarrow Y$.

Once an enforceable set is obtained, the task of normalizing the underlying view is similar to that of normalizing a database scheme. A given view will be normalized by applying the same *Synthesis* algorithm on the view scheme and its enforceable set. Thus, the only problem that remains unsolved is the extraction of an enforceable set from the FD set over the entire database scheme.

Finding the enforceable set of a given FD set on a subscheme appears to be inherently exponential, since the number of FDs in the enforceable set could be an exponential function of the number of FDs in the given set. For example:

EXAMPLE 4.1. Let $U = \{A_1, A_2, \cdots, A_n, B_1, B_2, \cdots, B_n, C, X_1, X_2, \cdots, X_{n-1}\}$, $R = \{A_1, A_2, \cdots, A_n, B_1, B_2, \cdots, B_n, C\}$, and $F = \{A_1 \rightarrow X_1, B_1 \rightarrow X_1, X_1 A_2 \rightarrow X_2, X_1 B_2 \rightarrow X_2, X_1 X_2 A_3 \rightarrow X_3, X_1 X_2 B_3 \rightarrow X_3, \cdots, X_1 X_2 \cdots X_{n-1} A_n \rightarrow C, X_1 X_2 \cdots X_{n-1} B_n \rightarrow C\}$. The enforceable set is $F_R = \{A_1 A_2 \cdots A_{n-1} A_n \rightarrow C, A_1 A_2 \cdots A_{n-1} B_n \rightarrow C, A_1 A_2 \cdots B_{n-1} A_n \rightarrow C, A_1 A_2 \cdots B_{n-1} B_n \rightarrow C, \cdots, B_1 B_2 \cdots A_{n-1} A_n \rightarrow C, B_1 B_2 \cdots A_{n-1} B_n \rightarrow C, B_1 B_2 \cdots B_{n-1} A_n \rightarrow C, B_1 B_2 \cdots B_{n-1} B_n \rightarrow C\}$, and $|F_R| = 2^{|F|/2}$.

In this example, since all the FDs in F_R have different LHS attribute sets but identical RHS attribute sets, F_R cannot be further reduced or minimized to any smaller but equivalent FD set. This example shows how an enforceable set can grow exponentially from a particular set of FDs. In reality, such extreme cases hardly happen. Therefore, an efficient algorithm for extracting the enforceable set remains highly desirable.

The method proposed in the following is based on an inferential approach, although an algorithm with a similar flavor but based on resolution has recently been developed elsewhere for different purposes. Gottlob proposed the approach called "reduction by resolution" (RBR) [8], which reduces the given FD set to an enforceable set (called the embedded FD set in [8]) by reducing unnecessary attributes. It resolves the given FD set by each attribute which is not in the designated subscheme. On the contrary, our approach is to derive the enforceable set based on the inference axioms available in database theory.

The algorithm, *Get-Enforceable-Set*, presented below enrolls one particular inference axiom. Among the fundamental inference axioms listed in [13], axioms *Reflexivity* (F1), *Augmentation* (F2), and *pseudotransitivity* (F6) form a set called *Armstrong axioms* which has been proven to be functionally complete. Since axioms F1 and F2 produce only trivial FDs, F6 alone is sufficient for deriving the enforceable set. However, for efficiency purposes, additivity (F3) and projectivity (F4) have been implemented in functions *COMBINE* and *SPLIT*, respectivley. Since the method for finding the minimum cover is available as part of the *Syntheses* algorithm, this algorithm assumes that the input set of FDs F is already a *minimum set of FDs* [13].

In this algorithm, step 1 converts the given set of FDs to an equivalent set $\bar{F}$, in which every FD contains only one attribute on the RHS. Steps 2 and 6 extract the FDs in $\bar{F}$ that apply to the scheme V. Steps 3, 4, and 5 eliminate the FDs which appear to be useless as far as deriving the enforceable set is concerned. Steps 7 and 8 apply pseudotransitivity (F6) repeatedly to extract the remaining FDs of the enforceable set (Figure 6).

The complete algorithm is as follows:

```
Algorithm Get-Enforceable-Set:
  Function SPLIT (F: set of FDs):
      Convert the given FD set into an equivalent FD set which contains
      only the FDs with singleton on the right-hand side of "→".
    1. For each FD $X \rightarrow A_1 A_2 \cdots A_n \in F$, where the $A_i$'s are single attributes, split it into
       $X \rightarrow A_1,\ X \rightarrow A_2, \cdots,\ X \rightarrow A_n$.
    2. Return the obtained FD set.
  Function COMBINE (F: set of FDs):
      Convert the given FD set into an equivalent FD set in which every
      FD contains distinct set of attributes on the left-hand side of
      "→".
```

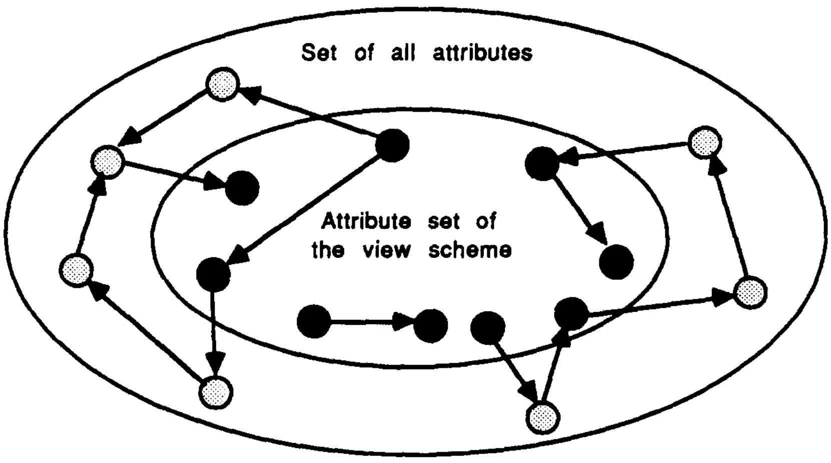

Fig. 6. Enforceable FD set.

```
    1. Withdraw all the FDs which have identical left-hand-side attribute set,
       $X \to A_1$, $X \to A_2, \cdots$, $X \to A_n$, from $F$, and insert the combined
       FD, $X \to A_1 A_2 \cdots A_n$, into $F$.
    2. Repeat step 1 until no FD pair in $F$ have the same left-hand side
       attribute set.
    3. Return $F$.
Function SUBSTITUTE (F: set of FDs, S: set of sets of attributes, A:
       attribute):
       Apply pseudotransitivity to obtain new FDs.
    1. For every FD in $F$ with attribute $A$ on the left-hand side, $XA \to Y$;
       (1a) withdraw it from $F$, and
       (1b) for every set of attributes $W \in S$, insert $XW \to Y$ into $F$.
    2. Return $F$.
Procedure:
    input:  $F$, the minimum set of FDs.
            $V$, the scheme of view, a set of attributes.
    output: $F_v$, a set of FDs.
    begin:
    step 1: $\bar{F} := SPLIT(F)$.
    step 2: Divide $\bar{F}$ into three disjoint sets,
            $\bar{F}_0 = \{X \to A \mid X \to A \in \bar{F} \wedge A \in V \wedge X \subset V\}$,
            $\bar{F}_1 = \{X \to A \mid X \to A \in \bar{F} \wedge A \in V \wedge X \not\subset V\}$, and
            $\bar{F}_2 = \{X \to A \mid X \to A \in \bar{F} \wedge A \notin V\}$.
    step 3: Collect all the right-hand-side attributes of FDs in $\bar{F}_2$ to form an
            attribute set $S_R$.
            Eliminate FD $X_1 X_2 \cdots X_k \to A$ from $\bar{F}_1$, if there exists $X_i \notin S_R$,
            $1 \leqslant i \leqslant k$.
            Eliminate FD $X_1 X_2 \cdots X_k \to A$ from $\bar{F}_2$, if there exists $X_i \notin S_R$
            $\cup V$, $1 \leqslant i \leqslant k$.
    step 4: Collect all the left-hand-side attributes of FDs in $\bar{F}_1$ and $\bar{F}_2$ to
            form an attribute set $S_L$.
            For every FD $X_1 X_2 \cdots X_k \to A \in \bar{F}_2$, if $A \notin S_L$ then eliminate
            this FD from $\bar{F}_2$.
    step 5: Repeat steps 3 and 4 until there is no further FD that can be
            eliminated.
    step 6: $F_0 := COMBINE(\bar{F}_0)$.
            $F_1 := COMBINE(\bar{F}_1)$.
            $G := F_1 \cup \bar{F}_2$.
    step 7: Withdraw an attribute $A$ from $S_R$.
            Withdraw all the FDs with $A$ on the right-hand side, $X_1 \to A$,
            $X_2 \to A, \cdots$, $X_n \to A$, from $G$.
            Collect all the left-hand-side attributes of these FDs to form an
            attribute set $S = \{X_1, X_2, \cdots, X_n\}$.
            $G := SUBSTITUTE(G, S, A)$.
    step 8: Repeat step 7 until $S_R = \emptyset$.
    step 9: $F_V := COMBINE\ (F_0 \cup G)$.
    end.
```

4.2. *OTHER RELATED ISSUES*

The now available *Get-Enforceable-Set* algorithm together with the *Synthesis* algorithm provides a systematic method to normalize a view scheme into a 3NF view flock. We can apply the *Get-Enforceable-Set* algorithm to the view scheme so as to obtain the enforceable FD set. We can then apply the *Synthesis* algorithm to the enforceable FD set to generate the 3NF view flock. Nevertheless, there remain a couple of interesting issues.

Overlap between Normalized View Flocks and Existing Relations

Some of the schemes in the 3NF view flock may coincide or overlap with the schemes of some base relations (Figure 7). Practically, these subschemes need not be added as new real relations. Instead, information that should be stored in them can be forwarded to the corresponding base relations. In other words, this indicates the situations where, although the view update as a whole is not translatable, the updates on some subschemes might be translatable. Therefore, the view flock must keep track of the linkage to the translatable portion of the update while storing only the nontranslatable portion directly. Since there are various ways that two relation schemes can overlap with each other, further study of this issue is needed.

View Deletion and Modification

In this paper, we have focused on view *insertion* only. Our rationale draws from mathematical logic, which happens to be the theoretical foundation of relational databases. A view tuple is considered as a predicate derived by a rule (view definition) from a pair of arbitrary premises (tuples). Inserting a tuple into a view is considered as stating a predicate, which happens to be the goal of a rule, with constant terms. For instance, if a rule, equivalent to the view

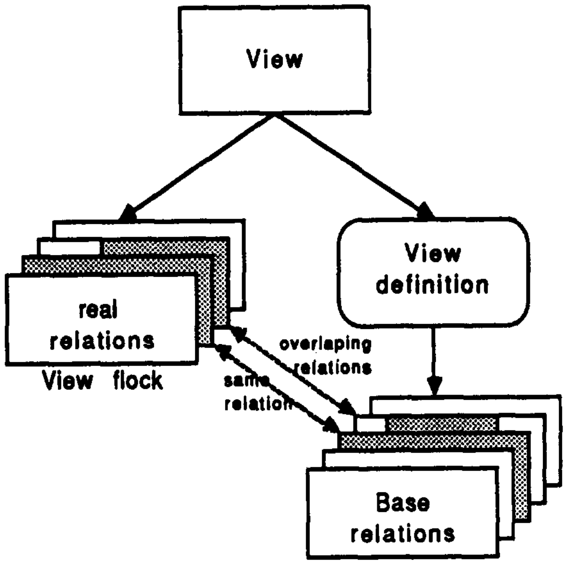

Fig. 7. Duplicated relations in view flock and base relations.

definition for r in Example 3.1, is specified as

$$father(x, y) \wedge husband(y, z) \rightarrow father\_in\_law(x, z)$$

then the predicate *father_in_law*("*Adam*", "*Barbara*") should be able to be stated even though the son of *Adam* and the husband of *Barbara* are unknown.

However, to delete or modify a view tuple is quite different. If predicates *father*("*Carl*", "*David*") and *husband*("*David*", "*Eva*") exist, the predicate *father_in_law*("*Carl*", "*Eva*") will be derived by the rule stated above. Deleting the predicate *father_in_law*("*Carl*", "*Eva*") will conflict with the fact *father*("*Carl*", "*David*") *husband*("*David*", "*Eva*") and $father(x, y) \wedge husband(y, z) \rightarrow father\_in\_law(x, z)$. Similarly, to modify, rather than delete, this predicate will cause the same conflict, since *father_in_law*("*Carl*", "*Eva*") will no longer exist regardless of the exact modification. This is the main reason why we have not been able to support view deletion and modification.

5. CONCLUSION

In order to deal with nontranslatable but legitimate view updates, we have suggested a more comprehensive formulation for the problem of view updates. Two distinct database states have been identified: the internal state denotes the system level where view updates are realized, while the perceptible state denotes the end-user level where view updates are usually specified. By formulating it as the more general problem of update realization, we are able to state more comfortably what types of internal transition and modification are involved in realizing each view update by each different update mechanism. We first introduced a straightforward update mechanism, which is further modified by view normalizations based on normalization theory. The problem of extracting enforceable FD sets for view schemes has been defined and solved.

In addition to issues discussed earlier, there remain a number of interesting problems. For instance, we are yet to take into account view definitions in order to identify the implication for update verification of each relational operation such as selection. The integrity constraint is also an important issue along the direction of the proposed approach. In order to retain the consistency of the database, more integrity constraints are required when additional real relations associated with views are created. There is a need to identify and minimize the required integrity constraints. Needless to say, view definitions again play an important role in identifying the required integrity constraints.

Many thanks to the referees for their useful suggestions.

REFERENCES

1. F. Bancilhon and N. Spyratos, Update semantics of relational views, *ACM Trans. Database Systems* 6(4):557-575 (Dec. 1981).
2. S. S. Cosmadakis and C. H. Papadimitriou, Updates of relational views, *J. Assoc. Comput. Math.* 31(4):742-760 (Oct. 1984).
3. U. Dayal and P. A. Bernstein, On the updatability of relational views, in *Proceedings of the 4th VLDB Conference*, West Berlin, 13-15 Sept. 1978, pp. 368-377.
4. C. J. Date, *An Introduction to Database Systems*, Addison-Wesley, vol. 12, Reading, Mass. 1983.
5. C. J. Date, *Relational Databases*, Addition-Wesley, 1986.
6. R. A. Frost, *Introduction to Knowledge Base Systems*, Macmillan, New York, 1986.
7. A. L. Furtado, K. C. Sevcik, and C. S. Dos Santos, Permitting updates through views of databases, *Inform. Systems* 4:269-283 (1979).
8. G. Gottlob, Computing covers for embedded functional dependencies, in *ACM Symposium on Principles of Database Systems*, Mar. 1987, pp. 58-69.

9. T. Imielinski, Query processing in deductive databases with incomplete information, in *Proceedings of the ACM-SIGMOD Conference on Management of Data*, Washington, May 1986, pp. 268-280.
10. A. M. Keller, The role of semantics in translating view updates, *IEEE Computer* 19(1):63-73 (Jan. 1986).
11. A. M. Keller and M. W. Wilkins, On the use of an extended relational model to handle changing incomplete information, *IEEE Trans. Software Engrg.* 11(7):620-633 (July 1985).
12. H. F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1986.
13. D. Maier, *Theory of Relational Databases*, Computer Science Press, Rockville, Md., 1983.
14. N. Spyratos, Translation structures of relational views, in *Proceedings of the 6th VLDB Conference*, Montreal, 1-3 Oct. 1980, pp. 411-416.
15. T. J. Teorey and J. P. Fry, *Design of Database Structures*, Prentice-Hall, Englewood Cliffs, N.J., 1982.
16. J. D. Ullman, *Principles of Database Systems*, 2nd ed., Computer Science Press, Rockville, Md., 1983.
17. J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. 1, Computer Science Press, Rockville, Md., 1988.

Received 10 August 1989; revised 10 November 1989, 20 March 1990