Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, May 20-22, 1991, pages 300-313.

A Novel Decomposition of Multilevel Relations Into Single-Level Relations*

Sushil Jajodia and Ravi Sandhu

Center for Secure Information Systems
and
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030-4444

Abstract

In this paper we give a new decomposition algorithm that breaks a multilevel relation into single-level relations and a new recovery algorithm which reconstructs the original multilevel relation from the decomposed single-level relations. There are several novel aspects to our decomposition and recovery algorithms which provide substantial advantages over previous proposals: (1) Our algorithms are formulated in the context of an operational semantics for multilevel relations, defined here by generalizing the usual update operations of SQL to multilevel relations. (2) Our algorithms, with minor modifications, can easily accommodate alternative update semantics which have been proposed in the literature. (3) Our algorithms are efficient because recovery is based solely on union-like operations without any use of joins. (4) Our decomposition is intuitively and theoretically simple giving us a sound basis for correctness. In this paper we also argue that some of the alternate update semantics which have been proposed for multilevel relations should be available as options, but should certainly not be made an integral part of the data model.

1 INTRODUCTION

In recent years, there have been several efforts to build multilevel secure relational database management systems (DBMSs). A major issue is how access classes are assigned to data stored in relations. The proposals have ranged from assigning access class to relations (as in [9]), assigning access classes to individual tuples in a relation (as in [6]), or assigning access classes to individual attributes of a relation (as in [11]).

Unlike these proposals, in the Secure Data Views (SeaView) project security classifications are assigned to individual data elements of the tuples of a relation [3, 4, 17]. For example see figure 1. Subjects having different clearances see different versions of the multilevel relation: A user having a clearance at an access class c sees only that data which lies at class c or below. Thus, a user with Top Secret clearance will see the entire relation in figure 1 while a user having Confidential clearance will see the filtered relation given in figure 2.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Mine	C	Sirus	C	C
Ent	U	Spy	S	Rigel	S	S
Ent	U	Coup	TS	Orion	TS	TS

Figure 1: A multilevel relation SOD

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Mine	C	Sirus	C	C

Figure 2: Confidential view of SOD

Multilevel relations in SeaView exist only at the logical level. In reality multilevel relations are decomposed into a collection of single-level base relations which are then physically stored in the database. Completely transparent to users, multilevel relations are reconstructed from these base relations on user demand. There are several practical advantages of being able to decompose and store a multilevel relation by a collection of single-level base relations. In particular the underlying trusted computing base (TCB) can enforce mandatory controls with respect to the single-level base relations. This allows the DBMS to mostly run as an untrusted application with respect to the underlying TCB.

In SeaView, the decomposition of multilevel relations into single-level ones is performed by applying two different types of fragmentation: *horizontal* and *vertical*. Thus the multilevel relation in figure 1 will be stored

*This work was partially supported by the U.S. Air Force, Rome Air Development Center through subcontract #C/UB-49;D.O.No.0042 of prime contract #F-30602-88-D-0026, Task B-O-3610 with CALSPAN-UB Research Center.

$D_{1,U}$

SHIP	
Ent	U

(a) Primary Key Group Relation

$D_{2,U}$

SHIP		OBJ	
Ent	U	Exp	U

$D_{3,U}$

SHIP		DEST	
Ent	U	Talos	U

$D_{2,C}$

SHIP		OBJ	
Ent	U	Mine	C

$D_{3,C}$

SHIP		DEST	
Ent	U	Sirus	C

$D_{2,S}$

SHIP		OBJ	
Ent	U	Spy	S

$D_{3,S}$

SHIP		DEST	
Ent	U	Rigel	S

$D_{2,TS}$

SHIP		OBJ	
Ent	U	Coup	TS

$D_{3,TS}$

SHIP		DEST	
Ent	U	Orion	TS

(b) Attribute Group Relations

Figure 3: SeaView decomposition of figure 1 into 9 single-level base relations

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Exp	U	Sirus	C	C
Ent	U	Mine	C	Talos	U	C
Ent	U	Mine	C	Sirus	C	C
Ent	U	Exp	U	Rigel	S	S
Ent	U	Mine	C	Rigel	S	S
Ent	U	Spy	S	Talos	U	S
Ent	U	Spy	S	Sirus	C	S
Ent	U	Spy	S	Rigel	S	S
Ent	U	Exp	U	Orion	TS	TS
Ent	U	Mine	C	Orion	TS	TS
Ent	U	Spy	S	Orion	TS	TS
Ent	U	Coup	TS	Talos	U	TS
Ent	U	Coup	TS	Sirus	C	TS
Ent	U	Coup	TS	Rigel	S	TS
Ent	U	Coup	TS	Orion	TS	TS

Figure 4: SeaView recovery algorithm applied to the single-level base relations of figure 3

as *nine* single-level fragments (one primary key group relation and eight attribute group relations) shown in figure 3. This leads to many problems with the SeaView decomposition and recovery algorithms:

1. *Repeated Joins.* The vertical fragmentation used in SeaView results in single-level relations that consist of the key attribute, a single non-key attribute and their classifications attributes. This means that nearly all queries involving multiple attributes necessitate repeated (left outer) joins of several single-level relations. It is well-known that join is an expensive operation and should be avoided whenever possible (see, for example, [20]).

2. *Spurious Tuples.* Whenever a relation is stored as one or more fragments, it must be possible to reconstruct the original relation exactly from fragments. This, however, is not the case with the SeaView decomposition. When the SeaView recovery algorithm is applied to the single-level relations in figure 3, a Top Secret user will be shown the relation given in figure 4. While the original Top Secret instance in figure 1 describes four missions for the Enterprise, a Top Secret user will see the sixteen missions of figure 4 using the SeaView approach.

3. *Incompleteness.* The SeaView decomposition puts severe limitations on the expressive capability of the database. Several instances that have realistic and useful interpretations cannot be realized in SeaView [13, 15].

4. *Left Outer Joins.* The SeaView recovery algorithm is based on the left outer join of relations. There are many theoretical complications and pitfalls which arise with outer joins (see, for example, [2]).

In [13], Jajodia and Sandhu have given a modified version of the SeaView decomposition and recovery algorithms. They store the relation in figure 1 as a collection of *twelve* single-level relations (four primary key group relations and eight attribute group relations) given in figure 5. Their recovery algorithm when applied to these single-level base relations yields exactly the original instance SOD in figure 1. While their algorithms eliminate the last three problems, the first problem remains: satisfying queries involving multiple attributes requires taking repeated natural joins of several single-level relations.

In this paper, we take a fresh look at the decomposition algorithm that breaks a multilevel relation into single-level ones and the recovery algorithm which reconstructs the original multilevel relation from the decomposed single-level relations. We give decomposition and recovery algorithms that have several advantages over previous algorithms [13, 17]:

1. Unlike the previous algorithms, the new decomposition and recovery algorithms are based on operational semantics for the update operations on multilevel relations. The semantics of multilevel relations are defined here by generalizing the usual update operations of SQL.

2. Our algorithms, with minor modifications, can easily accommodate alternative update semantics which have been proposed in the literature. It is even possible to keep the decomposition fixed and vary the recovery algorithms to realize these alternate semantics.

3. Our algorithms are computationally efficient because the new decomposition uses only horizontal fragmentation to break multilevel relations into single-level ones. The decomposition for the relation in figure 1 is shown in figure 6. Since the decomposition does not require any vertical fragmentation, it is possible to reconstruct a multilevel relation from the underlying single-level base relations without having to perform any (left or natural) joins; only unions are required to be taken.

4. The recovery and decompositions are simple to state and prove correct.

This paper is organized as follows. In section 2, we begin by giving basic definitions related to multilevel relations, and then we state four core integrity requirements which we feel must be satisfied by all multilevel relations. In section 3, we examine the semantics of various update operations in the context of multilevel relations. To this end, the familiar INSERT, UPDATE and DELETE operations of SQL are suitably generalized to deal with polyinstantiation. In section 4, we give our new decomposition and recovery algorithms that preserve the update semantics proposed in section 3. In section 5, several examples are given to illustrate the behavior of the update semantics as well as the decomposition and recovery algorithms. In section 6 we show how our algorithms, with minor modifications, can easily accommodate alternative update semantics which have been proposed in the literature. In section

$D_{1,U}$

SHIP		C_2	C_3
Ent	U	U	U

$D_{1,C}$

SHIP		C_2	C_3
Ent	U	C	C

$D_{1,S}$

SHIP		C_2	C_3
Ent	U	S	S

$D_{1,TS}$

SHIP		C_2	C_3
Ent	U	TS	TS

(a) Primary Key Group Relations

Eight Relations Identical to Figure 3(b)

(b) Attribute Group Relations

Figure 5: Jajodia-Sandhu decomposition of figure 1 into 12 single-level base relations

D_U

SHIP		OBJ		DEST	
Ent	U	Exp	U	Talos	U

D_C

SHIP		OBJ		DEST	
Ent	U	Mine	C	Sirus	C

D_S

SHIP		OBJ		DEST	
Ent	U	Spy	S	Rigel	S

D_{TS}

SHIP		OBJ		DEST	
Ent	U	Coup	TS	Orion	TS

Figure 6: New decomposition of figure 1 into 4 single-level base relations

7 we argue that some of the alternate update semantics which have been proposed should be available as options but should certainly not be made an integral part of the data model. Section 8 concludes the paper.

2 MULTILEVEL RELATIONS

In this section, we briefly review basic definitions and assumptions used with multilevel relations. The reader is assumed to be familiar with basic concepts of relational database theory. A *multilevel relation* consists of the following two parts.

Definition 1 [RELATION SCHEME] A state-invariant multilevel relation scheme is denoted by

$$R(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$$

where each A_i is a *data attribute*† over domain D_i, each C_i is a *classification attribute* for A_i and TC is the *tuple-class* attribute. The domain of C_i is specified by a range $[L_i, H_i]$ which defines a sub-lattice of access classes ranging from L_i up to H_i. The domain of TC is $[\text{lub}\{L_i : i = 1 \ldots n\}, \text{lub}\{H_i : i = 1 \ldots n\}]$, where lub denotes the least upper bound. □

Definition 2 [RELATION INSTANCES] A collection of state-dependent *relation instances*, each of which is denoted by

$$R_c(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$$

one for each access class c in the given lattice. Each instance is a set of distinct tuples of the form

$$(a_1, c_1, a_2, c_2, \ldots, a_n, c_n, tc)$$

where each $a_i \in D_i$ or $a_i = \text{null}$, $c \geq c_i$ and $tc = \text{lub}\{c_i : i = 1 \ldots n\}$. Moreover, if a_i is not null then $c_i \in [L_i, H_i]$. We also require that c_i be defined even if a_i is null, i.e., a classification attribute cannot be null. □

We assume that there is a user specified *(apparent) primary key* AK consisting of a subset of the data attributes A_i. In general AK will consist of multiple attributes.

We now list four integrity requirements which we feel must be satisfied by all multilevel relations. We refer the readers to [16] for their intuitive justification. We use the notation $t[A_i]$ to mean the value corresponding to the attribute A_i in tuple t, and similarly for $t[C_i]$ and $t[TC]$.

Property 1 [Entity Integrity] Let AK be the apparent key of R. A multilevel relation R satisfies entity integrity if and only if for all instances R_c and $t \in R_c$

1. $A_i \in AK \Rightarrow t[A_i] \neq \text{null}$,
2. $A_i, A_j \in AK \Rightarrow t[C_i] = t[C_j]$ (i.e., AK is uniformly classified), and
3. $A_i \notin AK \Rightarrow t[C_i] \geq t[C_{AK}]$ (where C_{AK} is defined to be the classification of the apparent key). □

The first requirement is exactly the definition of entity integrity from the standard relational model and ensures that no tuple in R_c has a null value for any attribute in AK. The second requirement says that all attributes in AK have the same classification in a tuple. This will ensure that AK is either entirely visible or entirely null at a specific access class c. The final requirement states that in any tuple the class of the non-AK attributes must dominate C_{AK}. This rules out the possibility of associating non-null attributes with a null primary key.

Notation. In order to simplify our notation, we will henceforth use A_1 instead of AK to denote the apparent primary key. Thus, for the remainder of this paper $R(A_1, C_1, \ldots, A_n, C_n, TC)$ will denote a multilevel relation scheme with A_1 as the apparent primary key.

At this point it is important to clarify the semantics of null values. There are two major issues: (i) the classification of null values, and (ii) the subsumption of null values by non-null ones. Our requirements are respectively that null values be classified at the level of the key in the tuple, and that a null value is subsumed by a non-null value independent of the latter's classification. These two requirements are formally stated as follows.

Property 2 [Null Integrity] A multilevel relation R satisfies null integrity if and only if for each instance R_c of R both of the following conditions are true.

1. For all $t \in R_c$, $t[A_i] = \text{null} \Rightarrow t[C_i] = t[C_1]$, i.e., nulls are classified at the level of the key.
2. We say that tuple t *subsumes* tuple s if for every attribute A_i, either (a) $t[A_i, C_i] = s[A_i, C_i]$ or (b) $t[A_i] \neq \text{null}$ and $s[A_i] = \text{null}$. Our second requirement is that R_c is *subsumption free* in the sense that it does not contain two distinct tuples such that one subsumes the other. □

We will henceforth assume that all computed relations are made subsumption free by exhaustive elimination of subsumed tuples.

The next property is concerned with consistency between relation instances at different access classes. The filter function maps a multilevel relation to different instances, one for each descending access class in the security lattice. Filtering limits each user to that portion of the multilevel relation for which he or she is cleared.

†In many cases it is useful to have an A_i represent a collection of *uniformly classified* data attributes. This extension requires straightforward modifications to our statements in this paper, which are all formulated in terms of the A_i's being individual data attributes.

Property 3 [Inter-Instance Integrity] A multilevel relation R satisfies inter-instance integrity if and only if for all instances R_c and all $c' \leq c$ we have $R_{c'} = \sigma(R_c, c')$ where the *filter function* σ produces the c'-instance $R_{c'}$ from R_c as follows:

1. For every tuple $t \in R_c$ such that $t[C_1] \leq c'$ there is a tuple $t' \in R_{c'}$ with $t'[A_1, C_1] = t[A_1, C_1]$ and for $A_i \notin A_1$

$$t'[A_i, C_i] = \begin{cases} t[A_i, C_i] & \text{if } t[C_i] \leq c' \\ <\text{null}, t[C_1]> & \text{otherwise} \end{cases}$$

2. There are no tuples in $R_{c'}$ other than those derived by the above rule.
3. The end result is made subsumption free by exhaustive elimination of subsumed tuples. □

Finally we have the following polyinstantiation integrity constraint which prohibits polyinstantiation within a single access class.

Property 4 [Polyinstantiation Integrity] A multilevel relation R is said to satisfy polyinstantiation integrity (PI) if and only if for every R_c we have for all A_i

$$A_1, C_1, C_i \rightarrow A_i$$ □

This property stipulates that the user-specified apparent key A_1, in conjunction with the classification attributes C_1 and C_i, functionally determines the value of the attribute A_i. In other words the real primary key of the relation is $A_1, C_1, C_2, \ldots, C_n$.

There are other definitions of the PI property which have been proposed. They all require the functional dependency (fd) component given above, but in addition impose some other condition. For instance the SeaView definition of PI has two requirements: in addition to the fd component it has a second, multivalued dependency (mvd) component [3, 17]. As another example, Lunt and Hsieh [18] have recently given a definition of PI which has the usual fd requirement plus a dynamic mvd requirement. We will return to these two cases later in the paper.

3 UPDATE OPERATIONS

In this section, we give a formal operational semantics for the three update (insert, update, and delete) operations on multilevel relations. Due to space constraints we have chosen to give an abstract description and complete formal definitions first and defer consideration of examples to section 5.

In developing the update semantics we are motivated by the following principles.

1. The update operations should be as close to standard SQL as possible.
2. An update should result in polyinstantiation only when absolutely required for closing signaling channels[‡] or for deliberately establishing cover stories.[§] Moreover, the fewest possible tuples should be introduced in such cases.

Consider a user logged on at access class c. We also refer to such a user as a c-user. Now a c-user directly sees and interacts with the c-instance R_c.[¶] From the viewpoint of this user the remaining instances of R can be categorized into three cases: those strictly dominated by c, those that strictly dominate c and those incomparable with c. The following notation is useful for ease of reference to these three cases.

$$\begin{array}{lcl} R_{c'<c} & \equiv & R_{c'}, \text{ such that } c' < c \\ R_{c'>c} & \equiv & R_{c'}, \text{ such that } c' > c \\ R_{c'\sim c} & \equiv & R_{c'}, \text{ such that } c' \text{ incomparable with } c \end{array}$$

Security considerations, and in particular the $\star$-property, dictate that a c-user cannot insert, update, or delete a tuple, directly or indirectly (as a side-effect) in any $R_{c'<c}$ or $R_{c'\sim c}$. Any effects of these operations must be confined to those tuples in R_c with tuple class equal to c, and in view of the inter-instance property, these changes must be properly reflected in the instances $R_{c'>c}$. In general this may require the addition, modification or removal of some tuples in $R_{c'>c}$ whose tuple class strictly dominates c. Unfortunately, there are several different ways to do this while maintaining inter-instance integrity. This fact complicates the semantics of the insert, update and delete operations.

3.1 The INSERT Statement

The INSERT statement executed by a c-user has the following general form, where the c is implicitly determined by the the user's login class.

$$\begin{array}{ll} \text{INSERT} & \\ \text{INTO} & R_c[(A_i[, A_j] \ldots)] \\ \text{VALUES} & (a_i[, a_j] \ldots) \end{array}$$

‡We deliberately use the term signaling channel rather than covert channel. A signaling channel is a means of information flow which is inherent in the data model and will therefore occur in *every* implementation of the model. We are of course only concerned with signaling channels which violate the $\star$-property. A covert channel on the other hand is a property of a specific implementation and not a property of the data model. That is, even if the data model is free of downward signaling channels, a specific implementation may well contain covert channels due to implementation quirks.

§Some researchers maintain that using polyinstantiation for establishing cover stories is a bad idea and should not be permitted. Our algorithms can easily be modified to have this feature. Rejecting polyinstantiation in such cases is not so straightforward as it may seem. There are denial-of-service implications as will be noted in section 3.2.

¶Strictly speaking in all cases we should be saying c-subject rather than c-user. It is however easier to intuitively consider the semantics by visualizing a human being interactively carrying out these operations. The semantics do apply equally well to processes operating on behalf of a user, whether interactive or not.

In this notation the rectangular parenthesis denote optional items and the "..." signifies repetition. If the list of attributes in omitted, it is assumed that all the data attributes in R_c are specified. Moreover, note that only data attributes A_i can be explicitly given values. The classification attributes C_i are all implicitly given the value c.

Let t be the tuple such that $t[A_k] = a_k$ if A_k is included in the attributes list in the insert statement, $t[A_k] =$ null if A_k is not in the list, and $t[C_l] = c$ for $1 \leq l \leq n$. The insertion is permitted if and only if:

1. $t[A_1]$ does not contain any nulls.
2. For all $u \in R_c : u[A_1, C_1] \neq t[A_1, C_1]$.

If so, the tuple t is inserted into R_c and by side effect into all $R_{c'>c}$. This is moreover the only side effect visible in any $R_{c'>c}$.

Thus, the insert statement works in a straightforward manner. A c-user can insert a tuple t in R_c if R_c does not already have a tuple with the same apparent primary key value and key class as t. In the inserted tuple, the access classes of all data attributes as well as the tuple class are set to c.

3.2 The UPDATE statement

Our interpretation of the semantics of the UPDATE command is close to the one in the standard relational model: UPDATE command is used to change values in tuples that are already present in a relation. It is a set operator; i.e., all tuples in the relation which satisfy the predicate in the update statement are to be updated (provided the resulting relation satisfies polyinstantiation integrity). Since we are dealing with multilevel relations, we may have to polyinstantiate some tuples; however, addition of tuples due to polyinstantiation is to be minimized to the extent possible.

The UPDATE statement executed by a c-user has the following general form, where c is implicitly determined to be the user's login class.

$$\begin{array}{ll} \text{UPDATE} & R_c \\ \text{SET} & A_i = s_i[, A_j = s_j] \ldots \\ \text{[WHERE} & p] \end{array}$$

Here, s_k is a scalar expression, and p is a predicate expression which identifies those tuples in R_c that are to be modified. The predicate p may include conditions involving the classification attributes, in addition to the usual case of data attributes. The assignments in the SET clause, however, can only involve the data attributes. The corresponding classification attributes are implicitly determined to be c.

The intent of the UPDATE operation is to modify $t[A_k]$ to s_k in those tuples t in R_c that satisfy the given predicate p. In multilevel relations, however, we have to implement the intent slightly differently in order to prevent illegal information flows. In particular if $t[C_k] < c$ and $t[A_k] \neq$ null the $\star$-property prevents us from actually updating $t[A_k]$ in place, since this would amount to a write down. We must instead keep both values of A_k.[‖] This is achieved by creating a new tuple t' in R_c which is identical to t except for such attributes A_k in the UPDATE statement. As discussed earlier the effect of the update must also be propagated up to $R_{c'>c}$ in a consistent manner. We now make these statements precise.

3.2.1 Effect of an UPDATE at the User's Access Class

First consider the effect of an update operation by a c-user on R_c. Let

$$S = \{t \in R_c : t \text{ satisfies the predicate } p\}$$

We describe the effect of the UPDATE operation by considering each tuple $t \in S$ in turn. The net effect is obtained as the cumulative effect of updating each tuple in turn. The UPDATE operation will succeed if and only if at every step in this process polyinstantiation integrity is maintained. Otherwise the entire UPDATE operation is rejected and no tuples are changed. In other words UPDATE has an all-or-nothing integrity failure semantics.

It remains to consider the effect of UPDATE on each tuple $t \in S$. There are two components to this effect. Firstly, tuple t is replaced by tuple t' which is identical to t except for those data attributes which are assigned new values in the SET clause. This is the familiar replacement semantics of UPDATE in single-level relations. The formal definition of the tuple t' obtained by replacement semantics is straightforward as follows.

$$t'[A_k, C_k] = \begin{cases} t[A_k, C_k] & A_k \notin \text{SET clause} \\ <s_k, c> & A_k \in \text{SET clause} \end{cases}$$

Secondly to avoid signaling channels, we may need to introduce an additional tuple t'' to hide the effects of the replacement of t by t' from users at levels below c (c is the level of the user executing the UPDATE). This will occur whenever there is some attribute A_k in the SET clause with $t[C_k] < c$. The tuple t'' is defined as follows.

$$t''[A_k, C_k] = \begin{cases} t[A_k, C_k] & t[C_k] < c \\ <\text{null}, t[C_1]> & t[C_k] = c \end{cases}$$

Thus, each tuple $t \in S$ is replaced by t' and possibly in addition by t'' (if t'' exists and survives subsumption). The update is successful if the resulting relation satisfies polyinstantiation integrity. Otherwise the update is rejected and the original relation is left unchanged. The readers should refer to [16] for additional explanation.

[‖]Another alternative in such cases is to reject the UPDATE. This amounts to denial-of-service to the c-user attempting this UPDATE due to the actions of some user below c. This denial-of-service may be acceptable in some situations, but it is not a real general-purpose solution to the problem.

3.2.2 Effect of an UPDATE Above the User's Access Class

Next consider the effect of the update operation on $R_{c'>c}$. This of course assumes that the update operation on R_c was successful. Unfortunately, the core integrity properties do not uniquely determine how an update by a c-user to R_c should be reflected in updates to $R_{c'>c}$. Several different options have been proposed [10, 16, 17, 18]. In this paper we will adopt the *minimal propagation rule* [16] which introduces exactly those tuples in $R_{c'>c}$ that are needed to preserve inter-instance property, i.e., put t' and t'' (if t'' exists and survives subsumption) in each $R_{c'>c}$ and nothing else. Alternate semantics are discussed in section 6.

Formally, the effect of the update operation is again best explained by focusing on a particular tuple t in S. Let A_k be an attribute in the SET clause such that: (i) $t[C_k] = c$ and (ii) $t[A_k] = x$ where x is non-null. That is the c-user is actually changing a non-null value of $t[A_k]$ at his own level to s_k. Now consider $R_{c'>c}$. Due to polyinstantiation there may be several tuples u in $R_{c'>c}$ which have the same apparent primary key as t (i.e., $u[A_1, C_1] = t[A_1, C_1]$) and match t in the A_k and C_k attributes (i.e., $u[A_k, C_k] = t[A_k, C_k]$). To maintain polyinstantiation integrity we must therefore change the value of $u[A_k]$ from x to s_k. This requirement is formally stated as follows.

1. For every $A_k \in$ SET clause with $t[A_k] \neq$ null define U to be the set

   $$\{u \in R_{c'>c} : \begin{array}{l} u[A_1, C_1] = t[A_1, C_1] \wedge \\ u[A_k, C_k] = t[A_k, C_k] \end{array} \}$$

   Polyinstantiation integrity dictates that we replace every $u \in U$ by u' identical to u except for

   $$u'[A_k, C_k] = < s_k, c >$$

   This rule applies cumulatively for different A_k's in the SET clause.

This requirement is an absolute one and must be rigidly enforced by the DBMS. The next requirement is imposed by the inter-instance integrity property of section 2.

2. To maintain inter-instance integrity we insert t' and t'' (if it exists and survives subsumption) in $R_{c'>c}$.

This second requirement is a weaker one than the first, in that inter-instance integrity only stipulates what minimum action is required. We can insert a number of additional tuples v in $R_{c'>c}$ with $v[A_1, C_1] = t'[A_1, C_1]$ so long as the core integrity properties are not violated. In particular if t' subsumes the tuple in $\sigma(\{v\}, c)$ inter-instance integrity is still maintained. Minimal propagation makes the simplest assumption in this case, i.e., only t' and t'' are inserted in $R_{c'>c}$ and nothing else is done.

3.3 The DELETE statement

The DELETE statement has the following general form:

DELETE
FROM R_c
[WHERE p]

Here, p is a predicate expression which helps identify those tuples in R_c that are to be deleted. The intent of the DELETE operation is to delete those tuples t in R_c that satisfy the given predicate. But in view of the $\star$-property only those tuples t that additionally satisfy $t[TC] = c$ are deleted from R_c. In order to maintain inter-instance integrity polyinstantiated tuples are also deleted from $R_{c'>c}$.

In particular, if $t[C_1] = c$, then any polyinstantiated tuples in $R_{c'>c}$ will be deleted from $R_{c'>c}$, and so the entity that t represents will completely disappear from the multilevel relation. On the other hand with $t[C_1] < c$ the entity will continue to exist in $R_{t[C_1]}$ and in $R_{c'>t[C_1]}$.

4 DECOMPOSITION AND RECOVERY

In this section, we give our decomposition and recovery algorithms which have been formulated in terms of update operations defined in the previous section. Again due to space constraints we have chosen to give an abstract description and complete formal statement first and defer consideration of examples to section 5.

4.1 DECOMPOSITION

Our decomposition has for each multilevel relation scheme

$$R(A_1, C_1, \ldots, A_n, C_n, TC)$$

a collection of single-level base relations

$$D_c(A_1, C_1, ..., A_n, C_n)$$

one for each access class c in the security class lattice. This is in contrast to the SeaView decomposition [17] and the Jajodia-Sandhu decomposition [13], both of which require several single-level relations at each access class (compare figures 3 and 5 with figure 6).

A c-user always sees and interacts with the c-instance R_c. Whenever a c-user issues an insert, update, or delete command against R_c, tuples are added, modified, or removed from the underlying base relation D_c. Any change in R_c must be properly reflected in $R_{c'>c}$ (and in $D_{c'>c}$), but this is accomplished during the recovery of a $R_{c'>c}$. Thus, when D_c is modified as the result of an update by a c-user, there are *no* changes made to any other $D_{c'}$, $c' \neq c$. Changes in $R_{c'>c}$ due to updates by c-users are accounted for by our recovery algorithm which uses $\bigcup_{c' \leq c} D_{c'}$ to reconstruct a $R_{c'>c}$.

4.1.1 The INSERT Statement

Suppose as a result of the INSERT statement given in section 3.1 a c-user successfully inserts the following tuple t in R_c: $t[A_k] = a_k$ if A_k is included in the attributes list in the insert statement, $t[A_k] = \text{null}$ if A_k is not in the list, and $t[C_l] = c$ for $1 \leq l \leq n$. In this case our decomposition will also insert the tuple t into D_c.

There are no other insertions. Our recovery algorithm will use $\bigcup_{c' \leq c} D_{c'}$ to reconstruct a $R_{c'>c}$, and since t is in D_c, it will be in $R_{c'>c}$ as well.

4.1.2 The UPDATE statement

We next consider the effect of an update operation by a c-user on R_c. As we have indicated earlier, only D_c will be modified by our decomposition algorithm.

Suppose that a c-user successfully executes the update statement in section 3.2. Once again, let

$$S = \{t \in R_c : t \text{ satisfies the predicate } p\}$$

For each $t \in S$, there are two cases to consider:

1. $t[A_1, C_1] = c$. In this case there can be no polyinstantiation of tuple t at the c level. There is exactly one tuple $u \in D_c$ with $u[A_1, C_1] = t[A_1, C_1]$. We replace u by the following tuple u': $u'[A_1, C_1] = u[A_1, C_1]$ and for $k \neq 1$,

   $$u'[A_k, C_k] = \begin{cases} <s_k, c> & A_k \in \text{SET clause} \\ u[A_k, C_k] & A_k \notin \text{SET clause} \end{cases}$$

   Note that in this case $u'[C_l] = c$ for $1 \leq l \leq n$.

2. $t[A_1, C_1] < c$. In this case tuple t will be polyinstantiated at the c-level. There are two separate subcases depending upon whether or not t has been polyinstantiated at level c prior to the update. These subcases are as follows.

   (a) t is not polyinstantiated at level c prior to the update. In this case there does not exist a tuple $u \in D_c$ with $u[A_1, C_1] = t[A_1, C_1]$. (Note that the tuple class of t must be strictly less than c.)
   We add a tuple u to D_c where u is defined as follows: $u[A_1, C_1] = t[A_1, C_1]$ and for $k \neq 1$,

   $$u[A_k, C_k] = \begin{cases} <s_k, c> & A_k \in \text{SET clause} \\ <?, t[C_k]> & A_k \notin \text{SET clause} \end{cases}$$

   The symbol '?' is a special symbol which can never be an actual value for an attribute. It plays an important role during recovery as we will see in a moment. Informally, a '?' means that this value is to be obtained from the corresponding tuple in $D_{t[C_k]}$.

   (b) t is polyinstantiated at level c prior to the update. In this case there will be one or more tuples $u \in D_c$ which satisfy the condition: $u[A_1, C_1] = t[A_1, C_1]$, and for $k \neq 1$, (i) if $t[C_i] = c$, then $u[A_i, C_i] = t[A_i, C_i]$ and (ii) if $t[C_i] < c$, then $u[A_i, C_i] = <?, t[C_i]>$. For each tuple u which satisfies this condition we replace u by the following tuple u': $u'[A_1, C_1] = u[A_1, C_1]$ and for $k \neq 1$,

   $$u'[A_k, C_k] = \begin{cases} <s_k, c> & A_k \in \text{SET clause} \\ u[A_k, C_k] & A_k \notin \text{SET clause} \end{cases}$$

4.1.3 The DELETE statement

Finally, suppose a c-user executes the DELETE statement given in Section 3.3, and as a result all tuples t that satisfy the predicate p and $t[TC] = c$ are deleted from R_c. In terms of the decomposition, for each such t, we delete from D_c the tuple u which satisfies the following condition: $u[A_1, C_1] = t[A_1, C_1]$, and for $k \neq 1$, (i) if $t[C_i] = c$, then $u[A_i, C_i] = t[A_i, C_i]$ and (ii) if $t[C_i] < c$, then $u[A_i, C_i] = <?, t[C_i]>$.

4.1.4 Summary

To summarize then, whenever a c-user updates the instance R_c, all changes are confined to the underlying base relation D_c. These changes leave ripple marks on $R_{c'>c}$, but this is accomplished when an $R_{c'>c}$ is constructed using our recovery algorithm to be described next.

4.2 RECOVERY ALGORITHM

We are now prepared to give our recovery algorithm. To recover the instance R_c at an access class c, following steps are taken:

1. Form the union $\bigcup_{c' \leq c} D_{c'}$. Extend each tuple t in the result by appending to it its tuple class computed as $t[TC] = \text{lub}\{t[C_i] : i = 1 \ldots n\}$. Call the end result R_c.

2. Next apply the following *Key Deletion Rule* to R_c:

   Let $t_1 \in R_c$ be such that $t_1[C_1] < c$ and R_c does not contain a t_2 such that $t_2[A_1, C_1] = t_1[A_1, C_1]$ and $t_2[TC] = t_1[C_1]$. Then we delete t_1 from R_c. If $t_1[TC] = c$, then we delete t_1 from D_c as well.

   (*Comment.* The motivation for the key deletion rule is that a low user has deleted the tuple key. We therefore delete all higher tuples with that low key as well. Clearly t_1 is no longer needed and its elimination amounts to garbage collection. We could alternately place tuples such as t_1 in a separate relation and have them examined by a suitably cleared subject before physically purging them from the database.)

3. Apply the following ?-*replacement rule* to R_c:

   Let t be a tuple in R_c with $t[A_k]$ = '?'. There are two cases.

   (a) There is a tuple $u \in R_c$ with $u[A_1, C_1] = t[A_1, C_1]$ and $TC[u] = t[C_k]$.
   In this case we replace '?' in $t[A_k]$ by $u[A_k]$.

   (b) There does not exist a tuple $u \in R_c$ with $t[A_1, C_1] = u[A_1, C_1]$ and $TC[u] = t[C_k]$.
   In this case we replace '?' by 'null' in $t[A_k]$.

4. Finally, make R_c subsumption-free by removing all tuples s such that for some $t \in R_c$ and for all $i = 1 \ldots n$ either (i) $t[A_i, s_i] = s[A_i, s_i]$ or (ii) $t[A_i] \neq$ null and $s[A_i]$ = null.

4.3 PROOF OF CORRECTNESS

It is easy to intuitively see why our algorithms are correct. Consider for example the INSERT statement. If a tuple t is inserted in R_c, it is also inserted in D_c. Since our recovery algorithm uses $\bigcup_{c' \leq c} D_{c'}$ to reconstruct R_c, t will be in the reconstructed relation. Similarly, in case of an UPDATE, any changes made to a tuple t in R_c are preserved in the corresponding tuple in D_c. Finally for DELETE, when a tuple t with $t[C_1] = c$ is deleted from R_c, it is also deleted from $R_{c'>c}$. Although in our decomposition, the corresponding tuple is deleted only from D_c, not from $D_{c'>c}$, when we recover a $R_{c'>c}$, the corresponding tuples in $D_{c'>c}$ are deleted using the key deletion rule. These arguments can be formalized; a complete proof is omitted due to lack of space.

It is also easy to see that our algorithms are free of downward signaling channels because all actions a c-user will only directly impact D_c, will never impact $D_{c'<c}$ or $D_{c' \sim c}$ and will impact $D_{c'>c}$ only during the key deletion phase of recovery. Furthermore any R_c is recovered solely from $\bigcup_{c' \leq c} D_{c'}$.

5 EXAMPLES

In this section, we give several examples to illustrate the update semantics as well as our decomposition and recovery algorithms.

5.1 The INSERT statement

To illustrate how the INSERT statement works, consider SOD_U and D_U as shown in figure 7. Suppose a U-user wishes to insert a second tuple to SOD_U. He does so by executing the following insert statement.

```
INSERT
INTO      SOD
VALUES    ('Voy', 'Exp', 'Mars')
```

As a result of the above insert statement, SOD_U and D_U will change to the relations shown in figure 8. If we wish to recover SOD_U, after step 1 of the recovery algorithm SOD_U is identical to D_U of figure 8. Since steps 2, 3, and 4 of the recovery algorithm make no changes to SOD_U, we have the desired result.

5.2 The UPDATE statement

To illustrate the effect of an UPDATE statement, consider the instance SOD_U and the corresponding base relation D_U given in figure 9. Let the instance SOD_S be identical to SOD_U, in which case D_S is empty, as shown in figures 10. Suppose an S-user makes the following update to SOD_S.

```
UPDATE    SOD
SET       DEST = 'Rigel'
WHERE     SHIP = 'Ent'
```

Using our update semantics then SOD_S will have one tuple, as shown in figure 11, and by step 1 of our decomposition algorithm, D_S, which was empty prior to this update, will have a single tuple, call it u, as shown in figure 11. Notice that u contains the pair <?,U> which indicates that during the recovery, '?' is to be be replaced by the attribute value in the corresponding U-tuple. Specifically, let us use the recovery algorithm to reconstruct SOD_S. The first step of the algorithm forms the union of relations D_U and D_S in figures 9 and 11. Since the key deletion rule does not apply, we move to step 3 (?-replacement rule) of the recovery algorithm, which will replace <?,U> in u by <Exp, U> (i.e., the corresponding attribute values for 'Ent' in the lower level relation D_U in figure 9). After the union is made subsumption-free (step 4), we end up with the instance SOD_S in figure 11, as desired.

Next, suppose an U-user executes the following command against SOD_U shown in figure 9:

```
UPDATE    SOD
SET       DEST = 'Talos'
WHERE     SHIP = 'Ent'
```

As a result of this update, our decomposition algorithm only modifies D_U from the instance in figure 9 to the one in figure 12. Readers should verify that if we use our recovery to obtain SOD_S we obtain the instance given in 13, although no changes were made to the underlying D_S as a result of the above update. Of course, SOD_U will change to the relation shown in figure 12.

Finally, suppose starting with the instance SOD_S shown in figure 13 a S-user invokes the following update.

```
UPDATE    SOD
SET       OBJ = 'Spy'
WHERE     SHIP = 'Ent' AND
          DEST = 'Rigel'
```

Using our update semantics, the SOD_S will change to the instance given in figure 14, *not* to the instance given below.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Exp	U	Rigel	S	S
Ent	U	Spy	S	Rigel	S	S

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U

SHIP		OBJ		DEST	
Ent	U	Exp	U	Talos	U

Figure 7: $\mathrm{SOD_U}$ and $\mathrm{D_U}$

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Voy	U	Exp	U	Mars	U	U

SHIP		OBJ		DEST	
Ent	U	Exp	U	Talos	U
Voy	U	Exp	U	Mars	U

Figure 8: $\mathrm{SOD_U}$ and $\mathrm{D_U}$ after INSERT

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	null	U	U

SHIP		OBJ		DEST	
Ent	U	Exp	U	null	U

Figure 9: $\mathrm{SOD_U}$ and $\mathrm{D_U}$

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	null	U	U

SHIP	OBJ	DEST

Figure 10: $\mathrm{SOD_S}$ and $\mathrm{D_S}$

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Rigel	S	S

SHIP		OBJ		DEST	
Ent	U	?	U	Rigel	S

Figure 11: $\mathrm{SOD_S}$ and $\mathrm{D_S}$ after UPDATE by S-User

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U

SHIP		OBJ		DEST	
Ent	U	Exp	U	Talos	U

Figure 12: $\mathrm{SOD_U}$ and $\mathrm{D_U}$ after UPDATE by U-User

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Exp	U	Rigel	S	S

SHIP		OBJ		DEST	
Ent	U	?	U	Rigel	S

Figure 13: $\mathrm{SOD_S}$ and $\mathrm{D_S}$ after UPDATE by U-User

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Spy	S	Rigel	S	S

SHIP		OBJ		DEST	
Ent	U	Spy	S	Rigel	S

Figure 14: $\mathrm{SOD_S}$ and $\mathrm{D_S}$ after UPDATE by S-User

This follows from our underlying philosophy: we need to polyinstantiate to either close a signaling channel or provide a cover story. In terms of the decomposition $\mathrm{D_S}$ will change from the instance in figure 13, to the one in figure 14. We leave it to the reader to verify that our recovery algorithm operates correctly.

5.3 The DELETE statement

To illustrate how the DELETE statement works, suppose a U-user executes the following DELETE statement against the relation $\mathrm{SOD_U}$ shown in figure 12. (Assume S-users see the instance given in figure 13, $\mathrm{D_U}$ is as in figure 12 and $\mathrm{D_S}$ is as in figure 13.)

```
DELETE
FROM      SOD
WHERE     SHIP = 'Ent'
```

Following our DELETE semantics not only will $\mathrm{SOD_U}$ become empty, $\mathrm{SOD_S}$ will become empty as will. As a consequence of the above DELETE, our decomposition algorithm will make $\mathrm{D_U}$ in figure 12 empty. Reader should verify that although $\mathrm{D_S}$ (shown in figure 13) does not change, if we were to recover $\mathrm{SOD_S}$ at this point, the key deletion rule in the recovery algorithm will delete the tuple for the starship 'Ent.'

6 OPTIONS AND EXTENSIONS

As we indicated earlier, the core integrity properties do not uniquely determine how an update by a c-user to R_c should be propagated to $R_{c'>c}$, and several different options have been proposed. This section discusses the relationship between our algorithms and these options.

Our algorithms can accommodate the SeaView MVD requirement [3, 4, 17] most easily. No changes are required in the decomposition algorithm; only recovery algorithm needs to be modified. Steps 1 and 2 of the recovery algorithm remain the same as before. Steps 3 and 4 are changed as follows:

3$'$. For each i, $2 \leq i \leq n$, repeat the following:

Whenever t_1 and t_2 are two tuples in $\mathrm{R_c}$ such that $t_1[A_1, C_1] = t_2[A_1, C_1]$, we add to $\mathrm{R_c}$ tuples t_3 and t_4 defined as follows:

$$\begin{aligned} t_3[A_1, C_1] &= t_1[A_1, C_1] \\ t_3[A_i, C_i] &= t_1[A_i, C_i] \\ t_3[A_j, C_j] &= t_2[A_j, C_j],\ 1 < j \leq n,\ j \neq i \\ t_4[A_1, C_1] &= t_1[A_1, C_1] \\ t_4[A_i, C_i] &= t_2[A_i, C_i] \\ t_4[A_j, C_j] &= t_1[A_j, C_j],\ 1 < j \leq n,\ j \neq i \end{aligned}$$

4$'$. Delete from R_c any tuple that has a '?' as a value.

5$'$. Same as step 4 of the original algorithm.

Our decomposition as well as recovery algorithms will have to be modified to accommodate the single tuple per tuple class approach of [19] or the closely related single maintenance level attribute approach adopted by the LDV model [10, 21]. These modifications are straightforward.

This brings us to the dynamic mvd requirement proposed in [18]. It too will require modifications to both our decomposition and recovery algorithms along the lines discussed in [15]. The major difference is that in the single-level relations D_c we will sometimes require '?' for classification attributes (rather than just for data attributes as in section 3). Details on the exact modifications are omitted due to lack of space.

It is also possible to have a single decomposition algorithm for updates and realize the several alternate semantics discussed above (and others from the literature) by varying only the recovery algorithm. Again due to limitation of space we are unable to elaborate on this idea here.

7 SEMANTICS OF POLYINSTANTIATION

A power of a data model is determined by its ability to accurately represent requirements of different users. It is important, therefore, that a data model be flexible and general enough so that it can support the needs of all applications over a long period of time (due to longevity of databases). Constraints are also important in a data model. They provide necessary restrictions on database instances: those that satisfy the constraints are allowable, while those that do not are not allowable. They are needed for integrity reasons in traditional databases, and additionally, for security reasons in multilevel secure databases.

In a data model, there are two basic types of constraints: *inherent* constraints and *explicit* constraints (see, for example, [22]). The former type of constraint is an integral part of the structure of the data model. For example, in the hierarchical model all relationships must be in the form of a tree; in the relational model all data must be stored in a tabular form. Another inherent constraint in the relational model is that duplicate tuples are not permitted in a relation (which translates into the requirement that each relation must have a key).

Unlike inherent constraints which are quite rigid, the explicit constraints provide a flexible mechanism for augmenting the set of inherent constraints with additional constraints that further restrict the set of allowable database states. A big advantage of the relational model over other data models (hierarchical, network, and others) is that it has very few inherent constraints, viz., entity integrity and referential integrity. All other constraints are in the form of explicit constraints which allows a great deal of freedom in terms of representing requirements of different applications.

It is our view that we should try our best to preserve this advantage when we move to the multilevel secure relational model. There is a need to keep inherent constraints to a minimum, while allowing for a wider variety of explicit constraints which can be added on a relation to relation basis. With the functional dependency condition as the only inherent requirement for polyinstantiation integrity (i.e., property 4 of section 2), a user has the maximum possible flexibility of being able to represent as many instances as possible. If we need to further restrict these instances, we can always do so by imposing additional explicit constraints on them.

In this respect we are particularly troubled by some of the proposals being made for update semantics in prototypes of multilevel relational DBMS's. While we believe these semantics are acceptable as options in certain specific situations, we object to efforts to build these semantics into the data model and impose them on each and every application.

As a concrete illustration of our concern consider the dynamic mvd PI property of [18] which has been proposed as a replacement for the original mvd PI property of SeaView [3, 4, 17] in response to criticism of the latter in [15]. To understand the implications of dynamic mvd consider the relation SOD given in figure 15 where it is shown that initially U, C, and S users see the same data. Now suppose a S-user executes the following update.

```
UPDATE   SOD
SET      DEST = 'Rigel'
WHERE    SHIP = 'Ent'
```

As per our earlier discussion SOD_S changes to the instance shown in figure 16 while SOD_U and SOD_C remain unchanged as in figure 15.

Next suppose a C-user executes the following update.

```
UPDATE   SOD
SET      OBJ = 'Spy'
WHERE    SHIP = 'Ent'
```

Now SOD_C will change from figure 15 to figure 17.

The question arises as to how this change should propagate to SOD_S. With the minimal propagation rule of section 3 the effect on SOD_S is as shown in figure 18. With the dynamic mvd rule of [18] the effect on SOD_S is as shown in figure 19. The difference in this example lies in the bottommost tuple of figure 19 which is not present in figure 18. In more elaborate examples these extra tuples multiply rapidly.

The justification given by [18] for the dynamic mvd amounts to the argument that if the two update statements above were executed in the opposite order, i.e., the C-user went first followed by the S-user, then SOD_S would go from figure 15 to figure 17 and then to figure 19. The dynamic mvd is therefore introduced to keep the semantics of update order independent. (It should be noted that for this opposite order of updates the update semantics of section 3 will also take SOD_S from figure 15 to figure 17 and then to figure 19.)

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U

Figure 15: Initial $SOD_U = SOD_C = SOD_S$

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Exp	U	Rigel	S	S

Figure 16: Updated SOD_S

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Spy	C	Talos	U	C

Figure 17: Updated SOD_C

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Spy	C	Talos	U	C
Ent	U	Exp	U	Rigel	S	S

Figure 18: Updated SOD_S by minimal propagation

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Spy	C	Talos	U	C
Ent	U	Exp	U	Rigel	S	S
Ent	U	Spy	C	Rigel	S	S

Figure 19: Updated SOD_S by dynamic mvd

We find the justification of order independence for dynamic mvd to be quite unacceptable in general, because there are many situations where order dependence is important. In a specific situation one might argue that the order independence of dynamic mvd's is the required semantics. However, this is properly an issue of application semantics to be determined by the Database Administrator and not something that one embeds as a fundamental property of a data model.

In fact the most common situation in database systems is that order of updates is extremely significant. Two familiar examples are given below.

Example 1. Consider that one transaction gives a 5% raise to all employees. This is followed by a second transaction which enrolls a new employee with a specified salary. It is unlikely that any organization would want the new employee to get a 5% raise. However if the semantics of update are order independent this will happen. After all if the two transactions executed in the opposite sequence the new employee would get a raise.

Example 2. Consider a debit transaction followed by a credit transaction on an account. Suppose the debit transaction is rejected due to insufficient balance. Also suppose the subsequent credit transaction raises the account balance sufficiently so that the previous debit transaction would have succeeded. That is if the transactions had executed in the opposite sequence both would succeed. Clearly no organization can afford to have order-independent update semantics in such cases.

There are any number of such examples which can be described and their extension to multilevel relations is straightforward.

8 CONCLUSION

In this paper we have described a new decomposition algorithm that breaks a multilevel relation into single-level relations and a new recovery algorithm which reconstructs the original multilevel relation from the decomposed single-level relations. We have demonstrated that there are several novel aspects to our decomposition and recovery algorithms which provide substantial advantages over previous proposals. We have also made the case that some of the alternate update semantics which have been proposed—notably the dynamic mvd of [18]—should be available as options but should certainly not be made an integral part of the data model.

Several researchers [1, 5, 7, 8, 12] have expressed concern about the expected performance of database systems based on kernelized architecture. We believe that the new decomposition algorithm can be effectively used to alleviate many of these concerns. As part of our future work, we will develop a stochastic model that is powerful enough to help us measure the actual performance improvements.

Acknowledgment

We are indebted to John Campbell, Joe Giordano and Howard Stainer for their support and encouragement, making this work possible. The opinions expressed in this paper are of course our own and should not be taken to represent the views of these individuals.

References

[1] "Multilevel Data Management Security," Committee on Multilevel Data Management Security, Air Force Studies Board, National Research Council, Washington, DC (1983).

[2] Date, C.J. *An Introduction to Database Systems.* Volume II, Addison-Wesley, (1983).

[3] Denning, D.E., Lunt, T.F., Schell, R.R., Heckman, M., and Shockley, W.R. "A Multilevel Relational Data Model." *Proc. IEEE Symposium on Security and Privacy*, 220-234 (1987).

[4] Denning, D.E., Lunt, T.F., Schell, R.R., Shockley, W.R. and Heckman, M. "The SeaView Security Model." *Proc. IEEE Symposium on Security and Privacy*, 218-233 (1988).

[5] Froscher, J. N. and Meadows, C., "Achieving a Trusted Database Management System Using Parallelism," *Database Security, II: Status and Prospects,* Landwehr, C.E. (editor), North-Holland, pages 151-160 (1989).

[6] Garvey, C. "Multilevel Data Storage Design." TRW Defense Systems Group (1986).

[7] Garvey, C., Hinke, T., Jensen, N., Solomon, J., and Wu, A., "A Layered TCB Implementation Versus the Hinke-Schaefer Approach," *Database Security, III: Status and Prospects,* Spooner, D. L. and Landwehr, C. (editors), North-Holland, pages 151-165 (1990).

[8] Graubart, R., "A Comparison of Three Secure DBMS Architectures," *Database Security, III: Status and Prospects,* Spooner, D. L. and Landwehr, C. (editors), North-Holland, pages 167-190 (1990).

[9] Grohn, M.J. "A Model of a Protected Data Management System." Technical Report ESD-TR-76-289, I.P. Sharp Associates Ltd., (1976).

[10] Haigh, J. T., O'Brien, R. C., and Thomsen, D. J. "The LDV Secure Relational DBMS Model." *Database Security IV: Status and Prospects*, Jajodia, S. and Landwehr, C. (editors), North-Holland, to appear.

[11] Hinke T.H. and Schaefer M. "Secure Data Management System." Technical Report RADC-TR-75-266, System Development Corporation (1975).

[12] Jajodia, S. and Kogan, B. "Transaction Processing in Multilevel-Secure Databases Using Replicated Architecture," *IEEE Symp. on Research in Security and Privacy,* Oakland, Calif., May 7-9, pages 360-368 (1990).

[13] Jajodia, S. and Sandhu, R.S. "Polyinstantiation Integrity in Multilevel Relations." *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, May 1990, pages 104-115.

[14] Jajodia, S. and Sandhu, R.S. "A Formal Framework for Single Level Decomposition of Multilevel Relations." *Proc. IEEE Workshop on Computer Security Foundations*, Franconia, New Hampshire, June 1990, pages 152-158.

[15] Jajodia, S. and Sandhu, R.S. "Polyinstantiation Integrity in Multilevel Relations Revisited." *Database Security IV: Status and Prospects*, Jajodia, S. and Landwehr, C. (editors), North-Holland, to appear.

[16] Jajodia, S. , Sandhu, R.S., and Sibley E., "Update Semantics of Multilevel Relations." *Proc. 6th Annual Computer Security Applications Conf.*, Tucson, AZ, December 1990, pages 103-112.

[17] Lunt, T.F., Denning, D.E., Schell, R.R., Heckman, M. and Shockley, W.R. "The SeaView Security Model." *IEEE Transactions on Software Engineering*, 16(6):593-607 (1990).

[18] Lunt, T.F. and Hsieh, D. "Update Semantics for a Multilevel Relational Database." *Database Security IV: Status and Prospects*, Jajodia, S. and Landwehr, C. (editors), North-Holland, to appear.

[19] Sandhu, R.S., Jajodia, S. and Lunt, T. "A New Polyinstantiation Integrity Constraint for Multilevel Relations." *Proc. IEEE Workshop on Computer Security Foundations*, Franconia, New Hampshire, June 1990, pages 159-165.

[20] Schkolnick, M. and Sorenson, P. "The Effects of Denormalization on Database Performance." *The Australian Computer Journal*, 14(1):12-18 (February 1982).

[21] Stachour, P. D. and Thuraisingham B. "Design of LDV: A Multilevel Secure Relational Database Management System." *IEEE Trans. on Knowledge and Data Engineering*, 2(2):190-209 (June 1990).

[22] Tsichritzis, D. C. and Lochovsky, F. H. *Data Models.* Prentice-Hall, (1982).