Enforcement of Integrity Constraints against Transactions with Transition Axioms

Sang Ho Lee*, Lawrence J. Henschen+, Myoung Ho Kim‡, Yoon-Joon Lee‡

*Database Section, Electronics and Telecommunications Research Institute, Korea
+Dept. of Electrical Eng. and Computer Science, Northwestern University, Illinois, U.S.A.
‡Dept. of CS and Center for AI Research, Korea Adv. Inst. of Science and Tech., Korea

Abstract

This paper addresses an enforcement technique of integrity constraints against transaction updates in a relational database system. Transition axioms have been used effectively in checking integrity constraints for single update statements. In this paper we extend the idea of transition axioms to a transaction which is a sequence of read and update statements. Integrity constraints in our scheme are simplified without database access before the actual operations are performed, avoiding the need to undo an illegal transaction. We also propose transaction partitioning which can reduce the overhead of checking integrity constraints significantly. Partitioning a transaction becomes crucial when a transaction is associated with multiple integrity constraints.

1 Introduction

An Integrity Constraint (IC) in a database is a predicate (assertion) on database states which represents time-invariant semantics of data and serves as the validity criteria of the database. When a database state is changed due to database manipulation operations, the database should continue to satisfy ICs after the update operations are completed.

A transaction in database systems is a sequence of operations including database operations or schema definitions or manipulation operations, that is atomic with respect to recovery. The notion of transaction has been around quite for a long time in the database community and has been introduced in terms of concurrency control and recovery which are important functionalities of database systems. A (typical) transaction must have four well-known ACID properties: atomicity, consistency, isolation and durability [6]. The consistency property of a transaction states that each successful transaction commits only legal results by definition so that it preserves the consistency of the database. In other words, the intermediate database states during transaction execution may be inconsistent but the final database state after the transaction commit time must be consistent with respect to ICs. When there are some ICs associated with a transaction, we need to enforce the ICs explicitly to keep the database state consistent at the commit time.

Many researchers [1,2,4,8,11,12,16] have studied on how to maintain a consistency of a database. Evaluation of ICs in their original form is known to be time-consuming. This evaluation, however, can be reduced by taking advantage of the fact that the database is consistent with ICs before updates, so that a simplified IC is true if and only if the original IC is true in database after update operations are completed.

However, most of those researches have focused on *simple* updates (the update is called to be simple when it can be expressed by single data manipulation statement) and have not explicitly provided the ways of how to apply their methods on complex updates which often occur in many typical transactions. Even though there have been a few IC enforcement techniques on complex updates in the literature [7,13], their methods have several limitations. First, the CHANGE operation in [7,13] is implemented by a DELETE followed by an INSERT, which may not be correct in some cases. For example, when there is no tuples affected by the CHANGE operation, the approach of a DELETE followed by an INSERT will produce spurious tuples. Second, the transaction in Qian's work [13] is not allowed to have two updates on the same relation, which is too severe restriction in practice. Third, Hue and Imielinski's method [7] makes use of only the matrix of ICs in Prenex Normal Form to simplify ICs, which turns out a preliminary simplification of ICs. Their

0730-3157/92 $3.00 © 1992 IEEE

simplification of ICs can be improved when informations about updates and ICs are available.

This paper addresses an enforcement technique of integrity constraints against a transaction. Our method, which is logic-based, checks ICs before actual updates are performed, and handles a transaction which can be complex updates. Our scheme is fully compatible with the notion of compilation in which the simplified ICs are stored and invoked later whenever appropriate (for example, upon user's request, on actual update operation time, etc). Our starting point is with McCune and Henschen's method [10] in which the simple update is of main concern. In this paper, we extend their method toward the transaction concept.

We assume the reader is familiar with the relational database, its connection with logic [5] and resolution-style automated reasoning [3].

2 Preliminaries

Before we present the idea of this work, we briefly describe Henschen and McCune's work [10] on which this paper is based. They present a technique which maintains the integrity of a relational database in a proof theoretic view. Their method takes an IC and an update expressed in first-order logic, and either proves that the update cannot violate IC or generates a formula IC' (a complete test) that is satisfied before the update if and only if IC would continue to be satisfied after the update. IC' is frequently much easier to evaluate than IC. IC' can also be compiled at database creation time and invoked when updates are attempted.

For space limitation, we only describe some materials [10] relevant to our presentation. To give a correct characterization of the relationship of the database states before and after the update, two new predicate symbols, $ROLD$ and $RNEW$, are introduced. $ROLD$ (resp., $RNEW$) represents the state of relation R before (resp., after) the update. With the new predicates, relationship between the old and new state of the database for several kinds of updates can be formulated precisely. We call such expresions transition axioms (TAX). The importance and usage of transition axioms are given in [10]. For INSERT $R(\vec{a})$, the transition axiom is $((\forall \vec{x})\ RNEW(\vec{x}) \leftrightarrow (ROLD(\vec{x}) \vee \vec{x} = \vec{a}))$. Let C be a constraint. $C[ROLD]$ (resp. $C[RNEW]$) denotes C with all occurrences of R replaced by $ROLD$ (resp. $RNEW$). $C[ROLD] \wedge TAX \rightarrow C[NEW]$ is the theorem on which the method is based on. TAX is of the form $(\forall \vec{x})\ (RNEW(\vec{x}) \leftrightarrow W(\vec{x}))$. Let $\neg C[TAX]$ be $\neg C[RNEW]$ with all atoms over $RNEW$ replaced with the corresponding instances of $W(\vec{x})$. Then $C[ROLD] \wedge TAX \rightarrow C[NEW]$ is a theorem if and only if $C[ROLD] \wedge \neg C[TAX]$ is unsatisfiable. The test clause is obtained from $C[ROLD] \wedge \neg C[TAX]$.

A transaction normally contains a sequence of database access statements. The individual statements in a transaction are executed sequentially one after the other. In terms of IC enforcements, statements which cause the transition of database states (i.e. insert, delete and update operations) are only of concern. Operations such as data retrieval is out of concern as far as database consistency goes. Even though the definition of a single data manipulation statement is varied on a particular database language, the typical forms of it can be categorized as follows [10].

- INSERT $R(\vec{a})$
- DELETE $R(\vec{a}, \vec{y})$
- CHANGE $R(\vec{a}, \vec{b}, \vec{z}, \vec{w})$ to $R(\vec{a}, \vec{c}, \vec{d}, \vec{w})$

Note that a single update in our representation may correspond to a set of single updates in other representations. For convenience, each single statement in a transaction is numbered for easy reference and each transaction is terminated with semicolon (;). $a, b, \ldots$ denote regular constants and $\vec{a}, \vec{b}, \ldots$ denote vectors of constants.

In what follows, we present the notion of order independent transaction which renders a transaction amenable to our scheme. The update statements in a transaction are executed sequentially in most cases. The effect of a single statement in a transaction potentially may be changed by another single statement in the same transaction, which implies that the sequential execution sometimes does some redundant work. For example, suppose we have a transaction T_1 and T_1'.

T_1 1. INSERT R(a,b)
2. CHANGE R(x,c) to R(x,d)
3. DELETE R(a,b);

T_1' 1. CHANGE R(x,c) to R(x,d);

A simple inspection shows that the transaction T_1 is equivalent to T_1' where a transaction is defined to be equivalent to another transaction if they produce the same database states. The above phenomenon arises when there are at least two single updates which conflict with each other. Here two update operations are said to conflict if they operate on the same data item.

Definition 2.1 A transaction T is *order dependent* if

and only if T contains at least two conflicting update operations. Otherwise T is *order independent.*

Example 2.1 (of Definition 2.1)

T_1 1. INSERT R(a,b)	T_2 1. INSERT R(a,b)
2. DELETE R(a,b);	2. DELETE R(c,d);

Transaction T_1 is order dependent while T_2 is order independent. □

An order independent transaction has an important advantage of its update statements being executed in parallel without considering their relative execution orders. With an order independent transaction we can consider its single updates in an arbitrary order.

Lemma 2.1 Every order dependent transaction can be transformed into equivalent order independent transactions.

Proof: First we show that any two adjacent conflicting updates in an order dependent transaction can be converted into a set of non-conflicting updates. Because there are three forms of single updates (i.e. INSERT, DELETE and CHANGE), there are nine possible combinations of them. We consider each of them separately.

Case 1. INSERT $R(\vec{a})$ is followed by DELETE $R(\vec{b},\vec{x})$. Let R_i and R_d be subsets of relation R referenced by $R(\vec{a})$ and $R(\vec{b},\vec{x})$, respectively. There are three cases from the relationship between R_i and R_d.

(a) $R_i = R_d$ (i.e. the two sets are identical): The pure effect of the two operations is nothing.

(b) $R_i \supset R_d$ (i.e. R_d is a proper subset of R_i): This case is impossible to occur.

(c) $R_i \subset R_d$ (i.e. R_i is a proper subset of R_d): The pure effect of the two operations is the same as DELETE $R(\vec{b},\vec{x})$.

Case 2. DELETE $R(\vec{b},\vec{x})$ is followed by INSERT $R(\vec{a})$. Let R_i and R_d be subsets of relation R referenced by $R(\vec{a})$ and $R(\vec{b},\vec{x})$, respectively. Similarly, there are three cases.

(a) $R_i = R_d$: The pure effect of the two operations is nothing.

(b) $R_i \supset R_d$: This case is impossible.

(c) $R_i \subset R_d$: The pure effect of the two operations is the same as INSERT $R(\vec{c})$ where $R(\vec{c})$ represents tuples in $(R_d - R_i)$.

Case 3. INSERT $R(\vec{e})$ is followed by CHANGE $R(\vec{a},\vec{b},\vec{z},\vec{w})$ to $R(\vec{a},\vec{c},\vec{d},\vec{w})$.

Let R_i and R_c be subsets of relation R referenced by $R(\vec{e})$ and $R(\vec{a},\vec{b},\vec{z},\vec{w})$, respectively. Again there are three cases.

(a) $R_i = R_c$: The pure effect of the two operations is INSERT $R(\vec{f})$ where the inserted tuple $R(\vec{e})$ is updated into $R(\vec{f})$ by the CHANGE operation.

(b) $R_i \supset R_c$: This case is impossible.

(c) $R_i \subset R_c$: The pure effect of the two operations is the same as CHANGE $R(\vec{a},\vec{b},\vec{z},\vec{w})$ to $R(\vec{a},\vec{c},\vec{d},\vec{w})$ and INSERT $R(\vec{f})$ where the inserted tuple $R(\vec{e})$ is updated into $R(\vec{f})$.

For the other six cases, the similar arguments are applied.

The termination condition is when there are no more conflicting updates in a transaction, and it is clear that the process eventually stops. This completes the proof. QED

3 Enforcement of ICs against Transactions

3.1 Transition Axioms for Transactions

To extend McCune and Henschen's method for transactions, we need to generate the transition axioms with respect to a transaction. Algorithm 3.1 produces the transition axiom for a transaction. If a transaction attempts to update several relations, Algorithm 3.1 is applied for each relation which is updated by the transaction, i.e. there is one transition axiom for each updated (affected) relation.

Algorithm 3.1 Producing the Transition Axiom

1. /* Part1 denotes a part of the relation either unaffected or deleted by the transaction, and Part2 denotes a part of the relation affected by INSERT or CHANGE */
   /* The final transition axiom is of the form $((\forall\vec{x})\ RNEW(\vec{x}) \Leftrightarrow$ part1 $\vee$ part2) */
   part1 $\longleftarrow$ $ROLD(\vec{x})$;
   part2 $\longleftarrow$ $\emptyset$;
2. /* take care of DELETE operation */
   For each DELETE operation $R(\vec{a},\vec{y})$ do
   part1 $\longleftarrow$ part1 $\wedge$ $\vec{x}_i \neq \vec{a}$;
   /* $\vec{x}_i$ is a corresponding subset of $\vec{x}$ */
3. /* CHANGE operation */
   For each CHANGE operation $R(\vec{a},\vec{b},\vec{z},\vec{w})$ to $R(\vec{a},\vec{c},\vec{d},\vec{w})$ do begin
   part1 $\longleftarrow$ part1 $\wedge$ $(\vec{x}_j,\vec{y}_j) \neq (\vec{a},\vec{b})$
   part2 $\longleftarrow$ part2 $\wedge$ $(\exists\vec{z}_j^*)$ $(ROLD(\vec{a},\vec{b},\vec{z}_j^*,\vec{w})$ $\wedge(\vec{x}_j,\vec{y}_j,\vec{z}_j) = (\vec{a},\vec{c},\vec{d}))$
   end;
   /* $\vec{x}_j,\vec{y}_j,\vec{z}_j$, and $\vec{z}_j^*$ is the corresponding renewed

subset of $\vec{x}$ */

4. /* INSERT operation */
   For each INSERT operation $R(\vec{a})$ do
   part2 $\longleftarrow$ part2 $\vee$ $(\vec{x} = \vec{a})$;
5. The transition axiom is,
   $(\forall \vec{x})$ $(RNEW(\vec{x}) \Leftrightarrow$ part1 $\vee$ part2).

□

Theorem 3.1 Algorithm 3.1 correctly produces the transition axiom for any order independent transaction. [9]

Example 3.1 (of Algorithm 3.1)

T: 1. CHANGE $R(a, y)$ to $R(b, y)$
2. INSERT $R(c, d)$;
At step 3, we have
part1 = $(ROLD(x, y) \wedge x \neq a)$
part2 = $(ROLD(a, y) \wedge x = b)$
At step 4, we get
part1 = $(ROLD(x, y) \wedge x \neq a)$
part2 = $(ROLD(a, y) \wedge x = b) \vee (x, y) = (c, d))$
The transition axiom is,
$(\forall x)(\forall y)$ $(RNEW(x, y) \Leftrightarrow (ROLD(x, y) \wedge x \neq a)$
$\vee((ROLD(a, y) \wedge x = b) \vee (x, y) = (c, d)))$. □

Example 3.2 (of Algorithm 3.1)

T : 1. CHANGE $R(a, y, z)$ to $R(b, y, z)$
2. CHANGE $R(f, c, z)$ to $R(f, d, z)$
3. DELETE $R(e, f, g)$;
Similarly the transition axiom of relation R is,
$(\forall x)(\forall y)(\forall z)$ $(RNEW(x, y, z) \Leftrightarrow (ROLD(x, y, z)$
$\wedge x \neq a \wedge (x, y) \neq (f, c) \wedge (x, y, z) \neq (e, f, g))\vee$
$((R(a, y, z) \wedge x = b) \vee (ROLD(f, c, z)$
$\wedge(x, y) = (f, d))))$. □

3.2 Transaction Partitioning

A transaction is very likely to be associated with more than one IC. Suppose that each of n single updates in a transaction T is related with $IC_1, \ldots, IC_n$, respectively. Checking all $IC_1, \ldots, IC_n$ against T as a whole is the waste of time because each of IC_i $(i = 1, \ldots, n)$ is related only one single update of T. For example, a single update in a transaction T which is related with IC_1 may have nothing to do with IC_2, presuming T is related with IC_1 and IC_2. In order to achieve a significant performance improvement, we partition T in accordance with ICs into several sub-transactions which are treated separately. This is the basic idea behind the simplification of a transaction through partitioning. Furthermore, partitioning transactions into sub-transactions is more amenable to the integrity method of Henschen and McCune [10].

Though it may be desirable to partition a transaction into finest sub-transactions as possible, we here describe a method of partitioning a transaction into two sub-transactions only.

We now introduce the following notations to describe our idea clearly.

Comp-Test(U) A derived complete test [10] to check the validity of an update U.

Eval(F) An evaluation of a closed formula F. The result of Eval(F) always is either true or false.

Perform(U) An actual update operation of an update U.

Definition 3.1 Let T_1 and T_2 be two disjoint sub-transactions of a transaction T such that $T_1 \cup T_2 = \emptyset$. T_1 is *executable prior to* T_2 with respect to IC if and only if executing two sequences, 'Eval(Comp-Test(T_1)), Perform(T_1), Eval(Comp-Test(T_2)), Perform(T_2)' and 'Eval(Comp-Test(T)), Perform(T)' produce the same results. That is, both sequences are either accepted or rejected with respect to IC.

Definition 3.2 T_s is defined to be a sub-transaction of T such that T_s contains all single updates which are known not to violate IC. Define T_c to be $T - T_s$. (Note that $T_c \cap T_s = \emptyset$ and $T_c \cup T_s = T$.)

Definition 3.3 A transaction T is *partitionable* if and only if T can be partitioned into T_s and T_c with T_s non-empty.

There are cases where a single update U to relation R does not violate an IC. These cases are if one of the following conditions holds.

1. R has no occurrence in IC.
2. U is DELETE (INSERT) and R has only negative (positive, respectively) occurrences in IC.
3. If U is DELETE (INSERT), then for each positive (negative, respectively) occurrence of R in IC, either one argument R is a regular constant which differs from the corresponding (regular) constant in the update form or the occurrence of R contains two identical arguments while the corresponding arguments in update form are different or vice versa.

The first and second cases have been shown in [10] and the third case has been described in [11].

Example 3.3 (of Definition 3.3)

IC: $(\forall x)(\forall y)(\forall z)$
$(SUPPLY(x,y,z) \wedge CLASS(z,T4) \Rightarrow x = C1)$
T 1. INSERT $SUPPLY(a,b,c)$
2. INSERT $CLASS(d,T4)$
3. INSERT $CLASS(c,T2)$
4. DELETE $SALE(TOY,e)$;

Single updates 3 and 4 are known never to violate IC by case 3 and case 1, respectively. Thus, T is partitionable into T_c={1,2} and T_s={3,4}. □

Suppose that a transaction T is partitioned into two sub-transactions T_1 and T_2, and T_1 is executable prior to T_2. The sequence, Test for T_1, if acceptable Perform T_1, Test for T_2, if acceptable Perform T_2, should succeed if and only if the original transaction would have succeeded. Note that a sub-transaction may contain only one single update.

Processing a partitionable transaction in its unpartitioned form incurs considerable overhead in simplification of ICs. It may not simplify ICs completely as well [9]. This is because extra disjunctions in clauses become an obstacle in extracting paramodulators [3] in some cases. It is obviously desirable to minimize a number of single updates which are to be processed at a time.

Theorem 3.2 T_s is executable prior to T_c.
Proof: Any single statement in T_s does not generate any test clause because a refutation is derived at the resolution stage (i.e. Comp-Test(T_s) = $\emptyset$). Thus, the database state remains consistent after T_s is performed (i.e. the effect of T_s is then reflected in the database state). Comp-Test(T_c) derived from the original (initial) database state which is assumed to be consistent, is certainly applicable because the database continues to be consistent after T_s is performed. Thus, two sequences 'Eval(Comp-Test(T_s)), Perform(T_s), Eval(Comp-Test(T_c)), Perform(T_c)' and 'Eval(Comp-Test($T_s \cup T_c$)), Perform($T_s \cup T_c$)' produce the same effects. The proof is completed. QED

Theorem 3.2 says that we can delete a set of single updates which belongs to T_s in order to simplify the transaction. T_s can be processed separately prior to T_c with no regard of database consistency.

3.3 Overall Descriptions

T_s does not generate a complete test at all (i.e. Comp-Test(T_s) = $\emptyset$) as explained previously. We often have a more simplified Comp-Test(T_c) if the effect of T_s is reflected in deriving Comp-Test(T_c) [9]. One way to achieve that under Closed World Assumption [14] is to include the effect of T_s into '$C[ROLD] \wedge \neg C[TAX]$'. If a single update in T_s is INSERT $R(\vec{a})$ (DELETE $R(\vec{x},\vec{a})$), put $R(\vec{a})$ ($\neg R(\vec{x},\vec{a})$, respectively) in '$C[ROLD] \wedge \neg C[TAX]$'. If a single update is CHANGE $R(\vec{a},\vec{b},\vec{z},\vec{w})$ to $R(\vec{a},\vec{c},\vec{d},\vec{w})$, then put $\neg R(\vec{a},\vec{b},\vec{z},\vec{w})$ and $R(\vec{a},\vec{c},\vec{d},\vec{w})$ in '$C[ROLD] \wedge \neg C[TAX]$'.

The overall method which extends the technique in [10] to the transaction notion is as follows.

1. Partition T into T_c and T_s, if possible.
2. Construct a transition axiom for each relation R which is affected by T_c using algorithm 3.2.
3. Follow the algorithm in [10] to construct Comp-Test(T_c) with '$C[ROLD] \wedge \neg C[TAX]$' reflected in accordance with the effect of T_s.

Then a complete test for a transaction T which is evaluated on the initial database, is the output of step 3. The correctness of our method can be easily seen by Lemma 2.1, Theorem 3.1 and Theorem 3.2.

Example 3.4 (shows the overall steps)

IC: $(\forall x)(\forall y)(\exists z)$
$(SALE(x,y) \Rightarrow SUPPLY(z,x,y))$
T 1. INSERT $SALE(a,b)$
2. INSERT $SUPPLY(c,a,b)$
3. DELETE $SUPPLY(e,f,g)$;

At step 1, we partition T into,
$T_s = \{3\}$, $T_c = \{1,2\}$.
For T_c, the transition axiom for $SALE$ is,
$(\forall x)(\forall y)$ $(SALENEW(x,y) \Leftrightarrow$
$(SALEOLD(x,y) \vee (x,y) = (a,b)))$.
The transition axiom for $SUPPLY$ is,
$(\forall x)(\forall y)(\forall z)$ $(SUPPLYNEW(x,y,z) \Leftrightarrow$
$(SUPPLYOLD(x,y,z) \vee (x,y,z) = (c,a,b)))$.
$\neg C[TAX]$ is, $\neg((\forall x)(\forall y)(\exists z)$ $(SALENEW(x,y) \Rightarrow$
$SUPPLYNEW(z,x,y)))$.
Then $C[ROLD] \wedge \neg C[TAX]$ is as below.
1. $\neg SALE(x,y)$ $SUPPLY(f(x,y),x,y)$
2. $SALE(f_1,f_2)$ $f_1 = a$
3. $SALE(f_1,f_2)$ $f_2 = b$
4. $\neg SUPPLY(z,f_1,f_2)$
5. $\neg SUPPLY(e,f,g)$
6. $f_1 \neq a$ $f_2 \neq b$ $z \neq c$
7. (1,2,4) $f_1 = a$
8. (1,3,4) $f_2 = b$

Note that clause 5 reflects the effect of T_s.
By the deletion strategies [10], we have
1. $\neg SALE(x,y)$ $SUPPLY(f(x,y),x,y)$

5. $\neg SUPPLY(e, f, g)$
9. (4,7,8) $\neg SUPPLY(z, a, b)$
10. $z \neq c$

From the above clauses, we can derive a refutation.

11. (10, $x = x$) □

That is, Comp-Test(T) = $\emptyset$. The transaction T never violates IC. □

4 Closing Remarks and Discussions

We have presented the general algorithm of enforcing ICs against a transaction T in the context of McCune and Henschen's method. The proposed method includes the construction of transition axioms for T, and the simplification of T through partitioning. It is not hard to see that our method is fully compatible with the notion of IC compilation.

It commonly occurs for a transaction to be associated with multiple ICs. Database statements which do not violate a constraint IC_i, are very likely to violate another constraint IC_j. Partitioning a transaction into appropriate subtransactions can reduce the overhead of IC checking significantly. One way of partitioning a transaction into two disjoint subtransactions has been presented.

The assumption that the structure of a transaction is known beforehands is practical in many areas. The notion of the *stored procedure* in Sybase is naturally compatible with our assumption. The application areas include on-line transaction processing in which structures of transactions are generally expected, and knowledge-base systems where maintenance of data and knowledge consistencies is important.

References

[1] Bernstein, P. and Blaustein, B., A Simplification of Algorithm for Integrity Assertions and Concrete Views, *Proc. of COMPSAC-81*, IEEE, 1981, pp. 90-99.

[2] Bry, F., Decker, H. and Manthey, R., *A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases*, *Proc. of Extending Database Technology*, 1988, pp. 488-505.

[3] Chang, C. and Lee, R.C., Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

[4] Das, S. and Williams, H., A Path Finding Method for Constraint Checking in Deductive Databases, *Data and Knowledge Engineering* 4:223-244 (1989).

[5] Gallaire, H., Minker, J. and Nicolas, J-M., Logic and Databases: A deductive approach, *ACM Computing Surveys* 16(2):153-185 (June 1984).

[6] Haerder, T. and Reuter, A., Principles of Transaction-Oriented Database Recovery, *ACM Computing Surveys* 15(4):287-317 (Dec. 1983).

[7] Hue, A. and Imielinski, T., Integrity Checking for Multiple Updates, *Proc. of ACM SIGMOD*, 1985, pp. 152-168.

[8] Kobayashi, I., Validating Database Updates, *Information System* 9(1):1-17 (1986).

[9] Lee, S.H., On compilation of Integrity Constraints against Transaction Updates, Master Thesis, Dept. of EECS, Northwestern University, Evanston, IL, 1986.

[10] McCune, W. and Henschen, L., Maintaining State Constraints in Relational Databases: A Proof Theoretic Basis, *ACM Journal of ACM* 36(1):46-68 (Jan. 1989).

[11] Nicolas, J-M., Logic for Improving Integrity Checking in Relational Data Bases, *ACTA Informatica* 18:227-253, 1982.

[12] Olive, A., Integrity Constraints Checking in Deductive Databases, *Proc. of 17th Very Large Data Bases*, pp.513-523, 1991.

[13] Qian, Xiaolei, An Efficient Method for Integrity Constraint Simplification, *Proc. of 4th Data Engineering*, 1988, pp. 338-345.

[14] Reiter, R., On Closed World Databases, in *Logic and Data Bases*, H.Gallaire and J.Minker Eds., Plenum Press, New York, 1978, pp. 55-76.

[15] Reiter, R., Towards a Logical Reconstruction of Relational Database Theory, in *Conceptual Modeling, Perspectives from Artificial Intelligence, Databases and Programming Languages*, M.Brodie, J. Mylopoulos and J. Schmidt Eds., Springer-Verlag, New York, 1984, pp. 191-233.

[16] Sadri, F. and Kowalski, R., A Theorem-Proving Approach to Database Integrity, in *Foundation of Deductive Databases and Logic Programming*, J.Minker Eds., Morgan Kaufmann Publisher, CA, pp. 313-362.