Semantic Update Optimization in Active Databases*

Jong P. Yoon
Department of Computer Science
Sookmyung Woman's University
Chungpa-Dong 2 Ka, Yongsan Ku, Seoul, South Korea
and
Larry Kerschberg
Center for Information Systems Integration and Evolution
Department of Information and Software Systems Engineering
School of Information Technology and Engineering
George Mason University, Fairfax, VA 22030-4444
{jyoon,kersch}@isse.gmu.edu

Abstract

In an active database, an update may be constrained by integrity constraints, and may also trigger rules that, in turn, may affect the database state. The general problem is to effect the update while also managing the "side-effects" of constraint enforcement and rule execution. In this paper an update calculus is proposed by which updates, constraints and rules are specified and managed within the same formalism. Constraints and production rules are expressed in a constraint language based on first-order logic. These logic expressions are used to semantically transform an original update into a sequence of updates that reflect the relevant constraints and production rules. The inference mechanism associated with processing a reformulated query ensures that: 1) the pre- and post-conditions of an update are satisfied, 2) update side-effects are propagated, and 3) repairs are made to tuples exhibiting constraint violations. Thus, a user-specified "update" is transformed, through semantic reformulation techniques, into a sequence of updates which together ensure semantic integrity of the original update as well as its propagated side-effects.

---

*This research was supported in part by an ARPA grant, administered by the Office of Naval Research under grant number N00014-92-J-4038.

This research presents several contributions. Integrity constraints and production rules are expressed in a declarative formalism so that they may be reasoned about. The update calculus formalism handles the semantic reformulation of an update to reflect relevant constraints and rules governing it. Finally, an algorithm is presented to handle constraint enforcement, production rule firing, and subsequent repair actions.

1 Introduction

Active databases, with their rule-processing capabilities, offer powerful mechanisms for the invocation of production rules that can reason about and update the database state. Because these rules are equivalent to database "triggers," we need ways to manage their execution so as to ensure consistent database states. Thus, we have two problems: 1) managing integrity constraints defined by logical formulae, and 2) supporting updates (insert, delete, modify) to the database while ensuring that constraint violations can be repaired.

The approach taken in this work is to define a constraint language that can be used to express not only integrity constraints, but also production rules for both the propagation of update effects and the repair of constraint violations. A user's update is posed as an SQL query which is then reformulated semantically with pre-conditions representing integrity constraints and post-conditions representing update effects and repair actions needed to maintain the relevant constraints. The reformulated query is then processed so as to ensure a consistent database state. In our approach the user, who may not be aware of the "triggers" to be activated by his query, may "preview" the update effects and potential actions by examining the pre- and post- conditions of the reformulated query.

The contributions of this work are the following: 1) a formalism to express both constraints and rules in an update language, 2) a semantic reformulation technique that transforms a user update into a sequence of updates incorporating relevant constraints and rules, and 3) an algorithm to execute the sequence of updates, maintain semantic integrity of the database, invoke associated triggered rules, and repair possibly inconsistent tuples.

1.1 Motivating Examples

EXAMPLE 1.1

Consider an employee-dependent database consisting of two relations:

`emp`(E, X, S, T)

% employee with employee number E, years of experience X, salary S, and tax rate T expressed in percent, and

`depn`(P, E)

% person with name P is a dependent of employee with number E.

Assume that an integrity constraint, $IC1$, specifies that employees who have more than 5 years experience should earn more than \$50K. Assume further that the database is in a consistent state. When employee salaries are increased by 10 percent, using UPDATE `emp` SET $S := S \times 1.1$, one needs to verify that the new salary S satisfies constraint $IC1$. This verification is performed one tuple at a time. Suppose, however, that the integrity constraint is associated directly with the update expression. The update may therefore be expressed as UPDATE `emp` SET $S := S \times 1.1$ WHERE $IC1$. The "where" clause expresses the constraint that those employees with more than five years experience should have a salary exceeding \$50K.

Further, if we have an additional integrity constraint $IC2$, specifying that the tax rate of employees who earn over \$50K should be more than 15 percent, we have a situation where the tax rate for an employee may have to be updated to satisfy constraint $IC2$. Alternatively, suppose, that the "where" clause of an update expression consists of both a pre-condition and a post-condition. The update may therefore be expressed as UPDATE `emp` SET $S := S \times 1.1$

WHERE $IC1$ is satisfied AND $IC2$ is not violated. Note that constraint satisfaction and constraint violation will be defined later in Section 3. The update will be complete if these two associated constraints are satisfied. However, what if the integrity constraint $IC2$ were violated? The following example provides an answer to this question. □

EXAMPLE 1.2

Consider a production rule $PR1$, in the employee-dependent database. $PR1$ produces the tax rate X, say 20%, for those who earn over \$50K and have more than three dependents. In the previous example, the update is executed for the tuples which satisfy the constraint $IC1$. It is also "committed" for tuples which do not violate the constraint $IC2$. Suppose, however, that some tuples, if not all the violating tuples, have salary over \$50K, and, in addition, have more than three dependents. By executing the rule $PR1$, we obtain the proper tax rate for employees earning over \$50K (by constraint $IC2$) and those who in addition have three dependents (by production rule $PR1$). Thus productions may be specified to specify how constraints are to be enforced. □

1.2 Related Work

Active databases monitor the database states as shown in [AWH92,DBB+88,DHL90,MD89] and hence appropriate rules are activated to trigger additional actions if the database state changes. These rules are activated in response to an update without user intervention.

A declarative update can be transformed into a procedural specification of database state transitions. The work on update specification transformation is investigated in [Man89, QW88]. View update problems have been dealt with by many researchers. Kakas and Mancarella [KM90] use an abductive approach so that constraint checking associated with an update is incorporated into the update to reject the generation of inconsistent states. Ceri and Widom [CW91] provide a facility whereby a user defines a view as an SQL expression, from which productions are generated to maintain a materialization of that view. Gottlob, Paolini, and Zicari [GPZ88] describe how primitive update operators can be rewritten into complex updates and how view updates are translated into database updates. Our approach is similar to this approach in that an update is performed within a view.

Kramer et al. [KLS92] and Cacace et al. [CCCR$^+$90] incorporate *updates into rule languages*. Widom et al. [WCL91] introduce an SQL-based production rule language into the *Starburst* rule system. Our approach incorporates *rules and constraints into updates*. That is, the update calculus described in this paper is an SQL-like language augmented by the semantics of rules and constraints.

Constraint checking and constraint violation repair are important issues in active databases. Many researchers have developed formalisms for specifying constraints [Kow78,Mor86,SK86] and enforcing constraints [CGM90,DBB$^+$88,SK86]. A constraint violation repair method has been proposed by Moerkotte and Lockemann [ML91]. They assume that constraint violations are caused by an unsound transaction and therefore symptoms causing the inconsistency are removed from that transaction. In contrast, we assume that constraint violations are caused by incomplete update specifications. The effects of an update are propagated, and database instances not satisfying constraints may be corrected. Ceri and Widom [CW90] have used production rules to repair inconsistent states. They present a method for translating constraints, which are used to detect inconsistent states, into constraint maintaining production rules. The translation, however, requires user intervention; it is static and manual.

Finally, the research of Hecht and Kerschberg [HK81], Morgenstern [Mor84] and Abiteboul and Hull [AH85] addresses the need for update propagation for maintaining overall database consistency.

1.3 Outline of Paper

The remainder of this paper is organized as follows: Section 2 formalizes the constraint language in first-order logic. When an update is posed to a database, constraints are enforced. Update verification using constraints is described in Section 3. Appropriate constraints are converted into SQL query expressions. Section 4 extends the constraint formalism to active database rules and shows that rule conversion into update expressions is similar to the technique shown in Section 3. Section 5 describes the repair technique for constraint violations. Section 6 applies these techniques to the propagation of update effects. Section 7 discusses implementation issues, and Section 8 presents our conclusions.

2 Constraint Language

The constraints we consider are expressed in first order logic. The syntax is adopted from Gupta and Widom [GW93]. The difference is a simplification for single database constraints and an extension to active database rules.

2.1 Syntax

An integrity constraint, denoted *IC*, is a first order logic sentence of the following form:

$$(IC): \quad \forall \bar{X} \exists \bar{Y} \; [R_1(\bar{X}_1) \wedge ... \wedge R_k(\bar{X}_k) \wedge g(\bar{X}, \bar{Y}, \bar{c}) \\ \Longrightarrow S_1(\bar{X}_1', \bar{Y}_1') \vee ... \vee S_n(\bar{X}_n', \bar{Y}_n')]$$

where

$R_1, ..., R_k, S_1, ...S_n$ represent relations.

$\bar{X} = \{X_1, ..., X_t\}$ is a set of universally-quantified ($\forall$) variables occurring only in $R_1, ..., R_k$, and g.

$\bar{Y} = \{Y_1, ..., Y_u\}$ is a set of existentially-quantified ($\exists$) variables occurring only in $S_1, ..., S_n$, and g.

$\bar{c} = \{c_1, ..., c_w\}$ is a set of constants occurring only in g.

$g(\bar{X}, \bar{Y}, \bar{c})$ is a conjunction of equality ($=$) and inequalities ($\neq, >, <, \geq, \leq$) involving variables from $\bar{X}$ and $\bar{Y}$.

$\bar{X}_i \subseteq \bar{X}$ is the set of variables that occur in $R_i, 1 \leq i \leq k$.

$\bar{X}_i' \subseteq \bar{X}$ is the set of universally-quantified variables that occur in $S_i, 1 \leq i \leq n$.

$\bar{Y}_i' \subseteq \bar{Y}$ is the set of existentially-quantified variables that occur in $S_i, 1 \leq i \leq n$.

2.2 Semantics

Assume that the domain of each variable in $\bar{X}$ and $\bar{Y}$ is the domain of the relation attribute in which that variable appears. The integrity constraint is *satisfied* if for all value assignments to the variables in $\bar{X}$ there exists an assignment of values to variables in $\bar{Y}$ such that **either**

1. For each $R_i, 1 \geq i \geq k$ in *IC*, there does not exist a tuple in the relation R_i with the values assigned to $\bar{X}_i$, **or**

2. Predicate g is not satisfied using constants $\bar{c}$ and the values assigned to $\bar{X}$, and $\bar{Y}$, **or**

3. For some $S_i, 1 \leq i \leq n$ in IC, there is a tuple in relation S_i with the values assigned to $\bar{X}_i'$ and $\bar{Y}_i'$.

We express the constraints of Example 2.1 as first order logic sentences in the form described as above.

EXAMPLE 2.1 Constraint $IC1$ specifies that employees who have more than 5 years of experience should earn more than \$50K.

$$(IC1): \quad \forall E, X, T, \exists S\ [\texttt{emp}(E, X, S, T) \land (X > 5) \\ \Longrightarrow (S > 50K)]$$

Constraint $IC2$ specifies that the tax rate of employees who earn over \$50K should be more than %15.

$$(IC2): \quad \forall E, X, S, \exists T\ [\texttt{emp}(E, X, S, T) \land (S > 50K) \\ \Longrightarrow (T > .15)]$$

□

3 Constraint Management During Updates

A database is said to be consistent if all integrity constraints are satisfied by a database state. However, if a database is updated, the database that is initially consistent with respect to a set of integrity constraints can become inconsistent. The problem is further complicated when the side-effects of an update are propagated. In this section we present an update language that incorporate pre- and post-conditions for an original update. These conditions contain constraints and productions that can be used for managing the consistency of the database.

3.1 The Update Language

As seen in Example 1.1, the constraint $IC1$ serves as a pre-condition for the update while the constraint $IC2$ serves as a post-condition. The update can be executed (and committed) if the pre-condition is satisfied. Moreover, the side-effects of an update may cause additional inconsistencies. Note that the typical `UPDATE - SET - WHERE` expression [KS91] verifies only the pre-condition using the "`WHERE`" clause, but not the post-condition. To verify the post-condition as well, we propose an update expression. The update expression has associated with it both a pre- and post-condition as shown below:

`UPDATE` relation
`SET` assignments
`PRECOND` *constraints* are satisfied
`POSTCOND` *constraints* are not violated

Using available constraints and rules, we reformulate a user-specified update into a semantically-rich update sequence. Section 3 discusses the conversion of constraints and rules into SQL expressions, and Section 4 associates those converted constraints and rules with an update.

3.2 Converting Constraints to Query Expressions

This section describes how to convert constraints to SQL query expressions. Constraints by nature ensure that a database state is consistent. Hence, a database state can either satisfy the constraints or violate them. Before developing the conversion technique further, we define the notions of *constraint satisfaction* and *constraint violation*.

Definition 1 (*Constraint Satisfaction*). *Constraint* $p \Longrightarrow q$ *is satisfied by a database if either* p *is false or* q *is true in the database.* □

Definition 2 (*Constraint Violation*). *Constraint $p \Longrightarrow q$ is violated by a database if p is true but q is false in the database.* □

Constraint IC can be satisfied by part of a database, if not an entire database, and it can also be violated by part of the database, if not an empty database. Consider the following general constraint IC:

$$(IC)\text{: } \forall \bar{X} \exists \bar{Y}\ [R_1(\bar{X}_1) \wedge ... \wedge R_k(\bar{X}_k) \wedge g(\bar{X}, \bar{Y}, \bar{c}) \Longrightarrow S_1(\bar{X}'_1, \bar{Y}'_1) \vee ... \vee S_n(\bar{X}'_n, \bar{Y}'_n)]$$

The set of tuples *satisfying the constraint IC* is expressed as the SQL query.

SELECT $*$
FROM $R_1(\bar{X}_1), ..., R_k(\bar{X}_k), S_1(\bar{X}'_1, \bar{Y}'_1), ..., S_n(\bar{X}'_n, \bar{Y}'_n)$
WHERE $\neg g(\bar{X}, \bar{Y}, \bar{c})$

The above SQL query results in a set of tuples satisfying the constraint IC, that is, those tuples satisfy the condition $\neg g(\bar{X}, \bar{Y}, \bar{c})$. Clearly, therefore, if the result of the above SQL query is empty, it means that all the database states are not correct, that is, the database is inconsistent.

The set of tuples *violating the constraint IC* is expressed as the SQL query.

SELECT $*$
FROM $R_1(\bar{X}_1), ..., R_k(\bar{X}_k), S_1(\bar{X}'_1, \bar{Y}'_1), ..., S_n(\bar{X}'_n, \bar{Y}'_n)$
WHERE $g(\bar{X}, \bar{Y}, \bar{c})$

The result is a set of tuples, each of which violates the constraint IC. If the result set is empty, the database is said to be consistent with regard to the constraint IC.

The first expression specifies a set of tuples for which the update effects must be propagated if the operation is not to be aborted. The second expression specifies a set of tuples which may be repaired if alerts are not the best solution.

EXAMPLE 3.1 Constraint $IC1$ specifies that employees who have more than 5 years of experience should earn more than \$50K.

$$(IC1): \quad \forall E, X, T, \exists S \; [\texttt{emp}(E, X, S, T) \wedge (X > 5) \\ \implies (S > 50K)]$$

Clearly, this constraint is equivalent to $\forall E, X, T, \exists S[\texttt{emp}(E, X, S, T) \wedge (X > 5) \wedge (S > 50K) \implies]$ as far as the definition is concerned in Section 2.1.

The set of tuples satisfying $IC1$ is expressed as

```
SELECT   *
FROM     emp
WHERE    ¬(X > 5) OR (S > 50K)
```

The set of tuples not satisfying $IC1$ is expressed as

```
SELECT   *
FROM     emp
WHERE    (X > 5) AND ¬(S > 50K)
```

A set of tuples violating the constraints will be repaired using techniques presented in Section 5. The database is said consistent if this SQL query returns the empty set. □

3.3 Update Verification Using Constraints

When an update U is posed to an active database, it is likely that constraints are available for checking the database state. The compilation of appropriate constraints is another consideration [YK92]. This paper, however, describes a method of confining the scope of the side-effects of an update. Suppose that constraints IC_i and IC_j are available. The constraint IC_i checks database states for the update U and the constraint IC_j checks results of the update. The database tuples where U does not apply can be moved outside the scope of the update process. Similarly, the database tuples where the effects of U causes additional violations can be moved outside the scope of the update commit. By combining these two scopes, a user-issued update expression can be rewritten:

```
UPDATE    relations
SET       assignment in $U$
PRECOND   EXIST tuples satisfying $IC_i$ AND the condition of $U$
POSTCONDEXIST tuples satisfying $IC_j$
```

The condition of the PRECOND and POSTCOND clauses specifies a set comparison between join attributes. The scope of either a constraint or a rule may be expressed as predicates in SQL, as will be demonstrated in the following example.

Example 3.2

Consider the following constraints.

$$
\begin{aligned}
(IC1): \quad & \forall E, X, T, \exists S \ [\texttt{emp}(E, X, S, T) \wedge (X > 5) \\
& \qquad \Longrightarrow (S > 50K)] \\
(IC2): \quad & \forall E, X, S, \exists T \ [\texttt{emp}(E, X, S, T) \wedge (S > 50K) \\
& \qquad \Longrightarrow (T > .15)]
\end{aligned}
$$

Suppose that once again the update is posed to augment employee salaries by ten percent.

```
UPDATE    emp
SET       $S = S * 1.1$
```

Consistent update is ensured by using the above two constraints. The first constraint serves as the pre-condition, while the second constraint serves as the post-condition. Now, we discuss how the PRECOND and POSTCOND are expressed. Recall that a constraint $p \rightarrow q$ holds if both p and q are true or p is false. By the same token, a constraint $p \rightarrow q$ does not hold if p is true and q is not. Using converted query expressions as shown in Example 3.1, the given update can be reformulated as following:

```
UPDATE      emp
SET         S := S * 1.1
PRECOND     EXIST (SELECT  *
                   FROM    emp
                   WHERE   ¬(X > 5) OR S > 50000)
POSTCONDEXIST (SELECT  *
                   FROM    emp
                   WHERE   ¬(S > 50000) OR T ≥ .15)
```

A user-specified update was reformulated into the semantically-rich update shown above. The reformulated update verifies the salary update itself and furthermore, checks the tax rate which may be affected by the update. The next sub-section describes how to repair constraint violations. It handles, for example, possible modification of *taxRate* resulting from the update on *salary*. □

4 Incorporating Rules into the Constraint Language

We now extend the method of constraint management to incorporate production rules used in active databases. A production rule executes a sequence of actions if the conditions of its left-hand-side are satisfied. We consider only production rules which update a database, that is, insert, delete, or modify, but not other user-defined programs, such as methods.

4.1 Syntax

A production rule, denoted *PR*, is a first order logic sentence similar to the constraint specification. The difference from the constraint language is that its consequent is a sequence of database operations (update, insert, and delete) on a database. In active databases, it is well known that those operations are executed in the order specified in a rule. The rule is defined as the following form:

$$\begin{aligned}(PR):\quad & \forall\bar{X}\exists\bar{Y}\ [R_1(\bar{X}_1)\wedge ... \wedge R_k(\bar{X}_k)\wedge g(\bar{X},\bar{Y},\bar{c}) \\ & \Longrightarrow O(S_1(\bar{X}_1',\bar{Y}_1'))\wedge ... \wedge O(S_n(\bar{X}_n',\bar{Y}_n'))]\end{aligned}$$

where

$R_1, ..., R_k, S_1, ..., S_n$ represent relations.

O represents a database modification operation such as update, insert or delete.

$\bar{X} = \{X_1, ..., X_t\}$ is a set of universally-quantified variables occurring only in $R_1, ..., R_k$, and g.

$\bar{Y} = \{Y_1, ..., Y_u\}$ is a set of existentially-quantified ($\exists$) variables occurring only in $S_1, ..., S_n$, and g.

$\bar{c} = \{c_1, ..., c_w\}$ is a set of constants occurring only in g.

$g(\bar{X},\bar{Y},\bar{c})$ is a conjunction of equality ($=$), inequalities ($\neq, >, <, \geq, \leq$), and assignment ($:=$) involving variables from $\bar{X}$ and $\bar{Y}$.

$\bar{X}_i \subseteq \bar{X}$ is the set of variables that occur in $R_i, 1 \leq i \leq k$.

$\bar{X}_i' \subseteq \bar{X}$ is the set of universally-quantified variables that occur in $S_i, 1 \leq i \leq n$.

$\bar{Y}_i' \subseteq \bar{Y}$ is the set of existentially-quantified variables that occur in $S_i, 1 \leq i \leq n$.

4.2 Semantics

If O in $O(S_i(\bar{X}_i',\bar{Y}_i'))$ denotes UPDATE, $g(\bar{X},\bar{Y},\bar{c})$ is a conjunction of $g_1(\bar{X},\bar{Y},\bar{c})$ and $g_2(\bar{X},\bar{Y},\bar{c})$, where $g_1(\bar{X},\bar{Y},\bar{c})$ denotes equality ($=$) or inequalities ($\neq, >, <, \geq, \leq$) and $g_2(\bar{X},\bar{Y},\bar{c})$ denotes assignment (:=). The rule *PR* can be rewritten as.

$$
\begin{aligned}
(PR)\text{: } & \forall\bar{X}\exists\bar{Y}\ [R_1(\bar{X}_1)\wedge ...\wedge R_k(\bar{X}_k)\wedge g_1(\bar{X},\bar{Y},\bar{c})\wedge g_2(\bar{X},\bar{Y},\bar{c}) \\
& \Longrightarrow O(S_1(\bar{X}'_1,\bar{Y}'_1))\wedge ...\wedge O(S_n(\bar{X}'_n,\bar{Y}'_n))]
\end{aligned}
$$

The production rule is *satisfied* if for all value assignments to the variables in $\bar{X}$ a database operation O is executed with a value assignment to variables in $\bar{Y}$ such that **if**

1. For each $R_i, 1 \geq i \geq k$ in *IC*, there does not exist a tuple in the relation R_i with the values assigned to $\bar{X}_i$, **and**
2. Predicate g_1 is satisfied by constants $\bar{c}$ and the values assigned to $\bar{X}$ and $\bar{Y}$, **and**
3. g_2 is satisfied using unifying the variables in $\bar{X}$ and $\bar{Y}$ with the values of $\bar{c}$, **then**
4. For some $S_i, 1 \leq i \leq n$ in *IC*, the database operation O is performed for a tuple in relation S_i with the values assigned to $\bar{X}'_i$ and $\bar{Y}'_i$.

4.3 Examples

EXAMPLE 4.1 Rule *PR*1 sets a tax rate of 20% for those who earn over \$50K and have more than three dependents.

$$
\begin{aligned}
(PR1)\text{:}\quad & \forall E,X,S,P,\exists T\ [\texttt{emp}(E,X,S,T)\wedge \texttt{depn}(P,E)\wedge (S>50K)\wedge \\
& (\mathsf{sum}(P)>3)\wedge (T:=.2)\Longrightarrow \mathsf{UPDATE}(\texttt{emp}(E,X,S,T))]
\end{aligned}
$$

Note that $\mathsf{sum}(P)$ returns the total number of appropriate P. The tuples in `emp`, if their salary is more than \$50K and they have more than three dependents, are updated with the value unified with variable T. □

EXAMPLE 4.2 Rule *PR*2 deletes the tuples for employees who have worked more than 20 years.

$$(PR2): \quad \forall E, X, S, P, \exists T\ [\texttt{emp}(E, X, S, T) \wedge (X > 20) \\ \Longrightarrow \mathsf{DELETE}(\texttt{emp}(E, X, S, T))]$$

The `emp` tuples, with experience greater than 20 years, are removed from `emp`. □

EXAMPLE 4.3 Rule *PR*3 creates the tuples of `high_paid` if their salary is more than \$80K.

$$(PR3): \quad \forall E, X, S, P, \exists T\ [\texttt{emp}(E, X, S, T) \wedge (S > 80000) \\ \Longrightarrow \mathsf{INSERT}(\texttt{high\_paid}(E, S))]$$

□

4.4 Converting Rules to Update Expressions

Rule *PR* can modify a portion of a database, if not the entire database. Consider the following production rule:

$$(PR)\text{: } \forall \bar{X} \exists \bar{Y}\ [R_1(\bar{X}_1) \wedge ... \wedge R_k(\bar{X}_k) \wedge g(\bar{X}, \bar{Y}, \bar{c}) \Longrightarrow O(S_i(\bar{X}_i', \bar{Y}_i'))]$$

Recall that O denotes either UPDATE, INSERT, or DELETE. If O in $O(S_i(\bar{X}_i', \bar{Y}_i'))$ denotes UPDATE, $g(\bar{X}, \bar{Y}, \bar{c})$ is a conjunction of $g_1(\bar{X}, \bar{Y}, \bar{c})$ and $g_2(\bar{X}, \bar{Y}, \bar{c})$, where $g_1(\bar{X}, \bar{Y}, \bar{c})$ denotes equality ($=$) or inequalities ($\neq, >, <, \geq, \leq$) and $g_2(\bar{X}, \bar{Y}, \bar{c})$ denotes assignment ($:=$). The rule *PR* can be rewritten as.

$$(PR)\text{: } \forall \bar{X} \exists \bar{Y}\ [R_1(\bar{X}_1) \wedge ... \wedge R_k(\bar{X}_k) \wedge g_1(\bar{X}, \bar{Y}, \bar{c}) \wedge g_2(\bar{X}, \bar{Y}, \bar{c}) \\ \Longrightarrow \mathsf{UPDATE}\ (S_i(\bar{X}_i', \bar{Y}_i'))]$$

Therefore, the rule *PR* can be converted into

```
UPDATE    $S_i(\bar{X}_i', \bar{Y}_i')$
SET       $g_2(\bar{X}, \bar{Y}, \bar{c})$
PRECOND   EXIST ( SELECT *
                  FROM $R_1(\bar{X}_1), ..., R_k(\bar{X}_k)$
                  WHERE $g_1(\bar{X}, \bar{Y}, \bar{c})$ )
```

Notice that the condition part of *PR* is to scope the problem view within which the assignment $g_2(\bar{X}, \bar{Y}, \bar{c})$ is executed. As seen in Section 2.3, the condition is called PRECOND. The POSTCOND clause consists of available constraints *IC* that verify the consistency of the update.

EXAMPLE 4.4 Rule *PR*1 produces a 20% tax rate for those who earn over $50K and have more than three dependents.

$$(PR1): \quad \forall E, X, S, P, \exists T\ [\mathtt{emp}(E, X, S, T) \wedge \mathtt{depn}(P, E) \wedge (S > 50K) \wedge \\ (\mathsf{sum}(P) > 3) \wedge (T := .2) \Longrightarrow \mathsf{UPDATE}(\mathtt{emp}(E, X, S, T))]$$

The rule *PR*1 can be converted into an update expression:

```
UPDATE    emp
SET       $T := 2$
PRECOND   EXIST ( SELECT $E$, sum($P$)
                  FROM emp, depn
                  WHERE $(S > 50K)$ AND $(\mathsf{sum}(P) > 3)$ )
```

The *impedance mismatch* problem has been solved by converting rules into an SQL expression so that sets of tuples may be examined versus tuple-at-a-time processing. A first-order-logic based rule formalism can be implemented in relational database management systems using these transformation techniques. □

5 Repairing Constraint Violations Using Rules

When the database state is updated, effects of the update can be propagated in active databases. The propagation of update effects may cause additional database inconsistencies. In traditional active database formalisms, these constraint violations must be corrected explicitly by users, or the updates causing the violation are rejected. Suppose, however, an appropriate rule were available in the database to deduce new facts to compensate for these constraint violations. This section describes how to make use of available rules for ensuring consistent database updates.

Consider an update U whose effects violate a constraint IC_j, and a rule PR_j whose actions can repair these constraint violations. By converting the rule PR_j as shown in earlier section, two update expressions are obtained.

UPDATE relation used in a user-issued query
SET assignment in U
PRECOND EXIST tuples satisfying IC_i AND the condition of U
POSTCONDEXIST tuples satisfying IC_j

UPDATE relation used in a rule
SET assignment in PR_j
PRECOND EXIST tuples violating IC_j AND the condition of PR_j
POSTCONDEXIST tuples satisfying IC_j

These two updates are activated in sequence; the converted update from a rule is performed before a user-issued update.

Example 5.1

Consider the following constraint and rule.

$$(IC2): \quad \forall E, X, S, \exists T \; [\mathtt{emp}(E, X, S, T) \wedge (S > 50K) \\ \Longrightarrow (T > .15)]$$

$$(PR1): \quad \forall E, X, S, P, \exists T \; [\mathtt{emp}(E, X, S, T) \wedge \mathtt{depn}(P, E) \wedge (S > 50K) \wedge \\ (\mathsf{sum}(P) > 3) \wedge (T := .2) \Longrightarrow \mathsf{UPDATE}(\mathtt{emp}(E, X, S, T))]$$

Suppose that the following update is posed to the active database.

```
UPDATE    emp
SET       S = S ∗ 1.1
```

As shown in Example 3.2, the constraint is used to ensure the post-condition of the update. At the same time, it is necessary to consider those tuples, if any, violating this constraint. If the above rule can be used to repair those constraint violations, the rule can be converted into an update expression. Clearly, the scope of those constraint violations should be taken into account in the pre-condition of the converted update, as expressed below.

```
UPDATE    emp
SET       S := S ∗ 1.1
POSTCONDEXIST ( SELECT   ∗
                FROM     emp
                WHERE    ¬(S > 50000) OR T ≥ .15)

UPDATE    emp
SET       T := 2
PRECOND   EXIST ( SELECT E, sum(P)
                  FROM emp, depn
                  WHERE (S > 50K) AND (sum(P) > 3)
                        AND ¬(T ≥ .15) )
POSTCOND EXIST ( SELECT   ∗
                 FROM     emp
                 WHERE    ¬(S > 50K) OR T ≥ .15)
```

These two updates are executed sequentially: the first update is to set the salary S for those tuples satisfying the pre-condition and to commit this update to those tuples satisfying the post-condition. The second update is to set the tax rate T for those tuples violating the post-condition of the first update and to commit this update to those tuples satisfying that post-condition. □

6 Propagation of Update Effects

In response to a database state change, active database rules are activated without the user's intervention. That is, database state changes trigger further rule activation and execution. Suppose that a rule PR_k is triggered in response of database state changes caused by an update U. For a given U, a sequence of updates is obtained as follows.

```
UPDATE    relation
SET       assignment in $U$
PRECOND   the condition of $v(U)$

UPDATE    relation
SET       assignments in $PR_k$
PRECOND   the condition of $PR_k$
```

Example 6.1

Consider the following update which is to increase employee salaries by ten percent where *years_of_expr* is over 2 years.

```
UPDATE   emp
SET      $S = S * 1.1$
WHERE    EXIST (SELECT   *
                FROM     emp
                WHERE    $X > 2$)
```

Suppose that the following rule $PR4$ can be triggered by the database state changes caused by the above update.

$$
\begin{aligned}
(PR4): \quad \forall E, X, T, S, \exists Y \ & [\texttt{emp}(E, X, S, T) \wedge \texttt{high\_paid}(E, Y) \wedge \\
& (S > 80K) \wedge (Y := \text{“high”}) \\
& \Longrightarrow \mathsf{UPDATE}(\texttt{high\_paid}(E, Y))]
\end{aligned}
$$

By incorporating rule *PR*4, the given update is reformulated to the following two update expressions:

```
UPDATE    emp
SET       $S = S * 1.1$
WHERE     EXIST (SELECT   *
                 FROM     emp
                 WHERE    $X > 2$)

UPDATE    high_paid
SET       $Y :=$ “high”
WHERE     EXIST (SELECT   $E$
                 FROM     emp
                 WHERE    $S > 80K$)
```

This sequence of two updates explains propagation of the update effects. If constraints regarding the changes to *salary* are available, they are taken into account by the PRECOND clause of the second update. If constraints regarding additional attributes affected by the changes to *salary* are available, they are used for the POSTCOND clause. □

7 Implementation Issues

The reformulation process described in this paper is depicted in Figure 1. For a user-issued update expression, appropriate constraints and rules need to be compiled; a discussion of compilation techniques appears in [Yoo93], but is beyond the scope of this paper. The output of this reformulation process is a sequence of updates that represents the semantic reformulation of the user's original update.

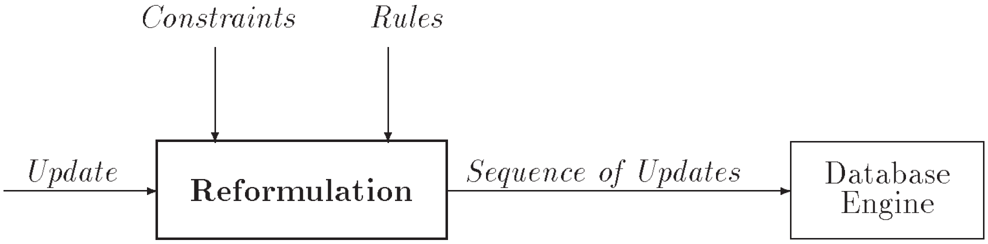

Figure 1: Semantic Update Reformulation

Consider now the actual processing of updates as depicted in Figure 2. The flow diagram shown in Figure 2 represents the integration of three major tasks of semantic update processing: update verification, constraint violation repair, and update effects propagation, as discussed in Sections 3, 5, and 6, respectively.

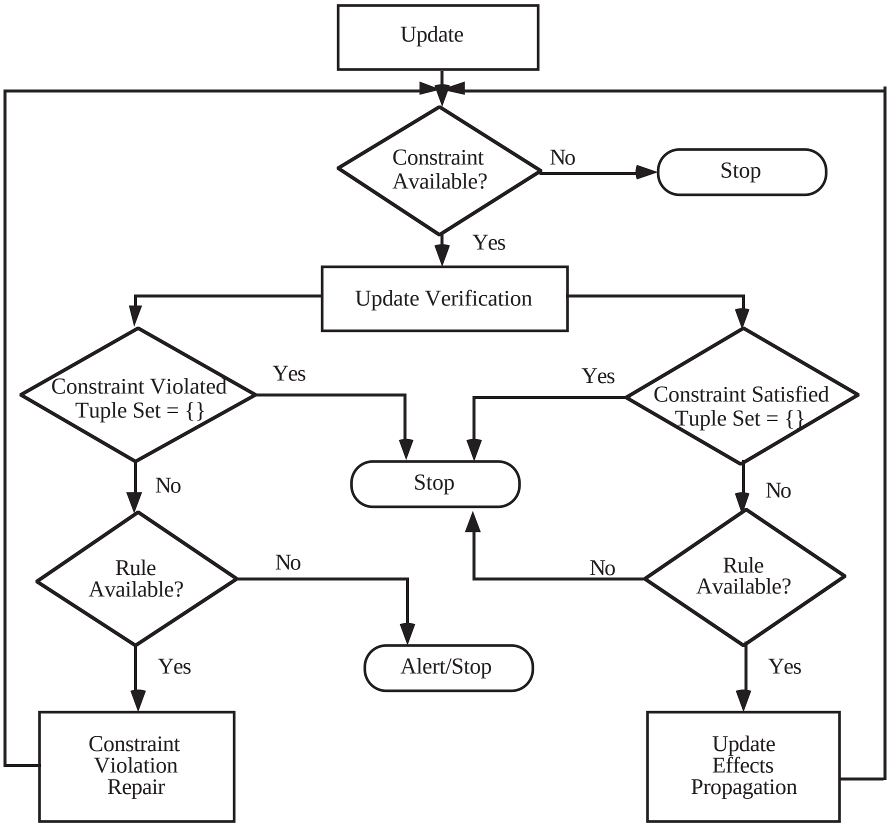

Figure 2: Semantic Update Processing Diagram

First we consider the update verification phase, in which the database is partitioned into two parts: a set of tuples satisfying the constraints and the remaining tuples which violate the constraints. The constraint language requires syntactic interpretation to be expressed in the SQL query language. In an implementation, this syntactic interpretation may be performed by using a parser and a lexical analyzer, or by indexing constraints to a data dictionary containing pre-defined query expressions. The advantage of our conversion technique is that set-at-a-time constraint evaluation is performed rather than tuple-at-a-time evaluation.

The repair phase of a constraint violation is processed only if the database state violates a constraint and a rule is available for deducing facts. The deduced facts may be corrections if they, in turn, satisfy all the constraints that were not satisfied originally. Otherwise, an alerter may be generated to appraise users of the constraint violation; the user can then take appropriate action.

In the propagation of update effects phase – a key feature of active databases – one or more rules may be activated as a side-effect of an update. In an implementation, rules can be activated against either those valid tuples which satisfy all constraints, or those tuples which are to be repaired due to constraint violations.

Note that the constraint violation repair and the update effects propagation phases may be performed in parallel. The database can be partitioned into tuples that satisfy the constraints and those the violate the constraints. We can take advantage of parallel algorithms and multiprocessor architectures, e.g., DB2 V3 or ORACLE Parallel Server V7, for query optimization.

8 Conclusions

This paper has presented a novel approach to constraint management in active databases. Updates to a database are reformulated to employ the semantics of both rules and constraints. We have presented a unified approach to consistent update management using constraints and rules in databases, and have made the following contributions:

- A unified database *update calculus* has been developed. In this approach, a user-specified database update is reformulated into a sequence of semantically-rich updates that have associated with them relevant constraints and rules.
- A system-derived update incorporates constraints and rules, so that database consistency can be maintained efficiently. The user may preview the effects of an update by examining reformulated sequence of updates.
- Updates are performed set-at-a-time only on valid database instances and the update effects are propagated only on valid instances. Invalid instances are repaired by rule deduction.

The benefits described in this paper include the following:

- Conversion of constraints and rules to SQL expressions supports set-manipulation in the update calculus versus the typical tuple-at-a-time rule evaluation used in other update schemes.
- Semantic query optimization [YK93] is a particular case of semantic update optimization discussed in this paper. Therefore, the semantic update reformulation framework can

be used to optimize queries semantically [LHQ91].

- Update reformulation provides users with a pre-viewing mechanism of active database rule processing. Users may also have control of rule activation, if necessary.

The formalism discussed in this paper will help database designers and database users to manage and control database updates in active databases. The techniques of constraint management – update verification, constraint violation repair, and update effects propagation – applied to the reformulation and management of updates, will permit users and developers to have increased confidence that their active applications are performing as designed and with the appropriate results.

References

[AH85] S. Abiteboul and R. Hull. Update propagation in the IFO database model. In *Proceedings of the International Conf. on Foundations of Data Organization*, pages 243–251, Kyoto, 1985.

[AWH92] Alexander Aiken, Jennifer Widom, and Jeseph M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In Michael Stonebraker, editor, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 59–68, San Diego, 1992.

[CCCR+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In Hector Garcia-Molina and H.V. Jagadish, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 225–236, Atlantic City, NJ, 1990.

[CGM90] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):163–207, June 1990.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 650–661, Brisbane, Australia, 1990.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 577–589, Barcelona, 1991.

[DBB$^+$88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, R. Ledin M. Hsu, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1):51–70, March 1988.

[DHL90] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Organizing long running activities with triggers and transactions. In Hector Garcia-Molina and H.V. Jagadish, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 204–214, Atlantic City, NJ, 1990.

[GPZ88] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.

[GW93] Ashish Gupta and Jennifer Widom. Local verification of global integrity constraints in distributed databases. In Peter Buneman and Sushil Jajodia, editors, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–58, Washington, D. C., 1993.

[HK81] Matthew S. Hecht and Larry Kerschberg. Update semantics for the functional data model. Technical Report Database Research Report No. 4, Bell Laboratories, Holmdel, New Jersey, January 1981.

[KLS92] M. Kramer, G. Lausen, and G. Saake. Updates in a rule-based language for objects. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 251–262, Vancouver, Canada, 1992.

[KM90] A. C. Kakas and P. Mancarella. Database updates through abduction. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 650–661, Australia, 1990.

[Kow78] R. Kowalski. Logic for data description. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 77–103, New York, 1978. Plenum Press.

[KS91] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill, Inc, New York, 1991.

[LHQ91] Sanggoo Lee, Lawrence J. Henschen, and Ghassan Z. Qadah. Semantic query reformulation in deductive databases. In *Intl. Conf. on Data Engineering*, pages 232–239, 1991.

[Man89] Sanjay Manchanda. Declarative expression of deductive database updates. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 93–100, Philadelphia, 1989.

[MD89] D. R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 215–224, Portland, Oregon, 1989.

[ML91] Guido Moerkotte and Peter C. Lockemann. Reactive consistency control in deductive database. *ACM Transactions on Database Systems*, 16:670–702, 1991.

[Mor84] M. Morgenstern. Constraint equations: Declarative expression of constraints with automatic enforcement. In *Proc of VLDB*, pages 111–125, 1984.

[Mor86] M. Morgenstern. The role of constraints in databases. In Larry Kerschberg, editor, *Expert Database Systems*, pages 351–368. Benjamin/Cummings, 1986.

[QW88] Xialolei Qian and Richard Waldinger. A transaction logic for database specification. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 243–250, Chicago, 1988.

[SK86] A. Shepherd and L. Kerschberg. Constraint management in expert database systems. In Larry Kerschberg, editor, *Expert Database Systems*, pages 309–368. Benjamin/Cummings, 1986.

[WCL91] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 275–285, Barcelona, Spain, 1991.

[YK92] Jong P. Yoon and Larry Kerschberg. A framework for constraint management in object-oriented databases. In *Proc. of the First International Conference on Information and Knowledge Management*, pages 292–299, Baltimore, MD, 1992.

[YK93] Jong P. Yoon and Larry Kerschberg. Semantic query optimization in deductive object-oriented databases. In *Proc. of the Third International Conference on Deductive and Object-Oriented Databases*, Phoenix, Arizona, December, 1993.

[Yoo93] Jong P. Yoon. *Constraint Management in Active Databases.* PhD thesis, George Mason University, Fairfax, Virginia, 1993.