Implementing Temporal Integrity Constraints Using an Active DBMS

David Toman Jan Chomicki
Department of Computing and Information Science
Kansas State University, Manhattan, KS 66506
{david,chomicki}@cis.ksu.edu

Abstract

The system presented in this paper allows automatic and efficient translation of integrity constraints formulated in past temporal logic into rules of the Starburst *DBMS. During the compilation the set of constraints is checked for the safe evaluation property. The result of the compilation is a set of* Starburst *SQL statements that define all the necessary rules needed for enforcing the given constraints. There is no need for an additional runtime constraint monitor. When the rules are activated, all updates to the database that violate any of the constraints are automatically rejected (i.e., the corresponding transaction is aborted). In addition to efficient implementation, this approach offers a clean separation of application programs and the integrity checking code.*

1 Introduction

Since the introduction of databases, the notions of data consistency and integrity constraints have been an playing important role in the database application design process. The constraints can usually be divided into two categories: *static* (referring to a static snapshot of the database) and *temporal* (referring to a sequence of such snapshots, ordered in time). Temporal constraints allow imposing restrictions on the transactions over the database like: *"salary of an employee cannot decrease"* or *"once a student drops out, she should not be readmitted"*.

Static constraints have been studied in many papers, e.g., [7]. They can be usually formulated in First Order Logic (FOL). For dealing with temporal constraints the choice of Temporal Logic is a natural solution. To achieve efficient implementation we restrict the language to the *Past* fragment of the Temporal Logic; the constraint checking can then be done without referring to the whole past history of the database as shown in [1].

The work on integrity constraints has been mainly focused on efficient detection of constraint violations. General integrity constraints are difficult to enforce efficiently, thus general-purpose integrity enforcement subsystems are currently present only in a few experimental database systems—commercial DBMS's can usually enforce only the simplest constraints, e.g., constraints on primary and foreign keys [4]. On the other hand there were recent proposals of general constraint management subsystems, especially in [6, 8, 11]. The first paper develops an SQL-based constraint specification language and then shows several techniques of converting such specification into triggers in the *Starburst* system. But the system does not allow fully automatic translation of logic formulas to *Starburst* rules. Also, only static constraints are covered in this approach. The second approach is closer to our work—Temporal Logic is chosen as the constraint specification language, but comparing to our language, the future fragment of TL was chosen there. In that approach quantifiers in logic formulas can have only a very restricted pattern. Also, checking the formulas for the safe evaluation property is solely the user responsibility whereas our method accepts arbitrary Past Temporal formulas as long as they can be safely converted to Relational Algebra; unsafe formulas are rejected by the system. Another approach can be found in [11]. The temporal language chosen in that approach has nonstandard freeze quantifiers and is not related to our work.

Our implementation allows the user to specify the constraints using the language of Past Temporal Logic. This allows the declarative specification of constraints instead of embedding the integrity checks in application programs. The advantages of declarative specification are clear: the designer can concentrate on "what constraints should be enforced" instead of "how to enforce them". It leads also to much more compact and understandable application programs. Moreover the application programs and the integrity constraints specification form independent "modules"—this allows building modular applications, where one (or more) modules specifies the integrity constraints.

We adopt the approach taken in [1] where Past Temporal Logic (PastTL) is used for specifying the temporal integrity constraints. For Past temporal formulas, the truth of a formula in the state n depends only on the finite prefix $D_0, D_1, \ldots, D_n$ of the temporal structure (i.e., the past at time n). Our approach detects violations of the constraints—situations where all the constraints are true in state n but not true in state $n+1$.

The paper is organized as follows. First we describe the syntax and the semantics of the specification lan-

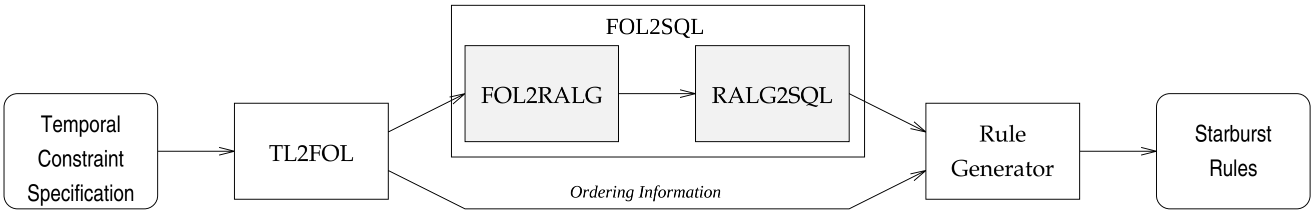

Figure 1: Structure of the system

guage. In the following section the transformation of the constraints to the *Starburst* rule language is shown. Also the possibility of adopting different rule systems is briefly discussed. This conversion uses compilation of First Order Logic formulas to SQL as one of its steps using modification of the approach described in [3]. The paper is concluded with a discussion of the possibilities of extension of this system in the future.

2 Specification Language

This section gives the syntax and semantics of the PastTL language used in our system. The standard notation for temporal formulas is used here [13].

Definition 2.1 (Syntax) The PastTL language is defined as set of formulas of one of the following kinds:

Atoms $R(x_1, \ldots, x_n)$ for all relations R in the database schema; x_i's are constants or variables.

$x \rho y$ where $\rho \in \{=, <, \leq, \neq, \geq, >\}$, and x, y are variables or constants.

$A \vee B$, $A \wedge B$, $\neg A$, $\exists x.A$, A **since** B, $\bullet A$ where A, B are PastTL formulas.

Definition 2.2 (Semantics) The truth value of a closed PastTL formula is defined with respect to an underlying temporal structure $D_0, D_1, \ldots D_n$ (where D_i denotes the state of the database at time i) as follows:

$R(d_1, \ldots, d_n)$, $x \rho y$, $A \vee B$, $A \wedge B$, $\neg A$, and $\exists x.A$ are interpreted as standard FOL formulas in the appropriate state D_i.

$\bullet A$ is true in D_i if $i > 0$ and A is true in D_{i-1}.

A **since** B is true in D_i if B is true in D_k for some $0 \leq k < i$ and for all $k < j \leq i$ A is true in D_j.

The constraints specified as closed formulas of PastTL language must be satisfied in every state D_i of the structure.

The whole specification of constraints consists of two parts. The first part defines the database schema. The second part is a list of constraints. Each constraint is specified as an identifier (name) followed by a closed PastTL formula. In the following text we use the following standard abbreviations: $\forall x.A$ for $\neg\exists x.\neg A$, $\blacklozenge A$ for true **since** A, and $\blacksquare A$ for $\neg\blacklozenge\neg A$.

Example 2.3 Using PastTL the constraint *"salary of of an employee cannot decrease"* is expressed as

$$\neg(\exists .e, s_1, s_2)(\blacklozenge emp(e, s_1) \wedge emp(e, s_2) \wedge s_2 < s_1)$$

and the constraint *"once student drops out, she should not be readmitted"* as

$$\neg(\exists .x)(admitted(x) \wedge \blacklozenge dropout(x))$$

3 Structure of the System

The whole compiler thus consists of several independent modules. The overall structure is shown in Figure 1.

1. The TL2FOL module converts the PastTL formulas to set of first order formulas and also produces information needed for composing of the FOL formulas to *Starburst* rules.
2. The FOL2SQL module are responsible for converting the FOL formulas to relational algebra expressions and SQL. It consists of two submodules: FOL2RALG which converts FOL formula to relational algebra expression and RALG2SQL that generates the final SQL statements. Both are independent on the rest of the system.
3. The Rule Generator module combines the information provided by the TL2FOL module with the SQL statements produced by the FOL2SQL module, and creates *Starburst* rules.

This arrangement allows easy modification in the future: we can easily adopt a different rule system (by changing the rule generator), different DML (by changing the FOL2SQL subsystem), or a different specification language. FOL serves here as a convenient intermediate language.

4 Temporal Information Management

First we show how to generate from a given past temporal logic formula a single FOL formula and a set of updates on auxiliary tables. Also we show an efficient implementation of this mechanism in the *Starburst* rule system.

4.1 From PastTL to FOL

The naive approach to checking (past) temporal constraints is to evaluate the formula w.r.t. the whole prefix of the temporal structure $D_0, \ldots, D_n$ (i.e., the whole DB history). In practice this is not acceptable—we would have to store every past state of the database.

To avoid this problem we first convert the PastTL formula into a set of FOL formulas. Here we use the idea from [1]:

```
create rule rule-name
    [when      {inserted(T_i)|deleted(T_i)|updated(T_i)}*|commit]
    [if        SQL predicate]
    then       SQL action*
    [precedes  rule-name*]
    [follows   rule-name*]
```

During the commit the rule system checks the guard of the rule (`when`); if the guard is true, then evaluates the condition (`if`), and if the condition is satisfied it executes the actions specified in the (`then`) clause. The evaluation order of the rules is controlled with the `precedes` and `follows` clauses. Only the `then` part of the rule is mandatory. Moreover, for all tables, specified in the `when` clause the system provides appropriate *transition tables* `inserted`, `deleted`, `old-updated`, and `new-updated` [5].

Figure 2: Ideal Rule syntax.

1. each temporal subformula α is replaced by the auxiliary atom r_α (introduction of this atom results in creating an auxiliary relation in the resulting code):

$$F \to F[r_\alpha/\alpha] \quad \text{for } \alpha \in \{\bullet A, A \textbf{ since } B\}$$

where the arity of r_α is equal to the number of free variables in α.

2. The auxiliary atom r_α for each temporal subformula α are defined by following table:

α	Auxiliary atom definition
$\bullet A$	$r^0_\alpha :=$ false $r^n_\alpha := A^{n-1}$
A **since** B	$r^0_\alpha :=$ false $r^n_\alpha := r^{n-1}_\alpha \wedge A^n \vee B^{n-1} \wedge A^n$

The superscripts 0, $n-1$, and n denote the appropriate (temporal) state of the database the formula will be evaluated in. Note that always only the last two consecutive states are referenced in this framework—the definition is by induction on the temporal structure, but the evaluation is incremental: each transition to a new state simply computes the 'new' interpretation of the auxiliary atoms from the current interpretation. In the rest of this paper we use φ_{r_α} to denote the formula that defines the 'new' interpretation of the auxiliary atom r_α.

The generated auxiliary atoms are ordered by the following partial ordering.

Definition 4.1 (Ordering) $r_\alpha \prec r_\beta$ *if* α *is subformula of* β*.*

Example 4.2 Using this approach formula

$$\forall x.\blacklozenge(\bullet p(x) \wedge \neg p(x)) \Rightarrow \neg p(x)$$

is converted as follows: we get one top-level FOL formula ($\mathcal{C}$) per PastTL constraint and one inductive definition of an auxiliary atom per temporal subformula of the original formula. In this case we have two temporal subformulas $\alpha_1 = \blacklozenge(\bullet p(x) \wedge \neg p(x))$ and $\alpha_2 = \bullet p(x)$ (both with one free variable). The first order translation then looks as follows:

Name	FO translation
$\mathcal{C}$	$\neg\exists x.(r^n_{\alpha_1}(x) \wedge p^n(x))$
α_1	$r^0_{\alpha_1}(x) :=$ false, $r^n_{\alpha_1}(x) := r^{n-1}_{\alpha_1}(x) \vee (r^{n-1}_{\alpha_2}(x) \wedge \neg p^{n-1}(x))$
α_2	$r^0_{\alpha_2}(x) :=$ false, $r^n_{\alpha_2}(x) := p^{n-1}(x)$

and $r_{\alpha_2} \prec r_{\alpha_1} \prec \mathcal{C}$.

Note that the auxiliary atom definitions can refer both to the n-th and $(n-1)$-th states, while the top-level constraint $\mathcal{C}$ only the n-th state—this allows us to check all the top-level constraints in the initial (0-th) state of the database where all the auxiliary atoms are false (i.e., the corresponding auxiliary relations are empty) by definition.

At the end of this transformation we obtain one FOL formula representing the top-level constraint, several formulas defining inductively the truth value of the auxiliary atoms r_α generated from this PastTL formula, and an ordering $\prec$ of the evaluation (i.e., rematerialization) of the auxiliary relations r_α and the top-level constraint. This information is used when generating the *Starburst* rules in the next phase of the compilation process.

4.2 Materialized view maintenance

In this section we show how the constraints are compiled to a set of *Starburst* rules. For each specified constraint we need an SQL statement that represents the top-level constraint, a list of SQL statements defining the new state of each auxiliary relation (all this is suplied by the FOL2SQL module), and an ordering $\prec$ of auxiliary relations and top-level constraint evaluation.

In the ideal case, the syntax of the used rules will be as shown in Figure 2. In that case we can simply create the rules from the SQL statements as follows. Let $<.>$ be the translation of a FOL formula to SQL, then for each constraint $\mathcal{C}$ we create rules as follows:

Top-level constraint $\mathcal{C}$. There is one rule for each constraint (it is the translation of the top-level FOL formula created in the TL2FOL phase):

```
create rule C
    when        commit
    if          not exists < C >
    then        rollback work
```

Auxiliary atoms. Let φ_{r_α} be the formula defining r_α^n for each auxiliary atom r_α (the whole definition of r_α is $r_\alpha^0 :=$ false, $r_\alpha^n := \varphi_{r_\alpha}$). We create an auxiliary relation (i.e., a table) for each atom r_α:

```
create table rα ( list-of-attributes );
```

This statement also defines the 0-th state—the table is empty (corresponds to $r_\alpha^0 :=$ false). The proper transition to the next state of r_α is achieved by the following rule (here we use the second part of the inductive definition of r_α, namely $r_\alpha^n := \varphi_{r_\alpha}$):

```
create rule rα
    when        commit
    then        update rα by < φ_rα >
    precedes    C, {rβ|rα ≺ rβ}
```

This rule is triggered when a transaction tries to commit; the execution of the rule updates the auxiliary relation r_α. This guarantees the consistency of the auxiliary tables. The proper order of triggering these rules is defined by the subformula ordering $\prec$; this is reflected in the `precedes` clause of the rule body—all subformulas of φ (i.e., the appropriate *Starburst* rules) have to be processed prior to φ's rule.

However the rule language of *Starburst* is not as flexible.

4.3 Problems with SQL

View rematerialization. The "new" state of each auxiliary relation r_α is computed as the SQL equivalent of a FOL formula φ_{r_α}; now we need to *replace* the whole table representing r_α by $< \varphi_{r_\alpha} >$ (r_α may be referenced in φ_{r_α}). This can be done by an analysis of the views to be materialized. We notice that the next state of each of the auxiliary relations r_α is defined by one of the following formulas:

$$
\begin{array}{ll}
(1) & r_\alpha^n := A^{n-1} \\
(2) & r_\alpha^n := r_\alpha^{n-1} \wedge A^n \vee B^{n-1} \wedge A^n
\end{array}
$$

The first case can be solved by following SQL code:

```
delete from rα;
insert into rα values
      ( < A^(n-1) > );
```

To apply this idea to the second case, we need to reformulate the "assignment" statement as the equivalent SQL `insert` and `delete` operations as follows:

```
delete from rα where X not in
      < rα^(n-1) ∧ A^n >;
insert into rα values
      (< B^(n-1) ∧ A^n > minus rα);
```

where X is list of all attributes of r_α. It is easy to see, that these two operations are equivalent to the assignment. The disadvantage of such solution is that we can not compile the right side of the assignment as a single formula, but we must produce two separate SQL statements.

Restrictions on the Rule Language. The second problem is connected with the *Starburst* rule language—the restrictions on the rules are as follows:

1. References to the transition tables are allowed only inside of the rule's body, and only the transition information for the table associated with this rule is available.
2. Each rule is connected with exactly one table; this means that the rule is triggered if and only if the associated table is accessed and moreover, the triggering is defined in the terms of "net" effect of a transaction on the table—`commit`ting itself may not be able to trigger any rule even if the associated table was accessed (i.e., updated with empty "net" outcome). Here we compromise the claim, that the constraints enforcement mechanism is independent on the application: to be on the safe side the application has to update auxiliary `commit` table (introduced just for this purpose) before the actual committing—this can be avoided by allowing a rule being triggered just by committing. Thus each rule must be associated with the `commit` table. If no constraint can be violated by a null transaction, then it is safe to associate the rules with individual tables.

When evaluating an SQL statement $< \varphi >$ we may need to access all the tables referenced in φ and if the evaluation refers to the $n-1$ state we need all the transition tables as well—the $n-1$ state is then restored as

$$
\begin{array}{rcl}
r^{n-1} \equiv r^n & - & (\texttt{new-updated}(r) \cup \texttt{inserted}(r)) \\
 & \cup & (\texttt{old-updated}(r) \cup \texttt{deleted}(r))
\end{array}
$$

This is implemented as follows. For each table T (including the auxiliary r_α tables) we use two additional auxiliary tables T`_add` and T`_sub` (holding the tuples inserted (deleted) from the table T respectively) and a view `old_`T that allows access to the transition information from all rules as follows

```
create view old_T as (
  select * from T
    minus select * from T_sub
    union select * from T_add );
```

Also we need to add rules that keep this information up to date; we use one rule that extracts the transition information, and two rules needed for cleanup of the auxiliary tables after the constraint checking is finished. Evaluation of these rules is synchronized with the remaining rules using the `precedes` and `follows` clauses.

Example 4.3 Now we are in position to finish the example 4.2; the system generates following code:

A rule for the top-level constraint:

$A \sqcap B$	$(x,g,t_1) \in A$	$(x,c,t_1) \in A$	$(x,f,\_) \in A$	$(x,\_,\_) \notin A$
$(x,g,t_2) \in B$	(x,g,t_1)	(x,g,t_2)	(x,g,t_2)	(x,g,t_2)
$(x,c,t_2) \in B$	(x,g,t_1)	$(x,c,t_1 \vee t_2)$	$(x,f,\bot)$	(x,c,t_2)
$(x,f,\_) \in B$	(x,g,t_1)	$(x,f,\bot)$	$(x,f,\bot)$	$(x,f,\bot)$
$(x,\_,\_) \notin B$	(x,g,t_1)	(x,c,t_1)	$(x,f,\bot)$	$N.A.$

$A \sqcup B$	$(x,g,t_1) \in A$	$(x,c,t_1) \in A$	$(x,f,\_) \in A$	$(x,\_,\_) \notin A$
$(x,g,t_2) \in B$	$(x,g,t_1 \vee t_2)$	$(x,c,t_1 \vee t_2)$	$(x,f,\bot)$	(x,c,t_2)
$(x,c,t_2) \in B$	$(x,c,t_1 \vee t_2)$	$(x,c,t_1 \vee t_2)$	$(x,f,\bot)$	(x,c,t_2)
$(x,f,\_) \in B$	$(x,f,\bot)$	$(x,f,\bot)$	$(x,f,\bot)$	$(x,f,\bot)$
$(x,\_,\_) \notin B$	(x,c,t_1)	(x,c,t_1)	$(x,f,\bot)$	$N.A.$

Figure 3: Bottom-up computation of GEN and CON properties

`create rule` $\mathcal{C}$ `on commit,`
  `when updated,`
  `if not` $< \neg\exists x.(r_{\alpha_1}(x) \wedge p(x)) >$
  `then rollback work`

and rules for maintaining the auxiliary tables r_{α_1} and r_{α_2}:

`create rule` α_1 `on commit,`
  `when updated,`
  `then (`
    `insert into` r_{α_1} `values (`
      $< (r_{\alpha_2}^{n-1}(x) \wedge \neg p^{n-1}(x)) >$
        `minus` r_{α_1}
  `) )`
  `precedes` $\mathcal{C}$`;`

The rule for r_{α_2} is similar. (we omitted the rules for accessing the previous state of the database here; note that the rule for r_{α_1} was simplified by discarding the first SQL statement—it is redundant due to the structure of the auxiliary relation defined for $\blacklozenge A$ and is discarded by the system).

4.4 Different Rule Systems

Despite the problems described in the previous section the *Starburst* rule system is extremely well suited for our purposes—it provides set-oriented rules and allows to specify priorities of the rules. In addition the rules are triggered after each transaction. This defines the "flow of time" in our model. All these features are utilized by our implementation.

On the other hand it will be possible to modify the rule generator for conceptually different rule systems. This might be considerably more complex in the case the rule system is tuple oriented (like in [12]): essentially we will have to simulate the set-oriented system explicitly by additional rules for each table (mentioned in the constraints specification) that will maintain the (also explicit) transition tables. In addition we will need rules that are triggered at the beginning and the end of each transaction—these rules will define the "flow of time", i.e., when the next time instant is reached (this will be similar to the `commit` event).

5 Compilation of FOL formulas

The last bit left out now is how to compile the First Order formulas that specify the top-level constraint and the auxiliary relations to Relational Algebra and eventually to SQL. This is done in two steps: first the formula is converted to Relational Algebra Normal Form and then to SQL.

The conversion of FOL formulas to Relational Algebra is based on ideas presented in [3]. All the algorithms were carefully redesigned to take advantage of the structure of the formulas they are working on. This usually leads to more efficient bottom-up execution.

The conversion from Relational Algebra to SQL is usually not paid attention to at all—but what seems to be an easy task turns out not to be as straightforward because of various restrictions in the SQL language. Many workarounds had to be invented to get the system running. The presented version produces standard SQL (i.e., accepted by most commercial DBMS's); the *Starburst* variant of the system can take advantage of extensions to the SQL present in the system. The SQL/92 standard [4] these extensions but unfortunately most available DBMS do not.

5.1 First Order Logic to Relational Algebra Normal Form

The whole conversion consists of three phases: first the FOL formula is checked for safe evaluation property (Join Anomaly Detection) and simultaneously simplified to a normal form (ENF). Then the simplified formula is converted into Relational Algebra Normal Form (RANF). This step removes all Join Anomalies. The last step is conversion from a formula to Relational Algebra (i.e., conversion of logical connectives $(\wedge, \exists, \vee, \neg)$ to operations in Relational Algebra $(\sigma, \times, \pi, \cup, -)$.

Join Anomaly Detection. Let $X = \{x_1, x_2, \ldots\}$ be the set of variable names, $G = \{g, c, f\}$ the set of distinct tags (g stands for GEN, c for CON, and f for not GEN and not CON using terminology from [3]), and L the set of (finite disjunctions of) atoms (base relations). We will define sets $FV(A), FV'(A) \subseteq X \times G \times L$ of *annotated* free variables for each FOL formula A by following inductive definition:

$$\begin{array}{lcl}
FV(A) & = & \{(x,g,A) \mid x \text{ is free in } A\} \\
FV(\neg A) & = & FV'(A) \\
FV(A \vee B) & = & FV(A) \sqcup FV(B) \\
FV(A \wedge B) & = & FV(A) \sqcap FV(B) \\
FV(\exists x.A) & = & FV(A) - (\{x\} \times G \times L) \\
\\
FV'(A) & = & \{(x,f,A) \mid x \text{ is free in } A\} \\
FV'(\neg A) & = & FV(A) \\
FV'(A \vee B) & = & FV'(A) \sqcap FV'(B) \\
FV'(A \wedge B) & = & FV'(A) \sqcup FV'(B) \\
FV'(\exists x.A) & = & FV'(A) - (\{x\} \times G \times L)
\end{array}$$

where the operations $\sqcup, \sqcap$ on the sets of (annotated) free variables are defined in Figure 3. We will use following notation in the rest of the paper: an underscore _ will be the abbreviation for a fresh existentially quantified variable, e.g., $\exists z_1, z_2.(x, z_1, z_2) \in A$ will be abbreviated as $(x, \_, \_) \in A)$. The decision, if a given formula F is evaluable (i.e., it is safe to convert the formula into equivalent Relational Algebra expression) is based on the $FV(F)$ set as follows:

Definition 5.1 *A (FOL) formula F is called* evaluable *if*

1. *$(x, g, \_) \in FV(F)$ for all free variables x in F.*
2. *$(x, c, \_) \in FV(A)$ for all subformulas $\exists x.A$ of F.*

A (FOL) formula F is called allowed *if the second condition is replaced by $(x, g, \_) \in FV(A)$ for all subformulas $\exists x.A$ of F.*

The *evaluable* property is the key for distinguishing formulas, that can be safely converted (see [3] for more detailed discussion). Conversion of formulas which do not meet this criterion may lead into Join Anomalies and unsafe reformulations of the original formula—all such formulas must be rejected.

Conversion to allowed and simplified formula. This step will be done simultaneously with the detection of the *evaluable* property—when traversing the term representing the given formula we will construct the sets FV and FV' together with an equivalent simplified formula defined as follows.

Definition 5.2 *A formula F is* simplified *if*

1. *All conjunctions are converted to a polyadic representation.*
2. *All disjunctions are converted to a polyadic representation.*
3. *The $\neg$ connective may occur only in the root of the formula.*
4. *No disjunction is inside of negation.*
5. *No conjunction of only negative formulas is inside negation.*
6. *No subformulas of the form $\exists x.(A \vee B)$.*
7. *No subformulas of the form $(\exists x.A) \wedge B$.*

To meet the requirements of Definition 5.2 we will convert all conjunctions and disjunctions in the given formula to the following polyadic representation[1]. Note that the separation of positive and negative conjuncts is not required in Definition 5.2, but it will be helpful when enforcing the other requirements of this definition (especially those dealing with negation):

$$\begin{array}{lcr}
A_1 \wedge \ldots \wedge A_n \wedge \neg B_1 \wedge \ldots \wedge \neg B_n & & \\
 & \longrightarrow & \wedge(A_1.\cdots.A_n, B_1.\cdots.B_n) \\
A_1 \vee \ldots \vee A_n & \longrightarrow & \vee(A_1.\cdots.A_n)
\end{array}$$

We will use following algorithm to traverse the input formula and to determine if it satisfies the *evaluable* property (and reject all formulas which do not). Simultaneously we build an equivalent *simplified* formula starting from the atoms and using rules for creating more complex formulas (we will use a "constructor" for each FOL connective—the constructors will be designed to preserve the *simplified* property):

Algorithm 5.3

$$\begin{array}{lll}
\multicolumn{3}{l}{\text{traverse}(F) =} \\
\quad \text{cases}(F) \text{ of} & & \\
\qquad A: & & (\wedge(A,\epsilon), FV(A), FV'(A)) \\
\qquad \neg A: & & \mathbf{N}(\text{traverse}(A)) \\
\qquad A \wedge B: & & \mathbf{A}(\text{traverse}(A),\text{traverse}(B)) \\
\qquad A \vee B: & & \mathbf{O}(\text{traverse}(A),\text{traverse}(B)) \\
\qquad \exists x.A: & & \mathbf{E}(\text{x},\text{traverse}(A))
\end{array}$$

The constructors **N**, **A**, **O**, and **E** will be defined as follows (note that besides transforming the formula to a *simplified* formula we also compute the FV and FV' sets of annotated free variables; this is done simultaneously with the simplification process—the rules for computing the FV and FV' sets match exactly the structure of the formula):

Atom: All atoms are left almost untouched; we simply compute the FV and FV' sets needed for determining the *evaluable* property and convert each atom to a (trivial) polyadic conjunction $\wedge(A, \epsilon)$ that is obviously simplified formula.

Logical connectives: Let A and B be *simplified* formulas. Then we can build conjunction, disjunction, and negation according to rules in Figure 4. The structure of the rules enforces that the resulting formula is simplified.

Existential quantification: Let A be a *simplified* formula. When building the *simplified* formula corresponding to $\exists x.A$ we first check if A is of the form $\vee A_i$, and in this case we perform the following transformation:

$$\begin{array}{lcl}
\mathbf{E}(x, \vee A_i) & \rightarrow & \mathbf{O}(\mathbf{E}(x, A_1), \mathbf{O}(\mathbf{E}(x, A_2), \ldots, \\
 & & \mathbf{O}(\mathbf{E}(x, A_{n-1}), \mathbf{E}(x, A_n)) \ldots))
\end{array}$$

Otherwise we distinguish the following cases (depending on the variable bound by the quantifier):

[1] In the following sections we will use following notation for lists: ϵ for empty list and $A.B$ for concatenation of lists; moreover we will identify single elements with one element lists.

N	$\neg A$	$\vee(L)$	$\wedge(\epsilon, N)$	A(otherwise)
	A	$\wedge(\epsilon, L)$	$\vee(N)$	$\neg A$

O	$\neg A_2$	$\vee(L_2)$	A_2(otherwise)
$\neg A_1$	$\mathbf{N}(\mathbf{A}(A_1, A_2))$	$\mathbf{N}(\mathbf{A}(A_1, \mathbf{N}(\vee(L_2))))$	$\mathbf{N}(\mathbf{A}(A_1, \mathbf{N}(A_2)))$
$\vee(L_1)$	$\mathbf{N}(\mathbf{A}(\mathbf{N}(\vee(L_1), A_2)))$	$\vee(L_1.L_2)$	$\vee(A_2.L_1)$
A_1(otherwise)	$\mathbf{N}(\mathbf{A}(\mathbf{N}(A_1), A_2))$	$\vee(A_1.L_2)$	$\vee(A_1.A_2)$

A	$\exists x_2.A_2$	$\neg A_2$	$\vee(L_2)$	$\wedge(P_2, N_2)$
$\exists x_1.A_1$	$\mathbf{E}(x_1, \mathbf{E}(x_2, \mathbf{A}(A_1, A_2)))$	$\mathbf{E}(x_1, \mathbf{A}(A_1, \neg A_2)))$	$\mathbf{E}(x_1, \mathbf{A}(A_1, \vee(L_2)))$	$\mathbf{E}(x_1, \mathbf{A}(A_1, \wedge(P_2, N_2)))$
$\neg A_1$	$\mathbf{E}(x_2, \mathbf{A}(\neg A_1, A_2))$	$\wedge(\epsilon, A_1.A_2)$	$\wedge(\vee(L_2), A_1)$	$\wedge(P_2, A_1.N_2)$
$\vee(L_1)$	$\mathbf{E}(x_2, \mathbf{A}(\vee(L_1), A_2))$	$\wedge(\vee(L_1), A_2)$	$\wedge(\vee(L_1). \vee(L_2), \epsilon)$	$\wedge(\vee(L_1).P_2, N_2)$
$\wedge(P_1, N_1)$	$\mathbf{E}(x_2, \mathbf{A}(\wedge(P_1, N_1), A_2))$	$\wedge(P_1, A_2.N_1)$	$\wedge(\vee(L_2).P_1, N_1)$	$\wedge(P_1.P_2, N_1.N_2)$

Figure 4: Rules for bottom-up construction of a simplified formula

1. $(x, \_, \_) \notin FV(A)$ This means, that x is not free in A and we can drop the quantifier.
2. $(x, f, \_) \in FV(A)$ Formula is not evaluable—we reject it.
3. $(x, c, G) \in FV(A)$ Formula is evaluable—we perform a transformation to obtain an allowed formula. The transformation was described in [3] and works as follows:

   $$\exists x.A \longrightarrow \exists x.(A \wedge \exists^* x.(\vee G)) \vee R$$

   where G is the third component of the (x, c, G) element in $FV(A)$—this is finite disjunction of atoms such that $(x, g, G) \in FV(\vee G)$, $\exists^* x(\vee G)$ denotes the sequence of existential quantifiers binding all free variables of $\vee G$ but x, and $R = A[G/\text{false}]$. Note that the right side of the transformation $(\exists x.(A \wedge \exists^* x.(\vee G)) \vee R)$ and both G and R have to be built using the **N**, **A**, **O**, and **E** functions in order to preserve the *simplified* property.
4. $(x, g, \_) \in FV(A)$ Formula is allowed—we simply add the quantifier and remove x from $FV(A)$ according the rules for building the FV and FV' sets.

Because each of the constructors **N**, **A**, **O**, and **E** preserves the simplified property of formulas, the constructed formula is also simplified.

Definition 5.4 *A formula F has the* allgen *property (we write allgen(F)) if $(x, g, \_) \in FV(F)$ for all variables x free in F.*

Let assume that F is a simplified formula built from atoms using only $\mathbf{N}(A)$, $\mathbf{A}(A, B)$, $\mathbf{O}(A, B)$, $\mathbf{E}(x, A)$ and let F has the allgen property. Then we know that F is allowed (this is easy to see—both conditions of definition 5.1 are satisfied; moreover all variables bound by $\exists$ satisfy the requirement of the second part of this definition because all $\exists$ quantifiers were "constructed" using **E**).

Conversion to RANF. Using the above transformation, we've been able to separate evaluable formulas (i.e., those formulas, we will be able eventually convert to SQL) from non-evaluable formulas and convert them to allowed formulas. The rules that build the simplified formula guarantee that all free variables in the obtained formula are 'safe'. Before we can convert the formula to relational algebra we need one more conversion: for the proper evaluation of the formula we need to propagate the allgen property to subformulas of the formula as follows (remember that the evaluation in the Relational Algebra is bottom-up).

Definition 5.5 *A subformula A' of A is called* generating *if it is not of the form $\neg B$. A simplified allowed formula A is in* RANF *if all generating subformulas have the allgen property.*

Assume that F is an allowed and simplified formula. Then it can be converted to an equivalent formula F' in RANF by following rules that are applied on F recursively according to the top-level connective:

atom: In RANF by definition.

$\exists x.A$: $(x, g, \_) \in FV(A)$ by definition of the **E** constructor and all other variables free in $FV(A)$ are also free in $FV(\exists x.A)$ and thus allgen(A) holds by our assumption.

$\vee L$: Let allgen($\vee L$) holds. Let x be free in $\vee(L)$. Assume there is $L_i \in L$ such that $(x, g, \_) \notin FV(L_i)$. Then $(x, g, \_) \notin FV(L_i \vee T)$ by definition of the FV set for any formula T, contradiction. Thus allgen($\vee L$) implies allgen(L_i) for all $L_i \in L$.

$\wedge(P, N)$: We know allgen($\wedge(P, N)$). But this does not guarantee that all members of P and N share this property. We will use the following transformations to propagate the *allgen* property to all elements of P and also all elements of N as follows:

Positive conjuncts: By definition of the *simplified* formula we know that the elements of P are either atoms or unions, and by definition of FV the *allgen* property holds for each atom. Let $\vee(L)$ be an element of P such that allgen($\vee L$) does not hold for variables $\{x_1, \ldots, x_k\}$. Because (by assumption) the whole conjunction has the *allgen* property, there must exist $P' \subset P$ such that $(x_i, g, \_) \in FV(\wedge(P', \epsilon))$ for all $0 < i \le k$. We can distribute P' into the $\vee L$ as follows (note that the distribution law guarantees equivalence of the original and the resulting formulas):

$$\wedge(P, N) \longrightarrow \wedge(P''.L', N)$$

where L is the "offending" union, $P'' := P - (L.P')$, and $L' = \vee(L_i \wedge P'), L_i \in L$ (all these formulas are constructed using **E**, **N**, **O**, and **A** to preserve the *simplified* structure of the formula). This transformation is repeated until all elements of P have the *allgen* property[2] (note that we may end with only one element in P. If moreover $N = \epsilon$, the polyadic conjunction $\wedge(P, \epsilon)$ is converted to single atom P).

Negative conjuncts: Here each element N_i of N represents subformula of the form $\neg N_i$; to meet the RANF requirements we need to enforce allgen(N_i). Using $A \wedge \neg B \equiv A \wedge \neg(A \wedge B)$, we find for each N_i subset $P' \subseteq P$ such that allgen($P' \wedge N_i$) (note that P' may be empty). Then following transformation will propagate the *allgen* property to all N_i's:

$$\wedge(P, N) \longrightarrow \wedge(P, N[(P' \wedge N_i)/N_i])$$

In each case we reduce the height of the term by one. On the subformulas we can apply the rules recursively.

Because the term representing the given formula has finite depth, this rules give us an algorithm that correctly converts given simplified allowed formula into RANF formula.

Conversion to Relational Algebra. Now we have prepared everything for the final step in the conversion of a logic formula to Relational Algebra. Because we've already converted the formula to RANF, this step will make no changes to the 'structure' of the formula; the only goal here is to convert:

1. nested existential quantifiers to projection (π),
2. disjunction into set union ($\cup$)[3], and
3. conjunction into product ($\times$), selection (σ_E), and (generalized) set difference ($-$). This will be the most difficult part: we have also to convert the variable bindings into set of equalities between the columns of the cartesian product and constants.

Remember, that we've been using the polyadic conjunction and except the root of the formula all negations are hidden inside the conjunction. Thus we don't have any special rule for converting negation.

5.2 Relational Algebra to SQL

Now we are prepared to convert relational algebra expressions from the previous step into actual SQL queries. This conversion due to the limitations of SQL language is not as straightforward as is generally believed. We need to carry out following transformations:

1. Standard SQL (SQL/89) can not handle nested occurrences of negation and union inside `SELECT` clause. We need to remove these subterms by assigning a (virtual) view to each Relational Algebra subterm of the form

$$\begin{aligned}
&\pi(\sigma_F(L)) \rightarrow \\
&\qquad \pi(\sigma_F(L[W/\textstyle\bigcup A_i])) \\
&\qquad \text{where } W := \textstyle\bigcup A_i \text{ for each } \textstyle\bigcup A_i \in L \\
&\pi(\sigma_F(L) - \pi(\sigma_{F'}(M) - \pi(\sigma_{F''}(N)))) \rightarrow \\
&\qquad \pi(\sigma_F(L) - W) \\
&\qquad \text{where } W := \pi(\sigma_{F'}(M) - \pi(\sigma_{F''}(N)))
\end{aligned}$$

where L, M, and N are finite lists of relations to be 'joined'[4].

2. Closed FOL formulas also can not be handled immediately because SQL does not allow empty projection (or 0-ary relation(s))—the solution will use a single attribute auxiliary relation `true(t)` which will always contain one tuple `( true )`. This relation will be 'joined' with the rest of the `FROM` clause and it's (only) attribute will be `SELECT`ed, e.g.,

$$\exists x.r(x) \xrightarrow{\text{FOL2RALG}} \pi_\emptyset(r) \xrightarrow{\text{RALG2SQL}} \begin{array}{l}\texttt{SELECT t}\\ \texttt{FROM true,r}\end{array}$$

3. When using formulas of PastTL, we need to access the current and the last states of the database—this is remembered during all the conversions and all the atoms are annotated by the state they need to be evaluated in. Here we must convert this information to proper names of tables and views (in our case T and `old_`T).

6 Future work

The front-end of the system. (the TL2FOL module): We can introduce other TL constructs in the PastTL language and try to capture bigger class of temporal constraints (for example real-time constraints described in [2]).

We can refine the translation from TL to FOL: in several cases we have found examples where the amount of information stored in the auxiliary relations can be reduced.

The FOL2RALG and RALG2SQL modules: Here we can write specific routines for query optimization. Moreover the transformation to RALG is not unique and optimization of this transformation may help to reduce the size of the final formula (but note that we can't avoid the exponential explosion in general).

The Rule generator: We can optimize the process of rematerialization of the auxiliary relations as shown in [9, 10]. This pass can be easily added on top of the existing system.

[2] Yes, this might be a problem: in the worst case we end with exponential expansion of the formula. Like $(A \vee B) \wedge (C \vee D) \rightarrow (A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (B \wedge D)$.

[3] We need to watch the attribute order in all subformulas of the union—this is done by the projection operator (π) that is used for permuting the attributes when necessary.

[4] This is not a problem in *Starburst* where extensions to SQL allow to handle the nested subqueries directly. Also systems conforming with the SQL/92 standard can handle nested queries directly.

We can change the rule system (mainly by removing some syntactic restrictions on the rules) that is better suited for handling converted logic formulas, especially conjunctions. This may lead to elimination of unnecessary rule invocations like rules for extracting the previous state of the database—the information is indeed present in the system, but the current syntax of the rules does not allow direct access of all the needed transition tables.

Also introduction of the `commit` event (now simulated by an auxiliary table that has to be updated prior to the commit of each transaction) will help make the constraint module totally independent on the application.

We can adopt another rule system (e.g., tuple-oriented). This may allow porting of this system to commercial DBMS, that supports at least minimal trigger system in which we can simulate the *Starburst* rule system.

7 Conclusion

We have seen that the language of Past Temporal Logic is suitable for specifying constraints over temporal database in a straightforward and declarative way. We have also shown that such specification can be efficiently translated to *Starburst* rules which guarantee enforcement of the constraints on the underlying database. There is no need for an additional run time constraint monitor. Moreover, during the translation process all the formulas are checked for safe evaluation property and formulas that do not meet this requirement are rejected by the system. The system is fully operational.

The general philosophy of our system fits well with the *Starburst* active DBMS developed in IBM Almaden which supports set-oriented triggers that are invoked at the end of the transactions rather than after each single update (as, e.g., in Postgres [12]). We also utilized the possibility of ordering the evaluation of the rules in an essential way.

Our approach expands the range of the active database system applications without requiring changes to the DBMS itself.

8 Acknowledgments

We are very grateful to Jennifer Widom for encouraging us to use Starburst and supplying the system. This research was partially supported by NSF grant IRI-9110581.

References

[1] Chomicki J.. History-less Checking of Dynamic Integrity Constraints. *IEEE International Conference on Data Engineering*, Phoenix AZ, February 1992.

[2] Chomicki J.. Real-Time Integrity Constraints. In *Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1992.

[3] Van Gelder A. and Topor R.W. Safety and Translation of Relational Calculus Queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.

[4] International Organization for Standardization (ISO). Database Language SQL. Document ISO/IEC 9075:1992. (Also ANSI X3.135-1992).

[5] Widom J., Cochrane R.J., and Lindsay B.G. Implementing Set-oriented Production Rules as an Extension to Starburst. In *Proc. International Conference on VLDB*, 1991.

[6] S. Ceri, J. Widom. Deriving Production Rules for Constraint Maintenance. In *Proc. International Conference on VLDB*, 1990.

[7] H. Gallaire, J. Minker, and J. Nicolas. Logic and Databases: a Deductive Approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[8] Gertz M., Lipeck U.W. Deriving Integrity Maintaining Triggers from Transition Graphs. In *International Conference on Data Engineering*, 1993.

[9] Blakeley J.A., Larson P.A., and Tompa F.W. Efficiently Updating Materialized Views. In *Proc. ACM SIGMOD*, pp.61–71, Washington D.C., June 1986.

[10] Gupta A., Mumick I.S., Subrahmanian V.S. Maintaining views incrementally. In *Proc. ACM SIGMOD* pp.157–167, Washington D.C., May 1993.

[11] Sistla A.P., Wolfson O. Temporal Triggers in Active Databases. *IEEE Transactions on Knowledge and Data Engineering*, (to appear) 1993.

[12] R. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching, and Views in Data Base Systems. In *ACM SIGMOD Int. Conf. on Management of Data*, 1991.

[13] Manna Z. and Pnueli A. The temporal logic of reactive and concurrent systems. *Springer-Verlag*, 1992.