Implementing Temporal Integrity Constraints Using an Active DBMS

Jan Chomicki and David Toman

Abstract—**The paper proposes a general architecture for implementing temporal integrity constraints by compiling them into a set of active DBMS rules. The modularity of the design allows easy adaptation to different environments. Both differences in the specification languages and in the target rule systems can be easily accommodated. The advantages of this architecture are demonstrated on a particular temporal constraint compiler. This compiler allows automatic translation of integrity constraints formulated in Past Temporal Logic into rules of an active DBMS (in the current version of the compiler two active DBMS are supported: *Starburst* and INGRES). During the compilation the set of constraints is checked for the safe evaluation property. The result is a set of SQL statements that includes all the necessary rules needed for enforcing the original constraints. The rules are optimized to reduce the space overhead introduced by the integrity checking mechanism. 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 straightforward implementation, this approach offers a clean separation of application programs and the integrity checking code.**

Keywords—**Temporal databases, integrity constraints, dynamic constraints, constraint checking.**

I. Introduction

SINCE the introduction of databases, the notions of data consistency and integrity constraints have been playing an important role in the database application design process. Integrity constraints can usually be divided into two categories: *static* (referring to a static snapshot of the database) and *temporal* (referring to a sequence of 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"*.

We propose a general architecture for a temporal integrity constraint compiler based on compilation of temporal specification (i.e., a set of temporal constraints) into a set of First Order Logic (FOL) definitions. FOL serves here as a very convenient intermediate language. The FOL definitions are then converted into a set of active rules that enforce the specified constraints without a need for an additional run-time constraint monitor. This arrangement allows easy modification of the system to incorporate different query/constraint languages in a uniform way. In addition, we show how the architecture can accommodate a number of different optimization techniques. Our implementation instantiates the general architecture by fixing the constraint language to be Past Temporal Logic (PastTL). The current implementation produces code for *Starburst* [6] and INGRES [20] active DBMS.

The implementation allows the user to specify the constraints declaratively 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 pursue further the approach taken in [1], [3] where PastTL was proposed as a language for specifying temporal integrity constraints. For PastTL formulas, the truth of a formula in the state n depends only on the finite *history* $D_0, D_1, \ldots, D_n$ of the temporal database (i.e., the past at time n). Our approach detects violations of the constraints, namely the situations where all the constraints are true in the state $n-1$ but not true in the state n.

We also develop space-optimization techniques for the proposed architecture. The optimizations are introduced at two different levels: the first optimization deals with the specification language (PastTL), the second with the intermediate language (FOL). When performing the optimizations we need to keep in mind that

- we are dealing with *temporal* constraints. This means that the optimizations need to explicitly handle the progression of time.
- the final goal of the compilation is to produce a set of active rules. In particular this means that all the formulas have to be converted to an appropriate DML (e.g., SQL). The optimization techniques should preserve *convertibility* to the chosen language (cf., sections V and VI).

So far, the work on integrity constraints has been mainly focused on efficient detection of constraint violations. General integrity constraints are included in [5] but 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 [5].

Static constraints have been studied in many papers, e.g., [9]. They can be usually formulated in FOL. For dealing with temporal constraints the choice of Temporal Logic seems to be a natural solution. In the implementation we

Manuscript received Oct. 3, 1994.
J. Chomicki and D. Toman are currently with the Department of Computing and Information Science, Kansas State University, Manhattan, KS 66506. E-mail: {chomicki,david}@cis.ksu.edu.
IEEECS Log Number K95048

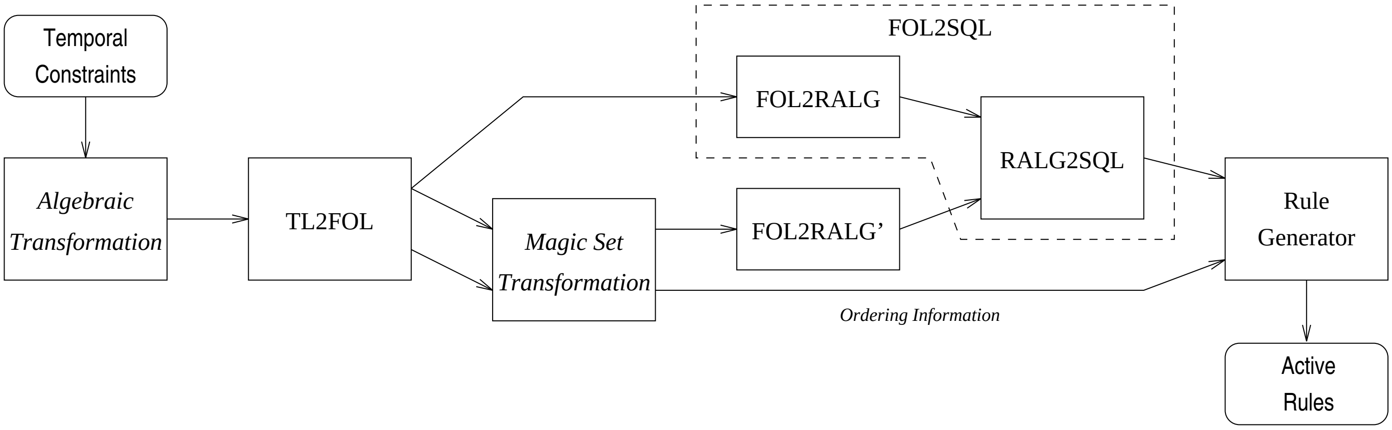

Fig. 1. Structure of the system

restricted the language to the *Past* fragment of the Temporal Logic (Temporal Logic with temporal operators referring solely to the past); the constraint checking then can be done by using a space-efficient encoding of the database history [1].

Utilization of active rules has an important advantage compared with other methods of integrity enforcement: there is no need for a standalone (temporal) integrity monitor. The use of a separate monitor would not improve efficiency as it would have to evaluate essentially the same set of queries against the database as the rules do. However, it would have to duplicate the transition datastructures maintained already by the active DBMS.

There were several other recent proposals of general constraint management subsystems, especially in [7], [10], [13]. The first paper [7] 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 [10] 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 PastTL 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 [13]. The temporal language chosen in that approach uses nonstandard freeze quantifiers instead of first-order ones. The expressive power of the constraint language depends on the underlying query language. For a detailed comparison of our approach with related work see [3].

The paper is organized as follows. In section II we describe the overall structure of the system. In section III we introduce the syntax and semantics of the specification language. In section IV the transformation of the constraints to the rule language of an active DBMS is shown. The general schema for generating active rules is instantiated for use with *Starburst* and INGRES active DBMS. The suitability of the respective rule languages is briefly discussed. The compiler uses an FOL to SQL translator (described in section V) as one of its steps. We use a modification of the approach in [4]. Section VI develops space-saving optimizations to minimize the overhead connected with the constraint enforcement mechanism. The paper is concluded with a discussion of the possibilities of future extensions of our system.

II. System Architecture

The general architecture consists of the following basic building blocks [19]:

- Temporal Constraint language to FOL compiler
- FOL to DML compiler
- Rule generator.

Our system extends this architecture by providing a number of optimization modules. The overall structure of the system is shown in Figure 1:

- In the first step an algebraic transformation of the original PastTL formula is performed. Unless additional information is supplied, only conservative transformations are performed.
- The TL2FOL module converts the PastTL formulas to a set of FOL formulas and also produces information needed for translating the FOL formulas to active rules (partial ordering information).
- A variant of the Magic Set transformation [18] is applied to each of the FOL formulas produced by the TL2FOL pass. This transformation may also modify the ordering information (i.e., some incomparable elements may become comparable).
- The FOL2SQL module is responsible for converting FOL to relational algebra and SQL. It consists of two submodules: FOL2RALG which converts a FOL formula to a Relational Algebra Normal Form (RANF) [4] expression. This expression is then combined with the magic conditions produced by the Magic Set Transformation that were also converted to Relational Algebra Normal Form using a modified FOL2RALG algorithm (cf. Algorithm 36). The output of these

two modules is then put back together. In the end the RALG2SQL module generates the final SQL statements.

- The Rule Generator module combines the information provided by the TL2FOL module with the SQL statements produced by the FOL2SQL module, and creates the active rules.

This arrangement allows easy modification in the future: we can easily adopt a different rule system (by changing the rule generator—this feature has been demonstrated by the implementation that supports two conceptually different rule systems), different DML (by changing the RALG2SQL subsystem and possibly the rule generator), or a different specification language. Also introduction of additional optimization techniques can be easily embedded in the current system: FOL serves here as a convenient intermediate language.

III. Specification Language

This section gives a brief overview of the syntax and semantics of the PastTL language used in our system. The standard notation for temporal formulas is used here; for full description of temporal logic see [8], [15].

Definition 1 (Syntax) The PastTL language is defined as the smallest set of formulas built using following rules:

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 (Semantics) The truth value of a closed PastTL formula is defined with respect to an underlying *history* of the database $D_0, D_1, \ldots, D_n$ (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 must be satisfied in every state D_i of the history of the database.

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 3: Using PastTL the constraint *"salary of of an employee cannot decrease"* is expressed as

$$c_1 : \neg(\exists.e, s_1, s_2)(\blacklozenge emp(e, s_1) \wedge emp(e, s_2) \wedge s_2 < s_1)$$

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

$$c_2 : \neg(\exists.x)(admitted(x) \wedge \blacklozenge dropout(x))$$

The truth of PastTL constraints is determined with respect to the history of the database: the constraint c_2 is true at the current moment (now) if there is no element x, such that x is in the relation *admitted* now (i.e., in the current state of the database) and also x was in the relation *dropout* in the past (i.e., in some of the past states of the database). The constraint specification consists of a definition of the database schema (including the types of all attributes) followed by a list of constraints. Each constraint is specified by an identifier (name) followed by a closed PastTL formula.

IV. Temporal Information Management

First we show how to convert a given PastTL formula to a single FOL formula and a set of inductively defined auxiliary atoms (these definitions are later converted to auxiliary relations and active rules). In section VI we introduce optimization techniques that help to cut down the (space) overhead introduced in the TL2FOL pass of the compiler.

A. From PastTL to FOL

The naive approach to checking (past) temporal constraints is to evaluate the formula with respect to the whole history $D_0, \ldots, D_n$. In practice this is not acceptable because every past state of the database would have to be stored. To avoid this problem we provide a space-efficient encoding of the history [1], [3] using a number of auxiliary atoms implemented as materialized views. To obtain the atom definitions we convert every PastTL formula into a set of FOL formulas.

Definition 4 (TL2FOL) Let F be a formula of PastTL.

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

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

where the arity of r_α is equal to the number of free variables in α, and the type of each attribute is the same as the type of the corresponding variable in α.

2. the auxiliary atoms r_α for each temporal subformula α are defined by following table:

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

The superscripts 0, $n-1$, and n denote the appropriate (temporal) state of the database the formula will be evaluated in.

3. The remaining formula (i.e., after the substitutions have been made) defines the *top-level* FOL translation of the original PastTL constraint.

The superscripts n and $n-1$ are symbolic references to database states, not specific numbers. Note that always only the last two consecutive states are referenced in this framework—the definition is by induction on the the length of the database history, 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 5 (Ordering) $r_\alpha \prec r_\beta$ if α is subformula of β. The top-level constraint is always the top element in this ordering.

Example 6: Using this approach the constraint *no employee is hired after she left*, expressed in PastTL as

$$\mathcal{C} : \neg(\exists x)(emp(x) \wedge \blacklozenge(\neg emp(x) \wedge \bullet emp(x)))$$

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 example we have two temporal subformulas: $\alpha_1 = \blacklozenge(\neg emp(x) \wedge \bullet emp(x))$ and $\alpha_2 = \bullet emp(x)$ (both with one free variable). The first order translation then looks as follows:

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

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

Note that the auxiliary atom definitions can refer both to the states n and $(n-1)$, while the top-level constraint $\mathcal{C}$ refers only the state n. This allows us to check all the top-level constraints in every state, in particular 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 the original 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 the active rules are generated in the next phase of the compilation process.

B. Maintenance of Auxiliary Relations

The conversion of a temporal constraint to FOL generates a set of auxiliary atom definitions. These auxiliary atoms are represented by auxiliary relations in the resulting code. The contents of the auxiliary relations has to be kept up to date in agreement with Definition 4. This is achieved automatically using the active DBMS rule system. For each specified constraint the SQL statement that represents the top-level constraint, the list of SQL statements defining the next state of each auxiliary relation (all these are supplied one by one by the FOL2SQL module), and the subformula ordering $\prec$ (defined above) are used as the input to the rule generator. The Rule Generator composes all the pieces into a set of active DBMS rules.

In the ideal case, the syntax of the used rules is 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 (including all optional optimization steps), then for each constraint $\mathcal{C}$ we create the 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 <C>
    then        rollback work
```

The rule is triggered when transaction attempts to commit (i.e., *before* the actual committing). This is necessary in the case of temporal constraints as even null transaction can possibly violate the constraints (note that this cannot happen, when only static constraints are used).

Example 7: Consider the constraint $\neg\exists x.(p(x) \wedge \bullet p(x))$. This constraint is violated when an arbitrary element x is in the relation p in two consecutive states. Clearly, if p is not empty, a null transaction will violate this constraint.

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

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

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

```
create rule r_α
    when        commit
    then        update r_α by <φ_{r_α}>
    precedes    C, {r_β | r_α ≺ r_β}
```

This rule is also 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's body. The rules corresponding to all subformulas of φ have to be processed prior to φ's rule.

When evaluating an SQL statement $<\varphi>$ we may need to access all the tables referenced in φ. Some of the atoms may refer to the $n-1$-st state of the database. The previous state of the database (i.e., the old contents of all the relations) is not stored in the database—this would require to maintain two copies of almost the same data. Instead, *transition* tables are used to restore the previous state of the database. Note that the transition information (for relation r) has to be maintained by the DBMS anyway in order to allow aborting of a transaction. The access to this information should be provided by the (system maintained) transition tables `inserted`(r), `deleted`(r), `old-updated`(r), and `new-updated`(r). The state $n-1$ of a relation r is restored

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

Fig. 2. Ideal Rule syntax.
During the commit the rule system checks the guard of the rule (`when`); if the guard is true, then evaluates the condition (`if`). 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` [6].

as

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

Example 8: Now we can finish the example 6. The system generates the following code: A rule for the top-level constraint:

$\texttt{create rule } \mathcal{C}$
$\quad\texttt{when commit,}$
$\quad\texttt{if not} < \neg\exists x.(r_{\alpha_1}(x) \wedge emp(x)) >$
$\quad\texttt{then rollback work}$

and two rules to maintain the auxiliary tables r_{α_1} and r_{α_2}:

$\texttt{create rule } \alpha_1$
$\quad\texttt{when commit,}$
$\quad\texttt{then } \mathit{update}\ r_{\alpha_1}\ \texttt{by}$
$\qquad < r_{\alpha_1}^{n-1} \vee (r_{\alpha_2}^{n-1}(x) \wedge \neg emp^{n-1}(x)) >$
$\quad\texttt{precedes } \mathcal{C};$

$\texttt{create rule } \alpha_2$
$\quad\texttt{when commit,}$
$\quad\texttt{then } \mathit{update}\ r_{\alpha_2}\ \texttt{by} < emp^{n-1}(x) >$
$\quad\texttt{precedes } \mathcal{C}, \alpha_1;$

C. Problems with SQL

The actual DML and rule languages usually do not allow to express the rules needed to enforce the constraints directly. The implementors of the Rule Generator are faced with two main obstacles:

- problems with the data manipulation language, and
- problems with the rule system used.

The overhead introduced by the restrictions of the DMLs and the rule systems languages is summarized in Figure 3.

C.1 View Rematerialization

The new state of each auxiliary relation r_α is computed by the SQL equivalent of a FOL formula φ_{r_α}. But we need to *replace* the whole contents of the table representing r_α by $< \varphi_{r_\alpha} >$ (where 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:

$\quad\texttt{delete from } r_\alpha;$
$\quad\texttt{insert into } r_\alpha < A^{n-1} >;$

To apply this idea to the second case, we need to reformulate the assignment statement (2) as the equivalent SQL `insert` and `delete` operations as follows:

$\quad\texttt{delete from } r_\alpha\ \texttt{where } X\ \texttt{not in} < r_\alpha^{n-1} \wedge A^n >;$
$\quad\texttt{insert into } r_\alpha < B^{n-1} \wedge A^n >\ \texttt{minus } r_\alpha^n;$

where X is list of all attributes of r_α. It is easy to see, that these two operations are equivalent to the original assignment. The disadvantage of this 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.

Example 9: Continuing our example, the *update* operations on the auxiliary tables r_{α_1} and r_{α_2} are replaced by

$\quad\texttt{insert into } r_{\alpha_1}$
$\qquad < r_{\alpha_2}^{n-1}(x) \wedge \neg emp^{n-1}(x) >\ \texttt{minus } r_{\alpha_1}^n;$

and

$\quad\texttt{delete from } r_{\alpha_2};$
$\quad\texttt{insert into } r_{\alpha_2} < emp^{n-1}(x) >;$

respectively. Note, that the first update is simplified (by removing the `delete` operation) due the structure of the definition generated from the subformula rooted by the connective ◆.

C.2 Restrictions on Rule Languages

The syntax and semantics of active rule systems varies greatly among active DBMS. Our implementation currently supports two major approaches to rule systems:

Set-oriented rule systems. A representative of such a system is the *Starburst* active DBMS. The rule language of *Starburst* is very close to the ideal rule syntax (cf. Figure 2). However, the restrictions on the rule syntax makes direct application of the rule templates from the previous section not possible. Especially:

- 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 particular rule is available.
- 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 slightly compromise the claim that the constraints

Number of	Ideal Syntax	*Starburst*	INGRES
Rules	C+T	C+4T+3B	C+4T+4B
Auxiliary Tables	T	3T+2B+1	3T+2B+1
Virtual Views	0	0	U
Database Procedures	0	0	C+3T+3B+1

C–number of constraints,
T–number of temporal connectives,
B–number of base tables, and
U–number of embedded disjunctions.

Fig. 3. Size of code generated for various systems.

enforcement mechanism is independent of 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 attempt to commit. 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.

The workaround for these problems is as follows. For each table T (including the auxiliary tables r_α) we use two additional auxiliary *transition* tables T`_add` and T`_sub` holding the tuples inserted (deleted) from the table T respectively:

```
create table T_add ( list-of-attributes );
create table T_sub ( list-of-attributes );
```

and a view `old_`T that allows access to the previous state $n-1$ of the table T from all rules as follows

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

Also we need to add rules that keep this information up to date during the constraint checking phase. 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.

```
create rule old_T on T
  when (inserted,deleted,updated),
  then (
    insert into T_sub (
      select * from deleted()
            union distinct
      select * from old-updated() ),
    insert into T_add (
      select * from inserted()
            union distinct
      select * from new-updated() ),
  precedes (list of all rules using old_T) );

create rule del_add_T on T_add
  when inserted,
  then delete from T_add, follows C;
create rule del_sub_T on T_sub
  when inserted,
  then delete from T_sub, follows C;
```

These rules and tables are used merely to extract the transition information at the end of the transaction—during the execution of the transaction the transition information is managed by the DBMS itself (and this should be fixed by simple changes to the syntax of the rule language). Note also that the tables T`_add` and T`_sub` are cleared at the end of the constraint checking phase, thus they are always empty when a transaction ultimately commits. This allows to store them in a temporary storage which may improve the efficiency of the system. If any of the constraints is violated, the whole transaction is aborted and the transition tables are emptied by the system automatically.

Example 10: For the constraint from Example 6, similarly to the ideal case, one rule is created to enforce the top-level constraint:

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

Two rules are created to maintain the auxiliary relations. Note, that all the rules are connected with an additional auxiliary table `commit`. This is needed in the case of null transactions (see Example 7).

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

`create rule` α_2 `on commit,`
  `when inserted,`
  `then (`
    `delete from` r_{α_2}
    `insert into` $r_{\alpha_2} < emp^{n-1}(x) >$`)`
  `precedes` $\mathcal{C}, \alpha_1$`;`

In addition the system generates rules that maintain the contents of the auxiliary transition tables.

Tuple-oriented rule systems. In the case of tuple-oriented systems like INGRES the access to the previous state of the database relations is provided using the transition information similarly to the *Starburst* case. However, the transition information has to be maintained by additional rules in auxiliary *transition* tables[1]. Also, *Starburst*'s capability to order individual rules has to be compiled into an explicit sequence of INGRES statements. The code produced by the rule generator looks as follows:

1. For every table T (both database and auxiliary relations) a pair of additional auxiliary *transition* tables for

[1] This looks very similar to the T`_add` and T`_sub` tables introduced in the case of *Starburst* rules. However, in the *Starburst* case these tables are used just to overcome syntactic restrictions of the rule system as the transition information is maintained by the DBMS itself. In the case of INGRES, these tables are essential to mimick *Starburst*'s internal capabilities.

storing the transition information is needed:

```
create table T_add ( attributes );
create table T_sub ( attributes );
```

The contents of the tables is maintained using INGRES rules triggered by insertion, deletion, and update operations on the table T. These rules maintain the contents of the transition relations *during* the duration of the whole transaction. This effectively duplicates the transition information managed by the DBMS itself for the case of aborted transactions. However, assuming only small changes to the base relations by every transaction, the size of the duplicated information is not significant (when compared to the size of the whole database).

In the INGRES rule language rules have to be connected with a single database procedure, so the rules come in pairs with the corresponding database procedures:

```
create procedure Tinserted( attributes ) =
  insert into T_add values ( attributes );
create procedure Tdeleted( attributes ) =
  insert into T_sub values ( attributes );

create rule Tins after insert on T
  execute procedure Tinserted( new.attribs );
create rule Tdel after delete on T
  execute procedure Tdeleted( old.attribs );
create rule Tupdateins after update on T
  execute procedure Tinserted( new.attribs );
create rule Tupdatedel after update on T
  execute procedure Tdeleted( old.attribs );
```

At the end of the constraint checking the transition tables are cleaned up using following rule:

```
create procedure Tcleanup = begin
  delete from T_add;
  delete from T_sub;
end;

create rule Tdoclean after delete on commit
  execute procedure Tcleanup;
```

2. The actual code produced by the translation of the temporal constraint (i.e., all definitions of the auxiliary relations together with the top-level constraint) is used in the body of a single main rule (so we avoid the need for synchronization of the rules):

```
create procedure C_main =
  declare n integer;
  begin
    update r_{α_n} by < φ_{r_{α_n}} >;
        ⋮
    update r_{α_1} by < φ_{r_{α_1}} >;
    select :n = count(*)
      from true where exists ( < C > );
    if n = 0 then
      raise error 1 '';
    endif;
  end;

create rule C_go after insert on commit
  execute procedure C_main;
```

where $r_{\alpha_n} \prec \ldots \prec r_{\alpha_1} \prec \mathcal{C}$. Thus the *update* operations are executed in the body of the rule in the correct order.

3. All the rules generated from the constraints are synchronized with the *cleanup* rules using the following master procedure using a additional auxiliary table `commit`:

```
create table commit (i integer);
create procedure docommit = begin
  insert into commit values (1);
  if iierrornumber = 0 then
    delete from commit;
    commit;
  else
    rollback;
  endif;
end;
```

The application then has to use the `execute docommit` statement in the place of the `commit` statement.

Example 11: The constraint from Example 6 produces following INGRES procedure/rule pair:

```
create procedure C_main =
  declare n integer;
  begin
    delete from r_{α_2};
    insert into r_{α_2} < emp^{n-1}(x) >;
    insert into r_{α_1}
      < r_{α_2}^{n-1}(x) ∧ ¬emp^{n-1}(x) > minus r_{α_1};
    select :n = count(*)
      from true where exists
        (< ¬∃x.(r_{α_1}(x) ∧ emp(x)) >);
    if n = 0 then
      raise error 1 '';
    endif;
  end;

create rule C_go after insert on commit
  execute procedure C_main;
```

Again, this code comes with all the rules that maintain the contents of the auxiliary transition tables. Note that there are no separate rules for the individual auxiliary relations generated for the temporal subformulas of the original constraint.

V. Compilation of FOL formulas

The next issue that needs to be addressed is how to compile the FOL 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 the ideas presented in [4]. All the definitions were carefully converted to functions that take advantage of the structure of the formulas they are working on. This reduces the number of passes needed to traverse the input formula to two and leads to a 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

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

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

Fig. 4. Bottom-up computation of GEN and CON properties

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 the extensions to SQL present in the system. The SQL/92 standard [5] contains these extensions but unfortunately most available DBMS (like INGRES) do not.

A. First Order Logic to Relational Algebra Normal Form

The whole conversion consists of two phases: first the FOL formula is checked for safe evaluation property (Join Anomaly Detection) and simultaneously simplified to a normal form (ENF). All formulas that pass this phase are guaranteed to have equivalent safe reformulation to Relational Algebra. In the second step the simplified formula is converted into Relational Algebra Normal Form (RANF). This step removes all Join Anomalies.

A.1 Join Anomaly Detection

Definition 12: 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 [4]), and L the set of (finite disjunctions of) atoms (base relations). We 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:

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

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

where the operations $\sqcup, \sqcap$ on the sets of (annotated) free variables are defined in Figure 4.

We use the 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 an equivalent Relational Algebra expression) is based on the $FV(F)$ set as follows:

Definition 13: 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 [4] for a more detailed discussion). Conversion of formulas which do not meet this criterion may lead to Join Anomalies and unsafe reformulations of the original formula, and all such formulas must be rejected.

Our specification language for temporal constraints is PastTL; the evaluable property is extended to PastTL formulas as follows:

Definition 14: A PastTL formula is evaluable if and only if its FOL translation is evaluable (i.e., the top-level constraint is evaluable and all the definitions of the auxiliary views are evaluable as well).

A.2 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 15: A formula F is simplified if

1. Conjunctions are in a polyadic representation[2].
2. Disjunctions are in a polyadic representation.
3. The $\neg$ connective may occur only in the root of the formula or inside a conjunction.
4. No disjunction is inside a negation.
5. No conjunction of only negative formulas is inside a 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 15 we will convert all conjunctions and disjunctions in the given formula to

[2]Finite nesting of a binary connective is replaced by a single *polyadic* connective operating on a list of arguments.

the following polyadic representation[3]:

$$A_1 \wedge \ldots \wedge A_n \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \longrightarrow \wedge(A_1.\cdots.A_n, B_1.\cdots.B_n)$$
$$A_1 \vee \ldots \vee A_n \longrightarrow \vee(A_1.\cdots.A_n)$$

Note that the separation of positive and negative conjuncts is not required in Definition 15, but it is helpful when enforcing the other requirements of this definition (especially those dealing with negation).

We use the 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 16: Let F be a FOL formula.

traverse(F) =
cases(F) of

atom A	$\wedge(A, \epsilon)$
$\neg A$	$\mathbf{N}$(traverse(A))
$A \wedge B$	$\mathbf{A}$(traverse(A),traverse(B))
$A \vee B$	$\mathbf{O}$(traverse(A),traverse(B))
$\exists x.A$	$\mathbf{E}$(x,traverse(A))

The constructors **N**, **A**, and **O** are defined in Figure 5. The definition of the constructor **E** is more complex:

Definition 17: 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:

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

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

1. $(x, \_, \_) \notin FV(A)$: x is not free in A and we can drop the quantifier.
2. $(x, f, \_) \in FV(A)$: A is not evaluable and we reject it.
3. $(x, c, G) \in FV(A)$: A is evaluable. We perform a transformation to obtain an equivalent allowed formula. The transformation was described in [4] and is defined 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 $R = A[G/\text{false}]$ (all atoms that occur in G are replaced by *false* in A). 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. $(x, g, \_) \in FV(A)$: A is allowed and we simply add the quantifier and remove x from $FV(A)$ according the rules for building the FV and FV' sets.

Note that besides transforming the formula to a *simplified* formula we also compute the FV and FV' sets of annotated free variables as in Figure 4. This is done simultaneously with the simplification process. The rules for computing the FV and FV' sets match exactly the structure of the formula. The FV set is used to detect the evaluable property (see Definitions 17 and 19).

Lemma 18: Algorithm 16 converts the input formula to an equivalent simplified such that $(x, g, \_) \in FV(A)$ for all subformulas $\exists x.A$ of F (or rejects it if the formula is not evaluable).

Proof: By induction on the structure of the input formula. Each of the constructors **N**, **A**, **O**, and **E** preserves the simplified property of formulas, the constructed formula is also simplified.

Algorithm 16 together with Definition 17 gives a recipe for the conversion of a FOL formula to a simplified formula In order to convert the simplified formula to Relational Algebra we need to perform one more transformation.

Definition 19: A formula F has the *allgen* property (we write allgen(F)) if $(x, g, \_) \in FV(F)$ for all variables x free in F.

This property is easy to check. Moreover

Theorem 20: Let F be a simplified formula built from atoms using only **N**, **A**, **O**, and **E** constructors. If F has the allgen property, then F is allowed.

Proof: This is easy to see: both conditions of definition 13 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**).

Using the above transformation, we have 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 range restricted.

A.3 Conversion to RANF

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 (Definition 19) to the subformulas of the formula as follows (remember that the evaluation in the Relational Algebra is bottom-up).

Definition 21: 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.

Lemma 22: Let F be an allowed and simplified formula. Then it can be converted to an equivalent formula F' in RANF.

Proof: By following inductive rules that are applied on F recursively according to the top-level connective:

atom. In RANF by definition.

[3] In the following sections we will use following notation for lists: ϵ for empty list and $A.B$ for concatenation of lists. 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	$\exists x_2.A_2$	$\neg A_2$	$\vee(L_2)$	$\wedge(P_2, N_2)$
$\exists x_1.A_1$	$\mathbf{E}(x_1, \mathbf{E}(x_2, \mathbf{A}(A_1, A_2)))$	$\mathbf{E}(x_1, \mathbf{A}(A_1, \neg A_2)))$	$\mathbf{E}(x_1, \mathbf{A}(A_1, \vee(L_2)))$	$\mathbf{E}(x_1, \mathbf{A}(A_1, \wedge(P_2, N_2)))$
$\neg A_1$	$\mathbf{E}(x_2, \mathbf{A}(\neg A_1, A_2))$	$\wedge(\epsilon, A_1.A_2)$	$\wedge(\vee(L_2), A_1)$	$\wedge(P_2, A_1.N_2)$
$\vee(L_1)$	$\mathbf{E}(x_2, \mathbf{A}(\vee(L_1), A_2))$	$\wedge(\vee(L_1), A_2)$	$\wedge(\vee(L_1).\vee(L_2), \epsilon)$	$\wedge(\vee(L_1).P_2, N_2)$
$\wedge(P_1, N_1)$	$\mathbf{E}(x_2, \mathbf{A}(\wedge(P_1, N_1), A_2))$	$\wedge(P_1, A_2.N_1)$	$\wedge(\vee(L_2).P_1, N_1)$	$\wedge(P_1.P_2, N_1.N_2)$

Fig. 5. Rules for bottom-up construction of a simplified formula
The tables define rewrite rules that allow construction of more complex simplified formulas. The first line (column) defines the pattern(s) (i.e., the left side of the rules) and the body of the table defines the results of the rewriting (see Algorithm 16).

$\exists x.A.$ $(x, g, \_) \in FV(A)$ 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$(\vee L)$ hold. 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 $0 < i \leq k$. 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[4] (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 the 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 depth of the term by one. On the subformulas we can apply the rules recursively.
Because the term representing the given simplified formula has finite depth, the proof immediately gives us an algorithm that correctly converts given simplified allowed formula into a RANF formula.

B. Relational Algebra Normal Form to SQL

Now all the transformations of the original formula at the level of logic are finished. In the following section the conversion from RANF to SQL statements is described. This transformation is done again in two steps. First the logic formula is translated to Relational Algebra operations and then the Relational Algebra term is rewritten to SQL.

B.1 Conversion to Relational Algebra

The conversion to RANF prepared everything for the next step–the conversion of a logic formula to Relational Algebra. 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 ($\cup$)[5], 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 of the root of the formula all negations are hidden inside conjunctions. Thus we don't have any special rule for converting negation.

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

[5] 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.

B.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 generally believed. We need to carry out the following transformations:

1. Standard SQL (SQL/89) can not handle nested occurrences of negation and union inside `SELECT` clauses[6]. We need to remove these subterms by assigning a (virtual) view to each Relational Algebra subterm of the form

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

where L, M, and N are finite lists of relations to be joined.

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 its (only) attribute will be `SELECT`ed, e.g.,

$$\exists x.r(x) \overset{\text{FOL2RALG}}{\longrightarrow} \pi_\emptyset(r) \overset{\text{RALG2SQL}}{\longrightarrow} \texttt{SELECT t FROM true,r}$$

3. When using formulas of PastTL, we need to access the current and the last states of the database. This information is kept 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).

VI. Optimization

The translation described so far produces a set of active rules sufficient to enforce a given set of temporal constraints. However, the auxiliary relations introduced during the translation of PastTL formula to FOL (and eventually to active rules) increase the amount of data stored in the database. In this section general optimization techniques that allow to cut down this overhead are developed. Note that these optimizations can be used because the auxiliary relations are not used for direct querying of the database. Their contents is irrelevant for evaluating standard first order queries (over the current state of the database) but on the other hand the information stored is not sufficient for answering (general) temporal queries. Thus any restrictions on their contents or changes in their definitions do not affect the database user (we can think about these tables as being invisible to the user).

Also, we deal only with space optimization. The optimization of the running time of the queries is left to the query optimizer of the underlying DBMS.

[6] *Starburst* and systems conforming to the SQL/92 standard can handle nested queries directly.

We explore two methods of limiting the amount of data stored in auxiliary relations:

Context-based optimization. We can limit the number of tuples stored in an auxiliary relation by the analysis of the context(s) the relation is used in. This technique is similar to the Magic Set transformation [16], [17], [18]. However in the context of temporal formulas we need to be more careful than in the time less case. The information passed sideways must agree with the *flow of time* because we can't predict the future. The advantage of this method is its universal applicability (i.e., we don't need any statistical information about the database).

Algebraic optimization. The other option is to use techniques based on algebraic transformations. These can be used before the PastTL formula is converted to a set of FOL formulas (i.e., before the auxiliary relations are introduced). In this case laws that allow to move the temporal connectives over the first order ones are used. The transformations allow to move the places where the auxiliary relations are introduced up and down in the (parse tree of the) original PastTL formula. Careful choice of the transformations can significantly reduce the amount of tuples stored in these relations. Unfortunately the choice often requires obtaining additional information about the database (like average sizes of the relations involved, average size of joins, etc.).

When using the optimization techniques we need to keep in mind that the resulting formulas need to be converted to Relational Algebra (and later to SQL). Thus we need to be careful not to introduce constructs not expressible in SQL. Especially all the formulas have to be recursion-free and to have safe reformulations [4].

A. Magic Set Transformation

The main idea of this transformation is based on the following observation: the auxiliary relations r_α are used only in finitely many known contexts (all of which can be easily determined by traversing the original formula). The analysis of the contexts allows to restrict the contents of the auxiliary relations to relevant tuples only. This technique is very general: it is applicable to all auxiliary relations[7] in an application (i.e., it is not limited to auxiliary relations introduced by the TL2FOL conversion).

Note that the classical application of the Magic Set transformation to a set of Horn rules [18] works in a completely different setting: the magic sets are used to pass (partial) information about the intended outcome of the bottom-up evaluation of the rules. In our case the setting is different: we do not have any information about the future states of the temporal database (except for constant relations). But we can show how to exploit the definitions of the auxiliary relations to create restricting conditions in a similar fashion.

Also we use a different mechanism for computing the sideways information passing strategy (SIPS) than [17]

[7] I.e., relations defined by a formula from other relations.

where the strategy is based on adornments of literals (atoms). The adornments then provide conservative guidelines for computing the SIPS so that the resulting rules are evaluable (range-restricted). We take a more optimistic approach for computing the SIPS: In the first step we compute all the information that can be passed sideways in the form of a formula (without being concerned about its properties) and then we *approximate* the obtained formula to regain the properties necessary for converting it to SQL. This method also allows to pass arbitrarily complex conditions (e.g., conditions that constrain more than one variable, like $x < y$) sideways.

Example 23: Let $\varphi(x) := x > 10 \wedge (A(x) \vee r_\alpha(x))$ be a formula with a reference to the auxiliary relation r_α. Then clearly only those tuples in r_α that satisfy the condition $x > 10$ can affect the outcome of evaluation of $\varphi(x)$. Thus (assuming r_α was used only in this single place) when r_α is being rematerialized we can restrict the tuples stored to those for which the condition $x > 10$ holds. Note that the condition $x > 10$ is time-invariant.

The restricting condition induced by a context in which r_α is used is called a *magic condition* for r_α. In the previous example the context was "$x > 10 \wedge (A(x) \vee [\cdot])$" and the induced magic condition was $x > 10$. This condition has to be computed for each context r_α is used in. The overall magic condition for the auxiliary relation is then determined as the disjunction (i.e., union) of all such conditions.

Definition 24 (Magic Condition) Let φ be a formula and ψ be a subformula of φ. Let ψ' be another formula such that $FV(\psi') \subseteq FV(\psi)$. If $\varphi \equiv \varphi[\psi/\psi \wedge \psi']$[8] then ψ' is called a *magic condition* for ψ (in φ).

To compute the magic conditions for each occurrence of r_α the following algorithm is used. Note that we need to determine the magic conditions for the *leaves* of the (tree corresponding to the) formula; whereas the magic information for the root of the formula can be passed to the algorithm from the upper level (the magic condition for the root of the top-level constraint is *true*).

Notation. Let r_α be an auxiliary relation and φ_{r_α} the formula defining r_α. We denote m_r_α the magic condition for an (single) occurrence of r_α in some formula ψ. We denote $m\_\varphi_{r_\alpha}$ the magic condition restricting the contents of r_α (i.e., the *overall* magic condition with respect to all occurrences of r_α).

The following algorithm collects all the information that can be passed sideways to a particular leaf of a given formula. Note that together with the formula the algorithm takes another argument—the magic condition for the root of this formula (i.e., if a restricting condition has already been computed for the *root* of a given formula it can be passed down towards the leaves of the formula).

Algorithm 25 (Magic) Let ϕ be a FOL formula.

magic(ϕ,M) = case ϕ of

$A \wedge B$	magic(A,$M \wedge B$), magic(B,$M \wedge A$)
$A \vee B$	magic(A,M), magic(B,M)
$\exists x.A$	magic(A,M)
$\neg A$	magic(A,M)
atom r_α	$m\_r_\alpha := \exists x_1, \ldots, x_k.M$ where $\{x_1, \ldots, x_k\} = FV(M) - FV(r_\alpha)$.
atom A	skip

The correctness of this algorithm is proved using the following lemma and theorem:

Lemma 26: Let $B[p]$ be a formula having atom p as a leaf. Let A be another formula. Then $A \wedge B[p] \equiv A \wedge B[p \wedge A]$

P r o o f: By induction on the structure of B.

Theorem 27: Let φ be a formula and m_φ a magic condition for φ. Then magic(φ, m_φ) computes a magic condition for all auxiliary relations in the leaves of the input formula φ.

P r o o f: It is sufficient to prove $\varphi \wedge m\_\varphi \equiv \varphi[r_\alpha/r_\alpha \wedge m\_r_\alpha] \wedge m\_\varphi$. The proof is by induction on the structure of φ and the corresponding structure of m_r_α.

While computing the magic conditions for all contexts of r_α we need to be careful. Algorithm 25 computes magic condition for every leaf of the input formula. However:

- The information passed sideways has to *agree with the flow of time*, i.e., no information can be passed from the future.
- The information has to be *relevant*—the magic condition has to actually restrict the contents of the auxiliary relation.
- The transformation has to preserve the *evaluable* property. This is necessary because the transformed formula has to be converted to Relational Algebra and eventually to SQL.

Thus we often need to replace the computed magic condition by a weaker formula that satisfies the above restrictions. The following theorem shows the soundness of the replacement:

Theorem 28: Let $\varphi[\psi]$ be a formula with subformula ψ. Let ψ, ψ', and ψ'' be formulas with the same set of free variables. Then

$$(\varphi[\psi] \equiv \varphi[\psi'] \wedge \psi \supset \psi'' \supset \psi') \Rightarrow \varphi[\psi] \equiv \varphi[\psi''] \equiv \varphi[\psi']$$

P r o o f: By induction on the structure of φ we prove $\psi \supset \psi'' \supset \psi'$ implies $\varphi[\psi] \supset \varphi[\psi''] \supset \varphi[\psi']$ or $\varphi[\psi'] \supset \varphi[\psi''] \supset \varphi[\psi]$. Now we use the assumption $\varphi[\psi] \equiv \varphi[\psi']$ to obtain the desired result.

Theorem 28 is applied to magic conditions as follows:

Corollary 29 (Approximation) Let φ be a formula and ψ be a subformula of φ. Then

$$\varphi \equiv \varphi[\psi/\psi \wedge \psi'] \wedge \psi' \supset \psi'' \Rightarrow \varphi \equiv \varphi[\psi/\psi \wedge \psi'']$$

To find an appropriate weaker magic condition the following definition is used:

Definition 30: Let φ be a FOL formula and let ψ be subformula of φ. ψ is called *positive (negative)* in φ if it is in the scope of even (odd) number of negations in φ. We define a *approximation* of ψ in φ as

$$\text{Approx}(\psi) = \begin{cases} \text{true} & \text{if } \psi \text{ is positive in } \varphi \\ \text{false} & \text{if } \psi \text{ is negative in } \varphi \end{cases}$$

[8] $\equiv$ denotes semantic equivalence of the formulas.

Note that any safe (i.e., time-independent) approximation of ψ can be used here as long as $\psi \supset \text{Approx}(\psi)$ ($\text{Approx}(\psi) \supset \psi$) for ψ positive (negative) in φ.

Lemma 31: Let φ be a FOL formula and ψ a subformula of φ. Let ψ_1 and ψ_2 be two formulas (with the same set of free variables as ψ) such that $\psi_1 \supset \psi_2$. Then

$$\begin{array}{lcl} \psi \text{ is positive in } \varphi & \Rightarrow & \varphi[\psi_1/\psi] \supset \varphi[\psi_2/\psi] \\ \psi \text{ is negative in } \varphi & \Rightarrow & \varphi[\psi_2/\psi] \supset \varphi[\psi_1/\psi] \end{array}$$

P r o o f: By induction on the structure of φ (again, there is only one occurrence of ψ in φ).

Obviously in all cases $\varphi \supset \varphi[\psi/\text{Approx}(\psi)]$. This fact together with Corollary 29 is used to modify the original magic formula in such a way, that it is evaluable, relevant, and agrees with the flow of time:

Definition 32: Let r_α^n be a occurrence of r_α in a formula φ. A magic condition m_r_α for r_α *agrees with the flow of time* if for all leaves p^m of m_r_α, $m \leq n$ where n and m are the states of the database the respective atoms are evaluated in.

This means that all leaves must be at least as old as the occurrence of r_α. Unfortunately the agreement of sideways information passing with the flow of time is not guaranteed.

Example 33: Let $\varphi(x) := A(x)^n \wedge r_\alpha^{n-1}(x)$. The magic condition computed by Algorithm 25 for $r_\alpha^{n-1}(x)$ is $A(x)^n$. When $r_\alpha(x)$ is rematerialized at time $n-1$ the contents of $A(x)$ at time n is not known (a reference to the future). Thus the magic condition needs to be safely approximated in order to account for all possible values of $A(x)$ at time n. In this case the only safe approximation is *true*.

In the first step the agreement with the flow of time is achieved:

Definition 34 (Flow of Time) Let m_r_α be the magic condition computed by Algorithm 25 for the definition of r_α. Let $\text{Approx}(m\_r_\alpha)$ be the condition obtained by replacing all leaves in m_r_α using the table in Figure 6 (rows: the occurrences of r_α, columns: the leaves of m_r_α).

Note the difference between the first and second rows of the table: not only references to the future are replaced by their approximations but also the superscripts denoting the time of evaluation of a particular leaf are modified to match with the occurrence of r_α the magic condition is associated with. Thus if the occurrence of r_α is in the state $n-1$ (the first row of the table) then the leaves labeled with $n-1$ are *current* wirth respect to this occurrence of r_α (so the label is changed to n) and the leaves labeled n are *in the future*. Similarly, because of sequential rematerialization of the auxiliary relations we need to modify the leaves in the case of $\alpha \preceq \beta$ (i.e., r_β is rematerialized *after* r_α and thus the actual state of r_β has index $n-1$).

Clearly $\text{Approx}(m\_r_\alpha)$ agrees with the flow of time with respect to r_α (the proof is by induction on structure of m_r_α). Also $\text{Approx}(m\_r_\alpha)$ is a magic condition for r_α: this follows immediately from Lemma 31 and Corollary 29.

The order of rematerialization of the auxiliary relations is important: no information can be passed from a new state of an auxiliary relation that has not been updated yet. Also when two auxiliary relations are incomparable (in $\prec$) we need to pick which of them is going to be rematerialized first (i.e., which is smaller in $\prec$). The rule system of the active database evaluates the rules sequentially. Thus we can pick any linear ordering of the rematerialization of the auxiliary relations as long as it contains $\prec$.

This solves the problem with the flow of time, but the obtained magic condition may still be not relevant. Note that the magic condition for r_α in φ produced by the Algorithm 25 has always the form $\bigwedge_{i \in I} \psi_i$ where ψ_i are subformulas of the original formula φ. This leads to following definition:

Definition 35: Let m_r_α be the magic condition for r_α computed by Algorithm 25. Let m_r_α be of the form $\bigwedge_{i \in I} \psi_i$where ψ_i are subformulas of φ. A conjunct ψ_i is called *relevant to* r_α if

1. $FV(\psi_i) \cap FV(r_\alpha) \neq \emptyset$ or
2. $FV(\psi_i) \cap FV(\psi_j) \neq \emptyset$ for some ψ_j relevant to r_α.

Let R be the set of relevant φ_i's. We define

$$\text{Relevant}(m\_r_\alpha) := \bigwedge_{\psi_i \in R} \psi_i \wedge \bigwedge_{\psi_i \notin R} \text{Approx}(\psi_i)$$

Again $\text{Relevant}(m\_r_\alpha)$ is a magic condition for r_α; we used Lemma 31 and Corollary 29 to approximate subformulas of the original magic condition.

Finally we have to make sure that the magic condition can be evaluated (i.e., that it has the evaluable property). The situation is very similar to detecting the evaluable property for the constraints themselves. The only difference arises when a non-evaluable (sub-)formula is detected: in the former case the formula is rejected, but in the case of magic conditions we can always pick a weaker condition that is evaluable (note that the weakest condition—true—is always evaluable).

Algorithm 36: Let Eval be the function defined by Algorithm 16, where the $\mathbf{E}$ constructor is redefined as follows: Let A be a *simplified* formula (see Definition 15). If A is of the form $\vee A_i$, then:

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

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

1. $(x, \_, \_) \notin FV(A)$: x is not free in A and we can drop the quantifier.
2. $(x, f, \_) \in FV(A)$: formula is not evaluable and we replace it by $\text{Approx}(A)$. Now $(x, \_, \_) \notin FV(A)$ by definition of Approx, and we can proceed as in case 1. Note that if A is of the form $\wedge A_i$ we can approximate only those conjuncts that contain x among their free variables.
3. $(x, c, G) \in FV(A)$: formula is evaluable, we perform a transformation to obtain an allowed formula (this is the same as in the original definition of $\mathbf{E}$).
4. $(x, g, \_) \in FV(A)$: formula is allowed and we simply add the quantifier and remove x from $FV(A)$ according the rules for building the FV and FV' sets.

Similarly to the cases of Approx and Relevant we can show that $\text{Eval}(m\_r_\alpha)$ is a magic condition for r_α that is evaluable.

	A^{n-1}	A^n	$r_\beta^{n-1}(\alpha \preceq \beta)$	$r_\beta^n(\alpha \preceq \beta)$	$r_\beta^{n-1}(\beta \prec \alpha)$	$r_\beta^n(\beta \prec \alpha)$
r_α^{n-1}	A^n	Approx(A)	Approx(r_β)	Approx(r_β)	r_β^n	Approx(r_β)
r_α^n	A^{n-1}	A^n	r_β^n	Approx(r_β)	r_β^{n-1}	r_β^n

Fig. 6. Dependencies on the Flow of the Time.

Note that the if the input formula agrees with the flow of time then the resulting formula agrees with the flow of time as well. Both the Relevant and the Eval transformations obviously preserve this property.

After application of these three steps an evaluable magic condition that agrees with the flow of time is obtained. The process is repeated for each occurrence of the same auxiliary relation. The individual magic conditions are then glued together.

Considering all the occurrences of r_α, we need to distinguish two cases in order to apply Algorithm 25:

1. r_α occurs as a leaf of a parent formula (i.e., in the top-level constraint $\mathcal{C}$ or in φ_{r_β} for $\alpha \prec \beta$).
2. r_α occurs as a leaf of its own definition.

Note that r_α cannot occur as a leaf of any φ_{r_β} for $\beta \prec \alpha$ by Definition 5.

In the first case the solution is easy: for r_α being a leaf of φ_{r_β} we just compute the (partial) magic condition using the Algorithm 25, i.e., magic($\varphi_{r_\beta}, m\_\varphi_{r_\beta}$) (or magic($\mathcal{C}$, true) in the case of top-level constraint). The second case is more intricate: when trying to compute the magic condition for r_α it is not clear what condition M should be used in magic(φ_{r_α}, M) because the magic condition for r_α has not been computed yet. Moreover we can not simply use the magic conditions derived from the parent formulas only, as illustrated by following example:

Example 37: Assume constraint $\exists x.(A(x) \wedge \blacklozenge B(x))$. The first-order translation consists of the top-level constraint $\exists x.(A(x) \wedge r(x))$ and the (only) auxiliary atom is defined by

$$r^0(x) := \text{false};\;\; r^n(x) := B^{n-1}(x) \vee r^{n-1}(x)$$

(we have dropped the subscript $\blacklozenge B(x)$ of r). Assume that we try to restrict the contents of this auxiliary relation by $A(x)$. Let p be the restriction of r:

$$p^0(x) := \text{false};\;\; p^n(x) := (B^{n-1}(x) \vee p^{n-1}(x)) \wedge A^t(x)$$

At the time of the rematerialization of $p^n(x)$ at time t of A can be at most n (so we don't look into the future). The following table shows an example why this transformation is not acceptable:

	0	1	...	$n-1$	n
A	{}	{}	...	{}	{1}
B	{1}	{}	...	{}	{}
r	{}	{1}	...	{1}	{1}
p	{}	{}	...	{}	{}

The result of constraint evaluation in state n is different for r (true—correct result) and for p (false—incorrect result). On the other hand if A was a time-independent relation (like $x > 0$) the transformation would be valid.

We can simply use the constant *true* for M, but the previous example shows that a stronger condition can be used in general (also, in our setting the choice of *true* would produce *true* as the magic condition for this occurrence of r_α).

Definition 38 (Time-invariant Condition) Let Inv(φ) be the formula φ where all time-dependent leaves[9] are replaced by their approximations.

Lemma 39: Let r_α be an auxiliary relation. Let m'_r_α be $\bigvee_{i \in I} m^i\_r_\alpha$ where I is the set of all occurrences of r_α in formulas φ_{r_β} for $\alpha \prec \beta$. Then magic(φ_{r_α}, Inv(m'_r_α)) computes a magic condition for r_α in φ_{r_α}.

P r o o f: Inv(m'_r_α) is time-invariant condition for r_α with respect to all occurrences in parent formulas. Using Lemma 26 we can prove the claim by induction on time (the base case is trivial: $r_\alpha^0 :=$ false by Definition 4. The induction step follows directly from Lemma 26).

Example 40: Consider the constraint

$$\neg \exists x, y. y \leq 0 \wedge \blacklozenge emp(x, y)$$

that expresses *no employee has had salary less or equal to $0*. In this case the top-level constraint would be $\neg \exists x, y. r(x, y) \wedge y \leq 0$ and the auxiliary relation r would be defined by

$$r^0(x, y) := \text{false}, r^n(x, y) := emp^{n-1}(x, y) \vee r^{n-1}(x, y)$$

Clearly the condition $y \leq 0$ can be pushed into the body of r's definition.

When the magic conditions for all occurrences of an auxiliary relation r_α have been defined, then the overall magic condition is defined as follows.

Definition 41 (Overall Magic Condition) Let r_α be an auxiliary relation defined by φ_{r_α}. Let I be the set of all occurrences of r_α[10]. Then the overall magic condition $m\_\varphi_{r_\alpha}$ defined as

$$m\_\varphi_{r_\alpha} := \bigvee_{i \in I} \text{Eval}(\text{Relevant}(\text{Approx}(m^i\_r_\alpha)))$$

Clearly $m\_\varphi_{r_\alpha}$ is a magic condition for every occurrence of r_α. This follows from the observation that

$$\text{Eval}(\text{Approx}(m^i\_r_\alpha)) \supset \bigvee_{i \in I} \text{Eval}(\text{Approx}(m^i\_r_\alpha))$$

and Corollary 29. Note that the evaluability of the overall magic condition is guaranteed because we can distribute

[9] A leaf of a formula (i.e., an atom) is time-dependent if its extension can change with time.

[10] In the current version there are at most two references to the same auxiliary relation. But in general we can handle arbitrary number of references.

Rewrite Rule	Condition
$\exists x.\bullet\varphi \rightarrow \bullet\exists x.\varphi$	
$\bullet\varphi_1 \vee \bullet\varphi_2 \rightarrow \bullet(\varphi_1 \vee \varphi_2)$	
$\bullet\varphi_1 \wedge \bullet\varphi_2 \rightarrow \bullet(\varphi_1 \wedge \varphi_2)$	for $\lvert \mathrm{Ext}(\varphi_1 \wedge \varphi_2)\rvert < \lvert \mathrm{Ext}(\varphi_1)\rvert + \lvert \mathrm{Ext}(\varphi_2)\rvert$
$\exists x.(\varphi_1 \textbf{ since } \varphi_2) \rightarrow \varphi_1 \textbf{ since } \exists x.\varphi_2$	for $x \notin FV(\varphi_1)$

Fig. 7. Algebraic Transformations.

$\mathrm{Ext}(\varphi)$ denotes the set of all tuples that satisfy φ.

the relation r_α into the disjunction m_r_α. Thus for all variables free in r_α, $(x, g, \_) \in FV(r_\alpha \wedge m\_r_\alpha)$ (see Definition 13) and no other free variables appear in this formula. A successful conversion to SQL also requires that the magic transformation produces a set of non-recursive view definitions (this can be proved using the techniques presented in [17]).

The magic conditions m_r_α (for each φ_{r_α}) are used just before the auxiliary relations are rematerialized to restrict their contents.

B. Algebraic Transformations

The Magic Set transformation reduces the amount of data stored in auxiliary relations. Because the transformation is applied *after* the original PastTL formula has been converted to a set of first order formulas, it can not influence the choice of auxiliary relations introduced during this conversion. In the simplest case (that was considered in Section IV) the auxiliary relations are introduced in the places of subformulas of the original formula rooted by a temporal connective. This is not necessary: the original formula can be transformed to an equivalent formula in which the temporal connectives are in more suitable places.

Example 42: Let $\varphi(y) := \exists x.\blacklozenge P(x, y)$. Then the auxiliary relation $r_{\blacklozenge P(x,y)}$ has arity 2. On the other hand the auxiliary relation $r'_{\blacklozenge \exists x.P(x,y)}$ induced by the temporal subformula of the formula $\varphi'(y) := \blacklozenge\exists x.P(x, y)$ has arity only 1. Moreover it is clear that $r' \subseteq \pi_x(r)$ (thus the amount of stored data is reduced). Clearly φ and φ' are equivalent. The transformations are summarized in Figure 7.

VII. Future work

The generality and modular structure of the proposed compiler architecture allows easy adaptation to different environments. The modifications are usually confined to a single module of the compiler.

Other Query Languages. We can introduce other constructs in the constraint specification language to capture bigger class of temporal constraints (for example real-time constraints [2]), repeated activities (periodic sets), etc.

New Optimization Techniques. So far we have considered only space-saving optimization techniques. We can introduce optimizations that speed up the evaluation of the given constraints (note that space saving techniques also help towards efficient execution because we deal with smaller amounts of data). In this area we have several options:

1. We can write specialized routines for query optimization. The transformation to RANF 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).
2. We can optimize the process of rematerialization of the auxiliary relations as shown in [11], [12]. This pass can be easily added to the existing system.
3. The partial ordering of the rematerialization of the auxiliary relations is linearized during the compilation. Some linearization may yield better magic optimization than other ones. Similarly to the RANF case, we can study the impact of different linearizations on the resulting code.

Different Rule Systems. Despite the problems described in section IV-C the *Starburst* rule system is extremely well suited for our purposes: It provides set-oriented rules and allows to specify rule priorities. 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. We have also shown that the architecture can also utilize tuple-based rule systems [20]: Essentially we have to simulate the set-oriented system explicitly by additional rules for each table (mentioned in the constraints specification) that maintain the (also explicit) transition tables.

Performance Analysis. Measuring performance of a constraint enforcement system is a quite complex problem. Clearly the cost of evaluating the constraints is the same as running a set of queries at the end of every transaction (time overhead) and maintaining the auxiliary relations (space overhead). However, the time/space overhead depends directly on the transactions run against the database. Both the length of the transactions and the amount of changes to the database play a significant role in the analysis. Moreover a benchmarking method has to choose a *reference* system the results are compared with. It is not clear, what the reference system should be—there are several candidates: a database *without* any constraint enforcement, constraint enforcement with respect to the whole history of the database, etc. We plan to develop benchmarking method suitable for comparing the overhead and performance of temporal constraint enforcement systems.

VIII. Conclusion

We have seen that the language of Past Temporal Logic is suitable for specifying constraints over temporal databases in a straightforward and declarative way. We have also shown that this specification can be translated to active DBMS 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. Also several optimization steps are performed to cut down the overhead connected with the constraint checking.

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

IX. Acknowledgments

We are very grateful to Jennifer Widom for encouraging us to use Starburst and for supplying the system. Thanks also go to Inderpal Singh Mumick for discussions and for sending us his Ph.D. thesis. This research was partially supported by NSF grant IRI-9110581.

An early version of some of the results in this paper appeared in [19]. INGRES is a trademark of the Ingres Corporation.

References

[1] J. Chomicki. History-less Checking of Dynamic Integrity Constraints. *IEEE International Conference on Data Engineering*, Phoenix AZ, February 1992.
[2] J. Chomicki. Real-Time Integrity Constraints. In *Proc. Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1992.
[3] J. Chomicki. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Transactions on Database Systems*, to appear.
[4] A. Van Gelder and R. W. Topor. Safety and Translation of Relational Calculus Queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.
[5] International Organization for Standardization (ISO). Database Language SQL. Document ISO/IEC 9075:1992. (Also ANSI X3.135-1992).
[6] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing Set-oriented Production Rules as an Extension to Starburst. In *Proc. International Conference on VLDB*, 1991.
[7] S. Ceri, J. Widom. Deriving Production Rules for Constraint Maintenance. In *Proc. International Conference on VLDB*, 1990.
[8] D. M. Gabbay, I. Hodkinson, and M. Reynolds. Temporal Logic. Mathematical Foundations and Computational Aspects. Vol. 1. Oxford Logic Guides, Oxford University Press, 1994.
[9] H. Gallaire, J. Minker, and J. Nicolas. Logic and Databases: a Deductive Approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.
[10] M. Gertz, U. W. Lipeck. Deriving Integrity Maintaining Triggers from Transition Graphs. In *International Conference on Data Engineering*, 1993.
[11] J. A. Blakeley, P. A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proc. ACM SIGMOD*, pp.61–71, Washington D.C., June 1986.
[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD* pp.157–167, Washington D.C., May 1993.
[13] A. P. Sistla and O. Wolfson. Temporal Triggers in Active Databases. *IEEE Transactions on Knowledge and Data Engineering*, 1995 (to appear).
[14] 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*, 1990.
[15] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems. *Springer-Verlag*, 1992.
[16] I. S. Mumick. Query Optimization in Deductive and Relational Databases. PhD thesis, Stanford University, 1991.
[17] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic Conditions. In *Proc. ACM Symposium on Principles of Database Systems*, pp.314–330, Nashville, Tennessee, 1990.
[18] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman. Magic sets and other strange ways to implement logic Programs. In *Proc. ACM Symposium on Principles of Database Systems*, 1986.
[19] D. Toman and J. Chomicki. Implementing Temporal Integrity Constraints Using an Active DBMS. In *Proc. RIDE'94 Active Database Systems*, Houston, pp.87-95, 1994.
[20] INGRES/SQL Reference Manual for Unix and VMS operating systems. Release 6.4, Ingres Corp. 1991.

Jan Chomicki received his M.S. from Warsaw University, Warsaw, Poland, and his Ph.D. from Rutgers University, New Brunswick, New Jersey, both in Computer Science. He is currently an Assistant Professor at the Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas. He has published more than 20 papers on temporal, deductive and constraint databases, and has served on a number of program committees of major database conferences.

David Toman received the M.S. degree in computer science from the Masaryk University, Brno, Czech Republic in 1992. He is currently working towards his Ph.D. degree at the Department of Computing and Information Sciences, Kansas State University. His research interests include temporal, deductive, and constraint database systems, logic programming, and programming languages.