Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding *

Jan Chomicki

Computing and Information Sciences
Kansas State University
Manhattan, KS 66506
chomicki@cis.ksu.edu

May 22, 1995

Keywords: database integrity, integrity constraints, temporal logic, temporal databases, active databases, real-time databases, triggers.
1989 CR Classification: F.4.1, H.2.3.

Abstract: We present an efficient implementation method for temporal integrity constraints formulated in Past Temporal Logic. Although the constraints can refer to past states of the database, their checking does not require that the entire database history be stored. Instead, every database state is extended with auxiliary relations that contain the historical information necessary for checking constraints. Auxiliary relations can be implemented as materialized relational views.

1 Introduction

Integrity constraints form an essential part of every database application. It is customary to distinguish between two kinds of constraints: static and temporal (or dynamic). *Static* constraints refer to the current state of the database, e.g.,*"every manager is also an employee"*, while *temporal* constraints may refer to past and future states in addition to the current state, e.g., *"salaries of employees should never decrease"* or *"once a student drops out of the Ph.D. program, she should not be readmitted"*. Static constraints are relatively well understood and are even included in the SQL2 standard [37], although they have been implemented in only a few commercial DBMS. Static constraints are usually expressed in some variant of first-order predicate logic [28]. Various methods to efficiently check such constraints have been developed [8, 35, 50, 53, 52].

*To appear in *ACM Transactions on Database Systems.* Early versions of some of the results in this paper appeared in the references [10] and [11]. Research partially supported by NSF grant IRI-9110581.

In this paper, we present the case for *temporal* integrity constraints. We argue that they should be specified declaratively in a restricted version of First-Order Temporal Logic (called *FOTL* hereafter). We describe a method to efficiently check such constraints and outline how it may be implemented using existing database technology.

Temporal integrity constraints appear in many application areas: office automation, financial trading, credit record evaluation, judicial records, software project databases. In general, they are necessary whenever *time-related* policies need to be supported. We assume that the contents of a database accurately reflect the relevant part of the real world and that a stored database history exactly corresponds to the real world history. Therefore, a stored database history can only be modified by extending it. Such databases are often called *transaction time* databases [59].

Specifying temporal constraints declaratively instead of embedding appropriate checks in application programs has numerous advantages. First, it enables the database designer to concentrate on the policies (*what* is to be enforced) instead of the mechanism (*how* to enforce it). This advantage has been recognized for a long time in the context of relational query languages. Second, it greatly simplifies application programs and makes them more modular by factoring out integrity checks. Third, it makes it possible to change constraints without changing the database schema and the application programs. Fourth, it allows the automatic optimization of constraint enforcement. The use of a logical language like *FOTL* for the specification of constraints brings additional benefits. Each constraint can be specified independently as a logical formula and we can meaningfully talk about adding or removing a single constraint. Also, properties like consistency of constraints can be precisely defined. Recently, *FOTL* has been proposed as an appropriate language for expressing *temporal queries* [64, 26, 17, 18], so using it for expressing integrity constraints seems a natural next step.

We propose that temporal constraints be expressed as *Past FOTL* formulas, i.e., formulas that use only temporal connectives referring to the past. Most temporal integrity constraints can be specified using past formulas in a natural way. Furthermore, we show that integrity constraints expressed as past formulas can be very efficiently checked.

Constraints are usually enforced by checking whether they are true after every database change. Temporal constraints require checking not only the current database state but also the entire database history. This is, however, clearly infeasible and precludes any practical implementation. We propose a method to automatically derive the historical information that has to be kept in a database state for the purpose of checking temporal constraints (assuming the constraints are written as past formulas). In the case of the relational data model, the historical information is stored as *auxiliary* relations in every database state. Checking temporal constraints after an update can now be done using only the current database state (extended with auxiliary relations) and is thus very efficient. Auxiliary relations are updated by evaluating first-order logic formulas (relational calculus queries) whose definitions are automatically derived from the constraints. The space requirements of our method are analyzed using a novel notion of *bounded history encoding*.

The proposed evaluation method can be applied not only to temporal integrity constraints but also to any logical conditions involving database history. For example, *active databases* [19] contain rules which are $(Condition, Action)$ pairs. The $Condition$ part may refer to history, as in the rule *"if the price of a stock has dropped three times in a row, sell the stock"*. In this way, historical *trends* can be monitored. Also, the need for efficient storage of past information has been recognized in the context of applying $FOTL$ to simulation [62, 63].

Many temporal constraints and conditions involve the notion of a clock, e.g., *"a recalled book should be returned within seven days"*. We show that the necessary *real-time* constructs can be expressed in *Past Metric FOTL*, an extension of *Past FOTL*. We describe the necessary changes to the constraint evaluation method.

In the presentation of our method, we assume that the set of temporal constraints is fixed. Allowing it to change during the lifetime of the database leads to certain semantic problems. In section 7, we outline these problems and suggest how our method can be adapted to the case of evolving sets of constraints.

We discuss now an extended example of the use of temporal logic in active databases.

Example 1.1 Consider a camera monitoring traffic at an intersection of two 4-lane highways. The camera takes periodic snapshots of the intersection, say every second. Two such snapshots are shown in Figure 1. Every snapshot is stored as a database state. Suppose we want to use the camera to monitor various traffic violations. An example of such a violation is *"turning left from the right lane"* (depicted in Figure 1). This violation can be described by the following temporal condition:

> *a car is exiting the intersection in the direction d_2 and at some time in the past the car was entering the intersection in the right lane in the direction d_1 such that d_2 is to the left of d_1; in the meantime the car was in the intersection.*

This condition, as well as many other conditions corresponding to traffic violations, can be readily described using $FOTL$. Now we can imagine that the snapshots of the intersection are stored in an active DBMS. This DBMS has for every traffic violation a temporal trigger that fires whenever the violation is detected.

However, if many cars pass the intersection the amount of information stored can quickly overwhelm the available memory resources. It is thus important to keep only the information about those cars that can still produce a traffic violation in the intersection and remove the information about those that have already left the intersection. This objective is automatically achieved if our evaluation method is used.

The outline of the paper is as follows. In section 2, we describe the syntax and semantics of *Past FOTL*, and give several examples of constraints and triggers formulated using this language. In section 3, which forms the core of the paper, we describe our method of checking temporal constraints formulated in *Past FOTL*. We prove its correctness and analyze the issues arising in the implementation. We also show how to deal with temporal triggers. In section 4, we propose how to express real-time constraints and conditions in the context of *Past Metric FOTL*. In section 5, we take a more general perspective and define a framework

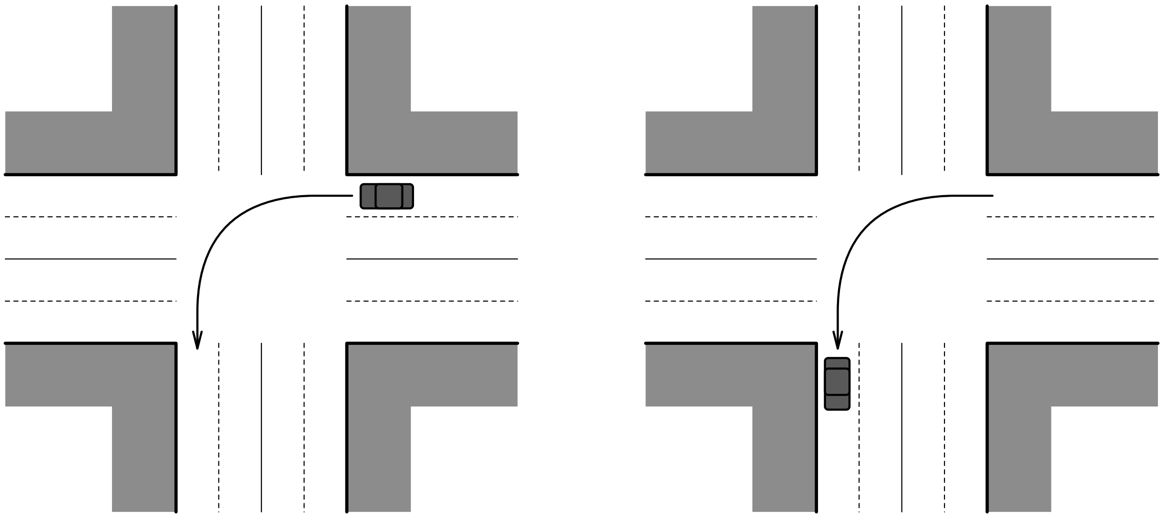

Figure 1: Catching a traffic violation

for constraint satisfaction where constraints can be arbitrary *FOTL* formulas, referring to both past and future. This makes it possible to compare our approach to other related work in section 6. Finally, in section 7 we draw conclusions and sketch possible extensions of this work.

2 Basic Notions

In this section, we introduce the basic concepts of *Past FOTL* that serves as our integrity constraint language. General *FOTL* is described in section 5.

2.1 Syntax

The expressions of *Past FOTL* are built from the following sets of symbols:

- a finite set of relation symbols;
- a finite set of constant symbols;
- a countably infinite set of variables;
- equality and inequality symbols: $\approx, \not\approx, <, >, \leq, \geq$ ("$\approx$" is used for equality to distinguish it from metalanguage identity);
- the logical connectives: $\vee, \wedge, \neg, \Rightarrow$;
- quantifiers: $\exists, \forall$;

- past-tense temporal connectives: $\bullet$ and **since** (future tense ones are introduced in section 5).

Each relation symbol is given with an arity $n \geq 0$. We refer to the sets of relation and constant symbols as the *vocabulary* of the language. This part is not fixed; we will consider languages over different vocabularies. A *term* is a constant or a variable. An *atomic formula* is of the form $t_1 \sim t_2$ or $p(t_1, \ldots, t_n)$, where the t_i's are terms, $\sim \in \{\approx, \not\approx, <, >, \leq, \geq\}$, and p is a relation symbol of arity n. Other formulas are defined by the following rules: if A, B are formulas and x is a variable, the following are also formulas: $A \vee B$, $A \wedge B$, $\neg A$, $A \Rightarrow B$, $\exists x A$, $\forall x A$, $\bullet\, A$ ("previous time A"), and A **since** B.

A formula is called:

- *pure first-order* if it does not use temporal connectives at all,
- *temporal* if it is of the form $\bullet\, A$, or A **since** B (i.e., the main connective is temporal).

The notions of *Past FOTL* formula and *past* formula are synonymous.

2.2 Semantics

A *first-order structure* M over a given vocabulary consists of a *universe* $|M|$, an *interpretation* $c^M \in |M|$ for each constant symbol c and an interpretation $p^M \subseteq |M|^n$ for each relation symbol p of arity n. We say that a *relation p of M is true about* a tuple $(a_1, \ldots, a_n)$ of elements of $|M|$ if $(a_1, \ldots, a_n) \in p^M$.

In order to interpret *Past FOTL* formulas, we assume that time is isomorphic to the natural numbers, i.e., *time instants* form an infinite sequence $i = 0, 1, 2, \ldots$ A *finite-time temporal database* (also called *temporal database*, *database history* or just *history*) is a finite sequence

$$D = (D_0, D_1, D_2, \ldots, D_k)$$

of first-order structures over the same vocabulary and with the same universe which we will call the *universe of the database* and denote $|D|$ (so $|D| = |D_0| = |D_1| = \ldots = |D_k|$). We refer to the structure D_i as to the *database state* at time instant i. We assume that a constant symbol c has the same interpretation in each database state (i.e. $c^{D_i} = c^{D_0}$, for each $i \leq k$) which we shall denote simply by c^D. The interpretation of a relation symbol may vary from one time instant to another.

We make the following assumptions about database states: the universe is *infinite* and *countable*, and the interpretation of any relation symbol is a *finite* relation (called a *database relation*). These are rather standard assumptions in database theory. (Notice that the equality and inequality symbols are interpreted as infinite relations with a fixed semantics. Those relations are not database relations.) For notational convenience only, we shall also assume that, unless otherwise stated, the universe of each database state equals to the set of natural numbers N, and the equality and inequality symbols have their standard interpretations.

Let $D = (D_0, D_1, D_2, \ldots, D_k)$ be a temporal database. A *valuation* is a mapping v that associates a value $v(x) \in N$ with each variable x. Notice that the value of a variable does not change with time (in other words, all variables we consider are *global* or *rigid*). It is convenient to extend the valuation to terms by setting $v(c) = c^D$, for a constant symbol c. Now we define the truth value of a formula φ at a time instant i under a valuation v by means of the *satisfaction* relation, *"φ is satisfied by D at the time instant i under the valuation v"*, in symbols $D, v, i \models \varphi$, defined inductively as follows:

- $D, v, i \models t_1 \sim t_2$, whenever $v(t_1) \sim v(t_2)$ where $\sim \in \{\approx, \not\approx, <, \leq, >, \geq\}$,
- $D, v, i \models p(t_1, \ldots, t_n)$, whenever the relation p of D_i is true about $(v(t_1), \ldots, v(t_n))$,
- $D, v, i \models \neg A$, iff not $D, v, i \models A$,
- $D, v, i \models A \wedge B$, iff $D, v, i \models A$ and $D, v, i \models B$, similarly for other logical connectives,
- $D, v, i \models \exists x A$ iff there exists $d \in N$ such that $D, v[x \leftarrow d], i \models A$, where $v[x \leftarrow d]$ is a valuation that maps x to d and otherwise coincides with v,
- $D, v, i \models \forall x A$ iff for all $d \in N$, $D, v[x \leftarrow d], i \models A$,
- $D, v, i \models \bullet A$ iff $i > 0$ and $D, v, i-1 \models A$,
- $D, v, i \models A$ **since** B iff for some j, $0 \leq j < i$, $D, v, j \models B$ and for all m, $j < m \leq i$, $D, v, m \models A$.

A formula with no occurrences of free variables is said to be *closed* or a *sentence*. Clearly, the truth value of a closed formula does not depend on the valuation, so if A is a sentence, we simply write $D, i \models A$. We also write $D, v \models A$ (resp. $D \models A$) to mean $D, v, i \models A$ (resp. $D, i \models A$) for all $i = 1, \ldots, k$. If $D \models A$, we say that D is a *model* of A.

Every *FOTL* formula A can be treated as a query. In this case the set

$$\{v : D, v, i \models A\}$$

constitutes the *answer* to A at instant i. The answer can be viewed as a relation of arity n where n is the number of free variables in A.

Other well known temporal connectives $\blacklozenge$ (*"sometime in the past"*) and $\blacksquare$ (*"always in the past"*) can be defined using **since**:

$$\blacklozenge A = \mathit{True} \textbf{ since } A$$

$$\blacksquare A = \neg \blacklozenge \neg A.$$

Notice that in our definitions of **since** we choose their its *strict* versions by requiring that $j < i$. It is easy to see that the non-strict version can be defined from strict ones and vice versa. Our evaluation method is formulated for the strict version of **since** but it can easily be adapted to the non-strict one. The strict version of **since** is preferable because it makes more temporal constraints practically evaluable than the non-strict version, as shown in section 3.

2.3 Examples

We give here several examples of how *Past FOTL* can be used for the specification of temporal integrity constraints or conditions.

Example 2.1 Consider a temporal database $(D_0, \ldots, D_k)$. Each state D_{i+1} in this database results from an update applied to the state D_i.

Assume that the database stores information about graduate students. A state consists of *new* information about the status of a number of students, i.e., whether they are admitted, graduate, drop out, or are reinstated. Old information is available in the previous states.

The constraint *"once a student drops out of the Ph.D. program, she should not be readmitted"* can be written as the past formula C_0:

$$C_0 : \neg(\exists x)(\mathit{Admitted}(x) \wedge \blacklozenge\ \mathit{Dropout}(x)).$$

A looser constraint, *"if a student drops out and is not subsequently reinstated, she should not be readmitted"*, can be formulated (this time the operator **since** is essential) as the formula C_1:

$$C_1 : \neg(\exists x)(\mathit{Admitted}(x) \wedge (\neg \mathit{Reinstated}(x) \ \mathbf{since}\ \mathit{Dropout}(x))).$$

Finally, the constraint *"a student cannot be reinstated more than twice"* is written as the formula C_2:

$$C_2 : \neg(\exists x)(\mathit{Reinstated}(x) \wedge \blacklozenge\ (\mathit{Reinstated}(x) \wedge \blacklozenge\ \mathit{Reinstated}(x))).$$

The last example demonstrates the convenience of using a declarative, logical language to express constraints. The additional restriction on the number of reinstatements is added in an incremental way, without any change to previously existing constraints.

Our final example is the constraint C_3: *"a teaching assistant cannot earn less than at any time in the past"*:

$$C_3 : \neg(\exists x, sal_1, sal_2)((\blacklozenge\ \mathit{Earns}(x, sal_1)) \wedge \mathit{Earns}(x, sal_2) \wedge sal_2 < sal_1).$$

Note that temporal integrity constraints relate events arbitrarily far apart in time. None of the above constraints can be directly expressed as a *transition constraint* that relates only the state after the update and the state before the update.

Example 2.2 The condition corresponding to the traffic violation *"turning left from the right lane"* (Example 1.1) can be described as:

$$\begin{aligned} C_4 : \ & (\exists d_1, d_2)(\mathit{LeftOf}(d_2, d_1) \wedge \mathit{Exiting}(x, d_2) \\ & \wedge(\mathit{Intersection}(x)\ \mathbf{since}\ (\mathit{Entering}(x, d_1) \wedge \mathit{RightLane}(x)))). \end{aligned}$$

3 Efficient temporal constraint evaluation

In this section we describe a method for efficient checking of temporal integrity constraints. The method is applicable only to constraints specified as *Past FOTL* formulas. First, we define the basic notions. Second, we describe the method, prove its correctness and analyze its space and time requirements. Finally, we discuss various implementation issues and show how to deal with temporal triggers.

3.1 Basic notions

A *temporal integrity constraint* is a closed *Past FOTL* formula. The *satisfaction set* $Sat(C)$ for a temporal integrity constraint C is defined as

$$Sat(C) = \{D : D = (D_0, \ldots, D_k) \wedge \forall 0 \leq i \leq k, (D_0, \ldots, D_i), i \models C.\}$$

The intuition here is that a database history $D = (D_0, \ldots, D_k)$ satisfies an integrity constraint C at time k if all the prefixes of D satisfy C.

A *naive* approach to temporal constraint evaluation requires storing the entire current history $(D_0, \ldots, D_k)$. When an update leading to a new state is presented, the current history is extended with the new state and the constraints are evaluated in the resulting new history. If they are satisfied, the new history becomes the current history. Otherwise, the current history remains unchanged (i.e., the update is rejected).

The basic idea behind our approach is that for evaluating a given finite set of temporal constraints it is not necessary to store the entire database history. Instead, every state D_i is augmented with new *auxiliary* relations to form an *extended state* $\hat{D}_i$. A finite sequence of extended states is called an *extended temporal database*. The auxiliary relations provide an *encoding* for the database history under the given set of constraints. Thus, the extended state $\hat{D}_i$ encodes the entire database history $(D_0, \ldots, D_i)$. At any time, only one extended state - the current one - is permanently stored. When an update leading to a new state D_{i+1} is presented, the new extended state $\hat{D}_{i+1}$ is constructed on the basis of D_{i+1} and the current extended state $\hat{D}_i$. The constraints are checked in the new extended state $\hat{D}_{i+1}$. If they are satisfied, the new extended state becomes the current extended state. Otherwise, the current extended state remains unchanged. Clearly, a history encoding needs to be correct, which means that checking constraints using the last extended state and using the entire history should give the same results.

A history encoding will be called *lossless* if no historical information is lost, and *lossy* otherwise. The naive approach described above may be viewed as providing a lossless history encoding. The encoding that we propose is *lossy*. Consequently, general temporal queries cannot be precisely answered in this approach because the relevant past information may be missing. Our encoding is, however, usually more space-efficient than storing the entire history. This is formalized later in this section by showing that the former requires bounded space for database histories containing values from any domain subset of bounded cardinality, while the latter may require unbounded space.

3.2 Method

Our method of evaluating temporal integrity constraints has two components: *constraint evaluation* in an extended database state and *auxiliary relation updates.* Below we describe the method and show its correctness.

Extended schema

We assume the database has a fixed, finite schema. Database states are instances of this schema. As before, we will denote by D_i the database state corresponding to instant i. The (finite) set of temporal integrity constraints is denoted by IC.

Consider every constraint $C \in IC$ in turn. The extended database schema contains, in addition to database relations, an *auxiliary* relation r_α for every temporal subformula α of C, i.e., for every subformula of the form $\bullet A$ or A **since** B. For each free variable of α, there is a different attribute of r_α. We call the vector of all such free variables, ordered by their first textual occurrence in C, the *variable vector* of α. The extended schema also contains a special auxiliary relation r_C, called a *constraint* relation. This relation is 0-ary. (Constraint relations are not strictly necessary but their presence simplifies the presentation.) Denote the set of extended relations corresponding to a constraint C by E_C. An extended database state contains instances of every auxiliary relation in the extended schema E_C.

Example 3.1 Consider the constraint C_1 from Example 2.1. This constraint has a single temporal subformula $\alpha_1 = (\neg Reinstated(x)\ \textbf{since}\ Dropout(x))$. Thus the extended schema contains a unary auxiliary relation symbol r_{α_1} (α_1 has one free variable) such that:

$$r_{\alpha_1}(x) \Leftrightarrow (\neg Reinstated(x)\ \textbf{since}\ Dropout(x)).$$

This new relation has a very intuitive meaning: it contains all students that dropped out and have not been subsequently reinstated. The extended schema also contains a 0-ary constraint relation symbol r_{C_1} where:

$$r_{C_1} \Leftrightarrow \neg(\exists x)(Admitted(x) \wedge (\neg Reinstated(x)\ \textbf{since}\ Dropout(x))).$$

Consider now the constraint C_2 from Example 2.1. The extended schema contains additionally two unary auxiliary relation symbols r_{α_2} and r_{α_3} such that:

$$r_{\alpha_2}(x) \Leftrightarrow \blacklozenge\ Reinstated(x).$$
$$r_{\alpha_3}(x) \Leftrightarrow \blacklozenge\ (Reinstated(x) \wedge \blacklozenge\ Reinstated(x)).$$

The extended schema contains also a 0-ary constraint relation symbol r_{C_2}.

If we are dealing with temporal conditions instead of constraints, then it is necessary to talk about *condition relations* that have as many attributes as there are free variables in the condition.

Example 3.2 Consider now the condition C_4 from Example 2.2. The extended schema contains additionally two auxiliary relation symbols r_C and r_{α_4} such that:

$$\begin{aligned} r_C(x) \Leftrightarrow \quad & (\exists d_1, d_2)(\mathit{LeftOf}(d_2, d_1) \wedge \mathit{Exiting}(x, d_2) \\ & \wedge(\mathit{Intersection}(x) \ \mathbf{since} \ (\mathit{Entering}(x, d_1) \wedge \mathit{RightLane}(x)))) \end{aligned}$$

and

$$r_{\alpha_4}(x, d_1) \Leftrightarrow \mathit{Intersection}(x) \ \mathbf{since} \ (\mathit{Entering}(x, d_1) \wedge \mathit{RightLane}(x)).$$

The first relation is a condition relation and contains information only about the cars that are exiting the intersection (and have committed the traffic violation), and the second - only about those still in the intersection. Thus irrelevant past information is automatically discarded.

It is easy to see that the number of auxiliary relation symbols is fixed for a fixed set of constraints. Also, in the above examples the size of auxiliary relations depends only on the set of values that appear in database relations.

Inductive definition of auxiliary relations

We show now how to obtain the formulas defining auxiliary relations. Clearly, we would like to have the property that an auxiliary relation r_α at instant i contains exactly those domain values that make α true at i. To achieve this, auxiliary relations are defined *inductively*. First, their instances at instant 0 are defined, and then it is shown how to obtain the instances at instant $i+1$ from those at instant i. Of course, in both cases the instances of auxiliary relations will also depend on the instances of the corresponding database relations.

Denote by r^i the relation corresponding to the (database or auxiliary) relation symbol r in the extended state $\hat{D}_i$ that appears in an extended temporal database $\hat{D} = (\hat{D}_0, \ldots, \hat{D}_k)$. We can view $\hat{D}$ as a relational database with the schema consisting of relation symbols r^i for $i = 1, \ldots, k$. (This view is used only to simplify the correctness proof. The implementation uses a fixed number of auxiliary relations for a fixed set of integrity constraints, as described before.) The instance of r^i in $\hat{D}$ is the instance of r in $\hat{D}_i$. The satisfaction of a first-order formula that uses the relation symbols r^i is defined in the standard way. Thus:

$$\hat{D}, v \models r^i(t_1, \ldots, t_n) \text{ iff } \hat{D}, v, i \models r(t_1, \ldots, t_n).$$

where in the left-hand formula $\hat{D}$ is viewed as a relational database.

Let α be a temporal subformula of a constraint C. We are going to construct a pure first-order formula $\alpha^{i,i}$ with the property that all the relation symbols that it uses are of the form r^i. Intuitively, this formula will define α in terms of the extended state $\hat{D}_i$. (The double superscript in $\alpha^{i,i}$ is for uniformity with the notation introduced later in this section.)

To obtain $\alpha^{i,i}$ we apply the inductive definitions (induction on the structure of the formula) in Figure 2. (The cases for $\vee$ and $\forall$ are omitted because those symbols can be eliminated from the formula by well-known logical equivalences.)

	α	$\alpha^{i,i}$
1	$True$	$True$
2	$w_1 \sim w_2$	$w_1 \sim w_2$
3	$p(t_1, \ldots, t_n)$	$p^i(t_1, \ldots, t_n)$
4	$\neg\beta$	$\neg\beta^{i,i}$
5	$\beta \wedge \gamma$	$\beta^{i,i} \wedge \gamma^{i,i}$
6	$(\exists y)\beta$	$(\exists y)(\beta^{i,i})$
7	β	$r^i_\beta(x_1, \ldots, x_n)$

	α	$\alpha^{i,i+1}$
1'	$True$	$True$
2'	$w_1 \sim w_2$	$w_1 \sim w_2$
3'	$p(t_1, \ldots, t_n)$	$p^{i+1}(t_1, \ldots, t_n)$
4'	$\neg\beta$	$\neg\beta^{i,i+1}$
5'	$\beta \wedge \gamma$	$\beta^{i,i+1} \wedge \gamma^{i,i+1}$
6'	$(\exists y)\beta$	$(\exists y)(\beta^{i,i+1})$
7'	$\bullet\, \beta$	$\beta^{i,i}$
8'	β **since** γ	$(\beta \textbf{ since } \gamma)^{i,i} \wedge \beta^{i,i+1} \vee \gamma^{i,i} \wedge \beta^{i,i+1}$

Figure 2: Definitions of $\alpha^{i,i}$ and $\alpha^{i,i+1}$

Notation:

- p is a database relation.
- in row 7, β is a temporal formula.
- $(x_1, \ldots, x_n)$ is the variable vector of the formula β.

Example 3.3 Consider the constraint C_6:

$$C_6 : \neg(\exists x)(P(x) \wedge \blacklozenge P(x)).$$

We have two auxiliary relations r_{α_5} and r_{C_6}. The unary relation r_{α_5} corresponds to the formula

$$\alpha_5 = \blacklozenge P(x).$$

Now

$$\alpha_5{}^{i,i} = r^i_{\alpha_5}(x)$$

and

$$C_6{}^{i,i} = \neg(\exists x)(P^i(x) \wedge r^i_{\alpha_5}(x)).$$

It remains to be shown how instances of auxiliary relations are obtained. They are defined inductively (induction on time combined with induction on structure).

The instance of every auxiliary relation r_α corresponding to a temporal subformula α of C at instant 0 is empty (thus $\alpha^{0,0} = \mathit{False}$). This is because temporal formulas of the form $\bullet\, A$ or A **since** B are false at 0. The instance of the constraint relation r_ψ at instant 0 is obtained by constructing the formula $\psi^{0,0}$ and evaluating it in the first database state D_0.

The instance of an auxiliary relation r_α at instant $i+1$ is defined in terms of the instances of all relations at instant i and the instances of database relations at instant $i+1$. This is done by constructing a pure first-order formula $\alpha^{i,i+1}$ and evaluating it in the pair of states $\hat{D}_i$ (an extended state) and D_{i+1}. The formula $\alpha^{i,i+1}$ is defined inductively (induction on time combined with induction on structure), as shown in Figure 2.

It should be clear that $\alpha^{i,i+1}$ is a pure first-order formula in which every relation symbol will be of one of the following forms:

- p^i or p^{i+1} where p is a database relation symbol, or
- r^i where r is an auxiliary relation symbol.

Example 3.4 Returning to Example 3.3, consider again the constraint C_6:

$$C_6 : \neg(\exists x)(P(x) \wedge \blacklozenge P(x))$$

and two auxiliary relations in E_{C_6}: r_{α_5} and r_{C_6}. Now

$${\alpha_5}^{i,i+1} = r^i_{\alpha_5}(x) \vee P^i(x)$$

and

$${C_6}^{i,i+1} = \neg(\exists x)(P^{i+1}(x) \wedge (r^i_{\alpha_5}(x) \vee P^i(x)).$$

Correctness

We address now the issue of the *correctness* of our method. Consider a sequence of states $D = (D_0, \ldots, D_k)$ and the corresponding sequence of extended states $\hat{D} = (\hat{D}_0, \ldots, \hat{D}_k)$, viewed as a relational database as before. We first prove the following lemma.

Lemma 3.1 *For any α which is temporal subformula of a constraint C or C itself, any valuation v, and any i, $0 \leq i < k$:*

$$\hat{D}, v \models \alpha^{i,i+1} \text{ iff } \hat{D}, v \models \alpha^{i+1,i+1}.$$

Proof: If $\alpha = p(t_1, \ldots, t_n)$ where p is a database relation symbol, the thesis holds because the rows 3 and 3' in Figure 2 define identical formulas. Similarly for α which is an equality or an inequality. If α is a temporal subformula of the constraint C, then $\alpha^{i,i+1}$ is the inductive definition of the auxiliary relation r_α at instant $i+1$. On the other hand, $\alpha^{i+1,i+1}$ is simply $r^{i+1}_\alpha(x_1, \ldots, x_n)$. Because r^{i+1}_α is defined inductively using the formula $\alpha^{i,i+1}$, the equivalence holds. The remaining cases are easily shown by structural induction that makes use of the fact that the rows 1, 2, 4, 5, and 6 parallel the corresponding rows 1', 2', 4', 5', and 6' in Figure 2. $\Box$

Theorem 3.1 *For any α which is temporal subformula of a constraint C or C itself, any valuation v, and any i, $0 \leq i \leq k$:*

$$D, v, i \models \alpha \text{ iff } \hat{D}, v \models \alpha^{i,i}.$$

Proof: We are going to use double induction: on i and on the structure of the formula α.

For $i = 0$ the theorem clearly holds. Assume that for some i, $i \geq 0$, and all proper subformulas δ of C:

$$D, v, i \models \delta \text{ iff } \hat{D}, v \models \delta^{i,i}.$$

We are going to prove that

$$D, v, i+1 \models \alpha \text{ iff } \hat{D}, v \models \alpha^{i+1,i+1}.$$

for all proper subformulas α of C. If α is a database atom, an equality or an inequality, this clearly holds which provides the base case for structural induction. The proofs for negation, conjunction and quantification are very similar, thus we consider here only quantification, i.e., $\alpha = \exists y \beta$. We have:

$D, v, i+1 \models \exists y \beta$
iff for some d, $D, v[y \leftarrow d], i+1 \models \beta$
iff for some d, $\hat{D}, v[y \leftarrow d] \models \beta^{i+1,i+1}$ (induction on structure)
iff $\hat{D}, v \models \exists y \beta^{i+1,i+1}$
iff $\hat{D}, v \models (\exists y \beta)^{i+1,i+1}$ (row 6 in Figure 2).

Now consider $\alpha = \beta$ **since** γ. Then:

$D, v, i+1 \models \beta$ **since** γ
iff $D, v, i \models \beta$ **since** γ and $D, v, i+1 \models \beta$ or $D, v, i \models \gamma$ and $D, v, i+1 \models \beta$
iff $\hat{D}, v \models (\beta \textbf{ since } \gamma)^{i,i}$ and $\hat{D}, v \models \beta^{i+1,i+1}$ or $\hat{D}, v \models \gamma^{i,i}$ and $\hat{D}, v \models \beta^{i+1,i+1}$
(induction on i and structure)
iff $\hat{D}, v \models (\beta \textbf{ since } \gamma)^{i,i}$ and $\hat{D}, v \models \beta^{i,i+1}$ or $\hat{D}, v \models \gamma^{i,i}$ and $\hat{D}, v \models \beta^{i,i+1}$
(Lemma 3.1)
iff $\hat{D}, v \models (\beta \textbf{ since } \gamma)^{i,i+1}$ (row 8' in Figure 2)
iff $\hat{D}, v \models (\beta \textbf{ since } \gamma)^{i+1,i+1}$ (Lemma 3.1)

The case $\alpha = \bullet\, \beta$ is handled similarly. $\square$

Theorem 3.1 implies that our method correctly computes the instances of auxiliary relations and correctly implements a history encoding.

Corollary 3.1 *For any α which is temporal subformula of a constraint C or C itself, any valuation v, and any i, $0 \leq i \leq k$:*

$$D, v, i \models \alpha \textit{ iff } \hat{D}, v \models r^i_\alpha(x_1, \ldots, x_n)$$

where $(x_1, \ldots, x_n)$ is the variable vector of α.

Corollary 3.2 *$D = (D_0, \ldots, D_k)$ is in the satisfaction set $Sat(C)$ iff r_C contains the empty tuple $()$ in the extended states $\hat{D}_0, \ldots, \hat{D}_k$.*

3.3 Implementation issues

Using a relational DBMS

We implement our method using a relational DBMS. An auxiliary relation r_α is implemented as at most two materialized relational views: one to materialize r_α in the first extended state $\hat{D}_0$, the other to materialize it in subsequent extended states. The first view is only necessary for constraint or condition relations (other auxiliary relations are empty in the first state by definition).

Example 3.5 Consider the constraint C_1 from Example 2.1:

$$C_1 : \neg(\exists x)(\mathit{Admitted}(x) \wedge (\neg \mathit{Reinstated}(x) \ \mathbf{since} \ \mathit{Dropout}(x))).$$

As discussed earlier, our method introduces two auxiliary relations for this constraint: r_{α_1} corresponding to the formula $\alpha_1 = (\neg \mathit{Reinstated}(x) \ \mathbf{since} \ \mathit{Dropout}(x))$ and the constraint relation r_{C_1}. Using the inductive definitions in Figure 2 we obtain for every $i \geq 0$:

$$r^{i+1}_{\alpha_1}(x) \Leftrightarrow r^{i}_{\alpha_1}(x) \wedge \neg \mathit{Reinstated}^{i+1}(x) \vee \mathit{Dropout}^{i}(x) \wedge \neg \mathit{Reinstated}^{i+1}(x).$$

The relation symbols with the superscript $i+1$ denote relations in the state after the update, and those with i denote relations in the extended state before the update. Moreover, the above definition *does not depend on the value of i itself*, thus can be used as a definition of a single relational view that references relations in the states before and after the update. This view should be materialized in every state, except for the first one, in the same way. In the first state, the view is initialized to an empty relation.

A definition for the relation r_{C_1} can be obtained similarly leading to two materialized view definitions: one applicable only to the first state, the other to every subsequent state.

If triggers are supported by the DBMS, they can be used to maintain the materialized views [9]. Moreover, triggers can also be used to determine whether temporal constraints are satisfied after an update and abort the update if it is not the case. Checking constraints in our framework is particularly simple because it consists of testing the presence of empty tuples in the constraint relations (Corollary 3.2).

We have built a prototype implementation of our method [16]. Constraints formulated as *Past FOTL* formulas are compiled into a set of STARBURST rules (triggers) [45, 69] that maintain the materialized views corresponding to auxiliary relations and determine whether the constraints are satisfied after an update (a transaction). The implementation is completely transparent – no changes to the rule system are necessary.

The STARBURST rule system enables the user to specify relative ordering of rules. This feature is used by our implementation because there is a natural ordering on auxiliary relations determined by the "is a subformula of" relationship. Consequently, the views are materialized in a specific order (the auxiliary relation corresponding to a formula after the relations corresponding to its subformulas). This makes it possible to avoid the recomputation of intermediate results. This optimization requires that the inductive definitions of $\alpha^{i,i+1}$ be modified: The definitions of $\alpha^{i,i+1}$ contain now also auxiliary relation symbols corresponding to the temporal subformulas of α.

On the other hand, if the view materialization rules are not ordered, then being wholly independent they can be processed in parallel. They have to be serialized only with respect to the final group of rules that determines whether an update is to be committed or aborted.

Domain independence

A formula is *domain independent* if, when treated as a query, it has an answer that does not depend on the underlying domain of attribute values. The following are examples of domain dependent pure first-order formulas: $p(x) \vee q$, $\neg q(x)$, $p(x) \vee \neg q(x)$.

The *active domain* $adom_Q(D)$ of a relational database D and a query Q is the set of domain values that appear in some database relation in D or as constants in Q [1]. The notion of active domain may be generalized in a straightforward way to database histories and integrity constraints formulated in *Past FOTL*. Let $D = (D_0, \ldots, D_k)$ and IC be a set of integrity constraints. Then

$$adom_{IC}(D) = \bigcup_{C \in IC} \bigcup_{i=1}^{k} adom_C(D_i).$$

A domain independent query formula can be evaluated not with respect to the infinite universe but rather with respect to the *active domain*. The active domain of a relational database is finite. The same property holds for *finite-time* temporal databases. For the integrity checking method described in this section to work it is essential that the query formulas be evaluated with respect to the active domain because then query answers are finite and can be materialized.

Unfortunately, the notion of *domain independence* itself is not very practical. [22] has shown that it is undecidable to verify whether a first-order formula is domain independent. Therefore, many efforts [21, 61, 65] have been directed towards finding effectively recognizable subsets of domain independent formulas.

A *Past FOTL* formula is *domain independent* if its answer (when the formula is treated as a query) at every instant $i \geq 0$ does not depend on the domain.

Consider an temporal integrity constraint $C \in IC$. As discussed earlier, for every formula α such that α is C or a temporal subformula of C, there are pure first-order formulas that define the auxiliary relation r_α. Denote by F_C the set consisting of the formulas $\alpha^{0,0}$ (which is identical to *False* for α which is a temporal formula), and $\alpha^{0,1}$.

We call an integrity constraint C *PFO-domain-independent* (*pure first-order* domain independent) if all the (pure first-order) formulas in F_C are domain independent. We require that constraints be PFO-domain-independent because auxiliary relations have to be materialized and stored in the database. Allowing domain dependent formulas would lead to incorrect results.

We have the following property:

Proposition 3.1 *If C is PFO-domain-independent, then it is also domain independent.*

Proof: It can be shown by induction on formula structure and i that once $\alpha^{0,0}$ and $\alpha^{0,1}$ (both elements of F_C) define domain independent relations, so does $\alpha^{i,i+1}$ and $\alpha^{i,i}$ for every i. Thus by Corollary 3.2 the truth value of C at every instant i is determined regardless of the domain. □

It is even more important that an analogous property holds for several effectively recognizable classes of domain independent formulas. We call an integrity constraint *PFO-evaluable* if all the formulas in F_C are *evaluable* [21, 61, 65]. Our translator from *FOTL* to STARBURST rules is applicable to PFO-evaluable *Past FOTL* formulas.

The converse of Proposition 3.1 does not hold, as shown by the following example.

Example 3.6 Consider the formula $(\exists x)(p(x) \wedge \blacklozenge \neg q(x))$. This formula is domain independent, because in every state $p(x) \wedge \blacklozenge \neg q(x)$ evaluates to a subset of a database relation p. However, the subformula $\alpha_1 = \blacklozenge \neg q(x)$ leads to the following formula ${\alpha_1}^{0,1}$:

$$ {\alpha_1}^{0,1} = r^0_{\alpha_1}(x) \vee \neg q^0(x) $$

which is domain dependent.

Therefore, our constraint evaluation method imposes on constraints a requirement stronger that just domain independence of constraint formulas themselves. Any naive approach requires only domain independence of constraints and handles the above example in a correct way.

The choice of temporal operators may influence PFO-domain-independence. We were careful enough to require in the definition of A **since** B that the instant when B is true is strictly earlier than the instant when A **since** B is true. Suppose we allow that those two instants can be the same and denote the resulting operator by **since**$'$. The operators **since** and **since**$'$ can be defined in terms of each other. In particular:

$$ A \textbf{ since } B = (\bullet (A \textbf{ since}' B)) \wedge A $$

and

$$ A \textbf{ since}' B = (A \textbf{ since } B) \vee B. $$

An appropriate inductive definition for **since**$'$, analogous to 8' in Figure 2, is obvious.

Two logically equivalent formulas, one of which uses **since** and the other **since**$'$, may have quite different properties w.r.t. PFO-domain-independence, as demonstrated in the following example.

Example 3.7 Consider the formula $p(x)$ **since** q. It is easy to see that it is PFO-domain-independent. However, the equivalent formula

$$ (\bullet (p(x) \textbf{ since}' q)) \wedge p(x) $$

leads to the definition of $\delta^{0,0}$ (where $\delta = p(x)$ **since**$'$ q) as q^0. The latter definition is clearly domain dependent because the auxiliary relation r_δ is unary, not 0-ary. Thus the formula

$$ (\bullet (p(x) \textbf{ since}' q)) \wedge p(x) $$

is not PFO-domain-independent.

Proposition 3.2 *If an* FOTL *formula using* **since$'$** *but not* **since** *is PFO-domain-independent, so is the equivalent formula, obtained using the above equivalence, that uses* **since** *but not* **since$'$** *.*

The converse is not true, as Example 3.7 shows. Proposition 3.2 remains true if we replace "PFO-domain-independent" by "PFO-evaluable".

Space and time requirements

We estimate here the space and time requirements of our method under the assumption that all constraints are PFO-domain-independent. First, we define the notion of *bounded* history encoding and show that the encoding provided by our method is bounded.

Recall that $adom_{IC}(D)$ (defined earlier in this section) denotes the active domain of a database history D and a set of integrity constraints IC. Denote by $|S|$ the cardinality of a finite set S.

Assume IC is a finite set of temporal constraints and the database schema is fixed and finite. A history encoding under IC is *bounded* if:

1. the set of auxiliary relations created for IC is fixed,
2. for every $n \in N$, there is an $m_n \in N$ such that for every database history $D = (D_0, \ldots, D_k)$ with $|adom_{IC}(D)| < n$, the size of the extended state $\hat{D}_k$ that encodes D is less than m_n.

An encoding is $T(n)$-bounded if for all n, $m_n < T(n)$; polynomially bounded if it is $T(n)$-bounded for some polynomial $T(n)$.

The first requirement guarantees that the encoding gives rise to a well-defined notion of *extended* schema, namely a fixed set of auxiliary relations. The second requirement captures the intuition that the amount of historical information stored in auxiliary relations does not depend on the length of the database history but only on the number of domain values that appear there. (The second requirement was termed *history-less evaluation* in [10, 11].)

It should be noted that having bounded history encoding is a property of an integrity checking method. This property characterizes the space required by the method if the cardinality of the set of domain values that appear in the database is bounded from above. Clearly, it does not mean that the method can only handle databases over sets of values of bounded cardinality. The property is useful for distinguishing methods that require space depending on the length of the database history from those that require space depending only on the cardinality of the set of values that appear in the history. A method that does not provide a bounded history encoding can require arbitrarily more space than one that does.

Example 3.8 Consider the constraint C_5:

$$C_5 : \exists x \blacklozenge P(x).$$

and an infinite sequence $(D'_k)_{k\in N}$ of finite-time temporal databases $D'_k = (D_0, \ldots, D_k)$ such that the relation corresponding to P in all even-numbered states is empty. In the odd-numbered states this relation contains a single tuple (a). In the naive approach the different occurrences of (a) in every odd-numbered state have to be stored separately, requiring an unbounded amount of space. In our method, it is sufficient to store all such x that $\blacklozenge P(x)$ holds. In this example there is just one such x and consequently, a constant amount of storage suffices for any k.

Consider now a different sequence $(D'_k)_{k\in N}$ of temporal databases $D'_k = (D_0, \ldots, D_k)$, one in which the relation corresponding to P in every state D_i, $0 \leq i \leq k$, contains a single tuple (i). Then both approaches require unbounded space. However, in our approach this is due to the fact that the set of values in $(D'_k)_{k\in N}$ is unbounded.

A database history may be stored in several different ways: as a sequence of states, a sequence of state differences, a single state together with an incremental update log, or a combination of these. In all these cases, the size of the stored historical data and the time to check temporal integrity constraints depends on the length of the database history. Even if $adom_{IC}(D)$ consists of a single value, there are 2^k different sequences of k database states. Thus any lossless encoding of a sequence of k states requires at least k bits and thus requires an arbitrarily large amount of space as k grows indefinitely. Using our terminology, an encoding that preserves the entire database history cannot be bounded.

We analyze now the history encoding provided by our method.

Auxiliary relations take up additional storage space and are updated at the same time the database is updated. Define the *length* of a constraint to be the number of symbols in its textual representation. Thus the number of subformulas of a constraint is at most linear in its length. Consequently, the *number* of auxiliary relations is independent of the size of a database state and is at most linear in the length of integrity constraints. It will often be significantly smaller. For example, different constraints may share the same temporal subformulas. Also, the arity of any auxiliary relation is at most linear in the length of the constraint, as it is equal to the number of free variables in the temporal subformula corresponding to the relation.

Theorem 3.2 *The auxiliary relations defined in this section provide a polynomially-bounded history encoding.*

Proof: The database schema and the set of constraints are fixed and finite, and the constraints are PFO-domain-independent. Thus, from the above considerations we conclude that the set of auxiliary relations is fixed. To see that the second condition for boundedness is also satisfied, let $D = (D_0, \ldots, D_k)$ be a database history and $\hat{D} = (\hat{D}_0, \ldots, \hat{D}_k)$ the corresponding extended history. Because the constraints are PFO-domain-independent, auxiliary relations can only contain values that appear in some database relation in one of the states $D_0, \ldots, D_k$. Under the given assumptions, the arity of auxiliary relations is fixed and finite. Thus, the size of the auxiliary relations in the extended state $\hat{D}_k$ is at most polynomial in the number of domain values that appear in the states $D_0, \ldots, D_k$. (In fact, only the constants appearing in relations referenced in temporal constraints have to be considered.) □

Is bounded history encoding provided by our method always more space-efficient than storing the entire history? It appears that in most situations this is indeed the case, as our method makes it possible to avoid storing repeatedly information that is redundant from the point of view of constraint checking. However, materializing auxiliary relations may require computing additional joins or Cartesian products, and storing their results, which may indeed increase the space requirements.

Example 3.9 Consider any temporal constraint with the subformula $\alpha = \blacklozenge(p(x) \wedge q(y))$. Any extended state will in this case contain the Cartesian products of the instances of p and q in the preceding state. Thus if the values in the instances of p (and q) in every past state are different, our method will not produce any space savings and will in fact require quadratically more space than a method that stores the entire history.

It is not difficult to see that in the worst case our method requires polynomially more space than a naive method that stores all the states in the history. The relationship to methods that use other lossless encodings of history, e.g., an update log, is not clear. On the other hand, any lossless encoding of history always requires unbounded space and thus in many cases requires arbitrarily more space than the encoding provided by our method (Example 3.8). Therefore, the latter seems preferable, although the exact comparative analysis of the space requirements of both methods remains an open problem.

The time necessary for checking temporal constraints clearly depends on the size of the history encoding. For our method, this time is polynomial in the cardinality of the active domain of the current history.

Optimizations

Clearly, updating auxiliary relations takes time polynomial in the size of the active domain of the current history. This cost is not incurred by the naive methods because such methods do not store any additional information. A very simple *hybrid* method combines our approach with a naive one. In this method, the instances of auxiliary relations are allowed to *lag behind* the database state, i.e., they correspond to some past database state. The most recent databases states, those occuring after the last stored extended state, are stored explicitly or using an update log. Integrity checking uses both the recent states as well as the out-of-date values of auxiliary relations. The extended state becomes a sort of a *logical snapshot* and can be updated independently of the database changes, for example at given intervals. By moving the computation of extended states off-line, we avoid the need to recompute auxiliary relations after each update. On the other hand, integrity checking itself becomes now less efficient. It remains to be seen whether the above hybrid method offers any performance advantages.

Other optimizations are possible as well. For example, a constraint can be transformed to a form that makes possible further reduction in the amount of historical information stored. This and other optimization issues are addressed in [16].

Example 3.10 Consider the constraint C_5 from Example 3.8:

$C_5 : (\exists x) \blacklozenge P(x).$

This constraint can be transformed to an equivalent constraint C_6:

$C_6 : \blacklozenge (\exists x)\ P(x).$

Now the auxiliary relation $r_{\blacklozenge \exists x P(x)}$ is a 0-ary relation and the amount of historical information stored is constant for every temporal database. Also, there is no need now for a separate constraint relation.

3.4 Temporal triggers

A *temporal condition* is an arbitrary *Past FOTL* formula. *Temporal triggers* are of the following form:

if *Condition* **then** *Action*

where *Condition* is a temporal condition. The syntax and the semantics of the action language is of no concern to us here. Some proposals can be found in [19, 57]. The only requirement that we put on it is that *Action* should be able to share variables with *Condition*.

A temporal trigger "**if** C **then** A" *fires* at instant k for a (ground) valuation v if

$$(D_0, \ldots, D_k), v, k \models C$$

but for all i, $0 \leq i < k$,

$$(D_0, \ldots, D_i), v, i \not\models C.$$

Trigger firing and constraint satisfaction are thus *dual* notions, because a constraint C is violated at instant k if for all i, $0 \leq i < k$,

$$(D_0, \ldots, D_i), i \models C$$

but

$$(D_0, \ldots, D_k), k \not\models C.$$

In this situation the trigger

if $\neg C$ **then** *Action*

fires at instant k. Typically, constraints are formulas of the form $\forall C$. Thus their negations are of the form $\exists \neg C$. Th existential prefix is usually skipped in the trigger condition so the variables can be shared between the *Condition* part and the *Action* part.

Concluding, the evaluation of temporal conditions in triggers can be done using the method described earlier in this section.

4 Real-time constraints and conditions

Transactions that update the database already define a flow of time. When a transaction commits, the current state corresponds to the current instant and the previous state corresponds to the previous instant. This flow of time is referred to by the temporal connectives of *Past FOTL*. However, in many applications it is necessary to model a flow of time that is independent of database updates, for example the flow of physical time. Two adjacent database states may have the same physical time, for instance if the observed time has granularity of a day or hour, but also they can differ by more than one time instant. Consider a library database. Books borrowed on the same day will be associated with the same physical time. On the other hand, if the library is closed for a few days, the time associated with adjacent databases states can differ by more than a day.

Constraints or conditions referring to the flow of physical time will be called *real-time*. Such constraints raise new maintenance issues. For example, the constraint *"a recalled book should be returned within a week"* can be violated just by the flow of physical time, without any updates to the database. Also, violations due to the flow of time cannot be repaired by aborting the offending transaction. We will address those issues later in this section. We conclude this section by showing how real-time constructs make automatic data aging possible.

4.1 Temporal logic with real-time connectives

We propose here a simple extension of *Past FOTL*, called *Past Metric FOTL*, in which real-time constraints and conditions can be formulated. We show how to generalize the integrity checking method described in section 3 to constraints expressed in *Past Metric FOTL*. In addition to the temporal connectives of *Past FOTL*, *Past Metric FOTL* has a family of new real-time connectives: unary ones of the form $\bullet_{\sim c}$ and binary ones of the form $\mathbf{since}_{\sim c}$ (where $\sim$ is one of $=, <, >$ and c is a nonnegative integer.)

Consider a finite-time temporal database $D = (D_0, \ldots, D_k)$. We assume that for each state D_i, $0 \leq i \leq k$, a single value $clock(i)$ represents the corresponding integer clock. The clock value in a state is recorded by the transaction that commits to produce this state. Clearly, $clock(i+1) \geq clock(i)$ for all i. Now the semantics of the new real-time connectives is defined as follows (v is a valuation):

- $D, v, i \models \bullet_{\sim c} A$ iff $i > 0$ and $D, v, i-1 \models A$ and $clock(i) \sim clock(i-1) + c$,
- $D, v, i \models A \ \mathbf{since}_{\sim c}\ B$ iff for some j, $0 \leq j < i$, $D, v, j \models B$ and $clock(i) \sim clock(j) + c$, and for every m, $j < m \leq i$, $D, v, m \models A$.

The semantics of *Past FOTL* constructs is the same as defined in section 2. Other real-time operators can be readily defined:

$$\blacklozenge_{\sim c} A = \mathit{True}\ \mathbf{since}_{\sim c}\ A$$

$$\blacksquare_{\sim c} A = \neg \blacklozenge_{\sim c} \neg A.$$

Example 4.1 The constraint *"a recalled book should be returned within a week"* is expressed by the following *Past Metric FOTL* formula

$$C_7: \neg(\exists x)(\neg return(x)\ \mathbf{since}_{>7}\ (recall(x) \wedge book(x))).$$

Example 4.2 Consider now a stock market application. We may be interested in the condition "has the price of stock x dropped more than once today compared to the closing price of yesterday?". We write a corresponding *MTL* formula in several steps. We first define two formulas $SameDay$ (meaning that the previous database state has the same clock value as the current one) and $NewDay$ (meaning that the current state is the first state with this particular clock value):

$$SameDay = \bullet_{=0} True$$

$$NewDay = \neg \bullet_{=0} True.$$

Yesterday's closing price of stock x can now be described as:

$$SameDay\ \mathbf{since}\ (NewDay \wedge (\bullet\ price(x,p))).$$

The original condition is thus formulated as:

$$\begin{aligned} C_2: \quad (\exists p_1, p_2, p_3) \quad & (p_1 > p_2 > p_3\ \wedge\ (SameDay \\ & \mathbf{since}\ price(x,p_3)\ \wedge\ (SameDay \\ & \mathbf{since}\ price(x,p_2)\ \wedge\ (SameDay \\ & \mathbf{since}\ NewDay \wedge (\bullet\ price(x,p_1)))))). \end{aligned}$$

4.2 Integrity checking

Our method for checking real-time constraints will be based on the same philosophy as the method presented in section 3. Every database state D_i will be augmented by auxiliary relations to yield an extended state $\hat{D}_i$. As before, auxiliary relations will be associated with temporal subformulas of integrity constraints. However, auxiliary relations corresponding to formulas whose main connective is a real-time connective will be defined differently than those corresponding to other temporal connectives.

Consider every temporal subformula α of a real-time integrity constraint C or C itself. If the main connective of α is not real-time, then the auxiliary relation r_α is defined as in Section 3. If $\alpha = \bullet_{\sim c}\beta$, then the auxiliary relation r_α has an attribute for every free variable of α. If $\alpha = \beta\ \mathbf{since}_{\sim c}\ \gamma$, then r_α has an attribute for every free variable of α, as well as an additional *distance* attribute. Finally, if $\alpha = \beta\ \mathbf{since}_{>c}\ \gamma$, a secondary auxiliary relation s_α is defined which has an attribute for every free variable of α but has no distance attribute.

The intuition behind the new auxiliary relations is as follows. Assume $(x_1, \ldots, x_n)$ is the variable vector of α. If $\alpha = \bullet_{\sim c}\beta$, then as for *Past TL* we have

$$D, v, i \models \alpha \text{ iff } \hat{D}, v \models r^i_\alpha(x_1, \ldots, x_n).$$

If $\alpha = \beta$ $\mathbf{since}_{\sim c}$ γ, then

$$\hat{D}, v \models r^i_\alpha(x_1, \ldots, x_n, c_0) \text{ iff } c_0 \leq c \wedge D, v, i \models \beta \ \mathbf{since}_{=c_0} \ \gamma.$$

If $\alpha = \beta$ $\mathbf{since}_{>c}$ γ, then

$$\hat{D}, v \models s^i_\alpha(x_1, \ldots, x_n) \text{ iff } D, v, i \models \beta \ \mathbf{since}_{>c} \ \gamma.$$

In Figure 3 we provide the appropriate definitions of $\alpha^{i,j}$ for $j = i, i+1$ that extend those in Figure 2. In Figure 4 the instances of auxiliary relations in the extended state $\hat{D}_{i+1}$ are defined in terms of those in the extended state $\hat{D}_i$ and the instances of the database relations in the state D_{i+1}. In the first extended state $\hat{D}_0$, the instances of all the auxiliary relations associated with temporal subformulas are empty, as for *Past FOTL*.

	α	$\alpha^{i,j}, j = i, i+1$
A	$\bullet_{\sim c}\beta$	$r^j_\alpha(x_1, \ldots, x_n)$
B	β $\mathbf{since}_{=c}$ γ	$(\exists y)(y = c \wedge r^j_\alpha(x_1, \ldots, x_n, y))$
C	β $\mathbf{since}_{<c}$ γ	$(\exists y)(0 \leq y < c \wedge r^j_\alpha(x_1, \ldots, x_n, y))$
D	β $\mathbf{since}_{>c}$ γ	$s^j_\alpha(x_1, \ldots, x_n)$

Figure 3: Definitions of $\alpha^{i,j}$ for real-time formulas

Notation:

- $(x_1, \ldots, x_n)$ is the variable vector of α.
- the variable y is different from $x_1, \ldots, x_n$.

Notice that as before the instances of auxiliary relations in the extended state $\hat{D}_{i+1}$ are defined using first-order formulas from the instances of the auxiliary relations in the extended state $\hat{D}_i$ and the instances of database relations in D_{i+1}. The defining formulas involve now linear arithmetic constraints.

For *Past Metric FOTL*, properties analogous to Theorem 3.1 and Corollaries 3.1 and 3.2 hold, guaranteeing the correctness of the integrity checking method. It is essential for the correctness of the method that the clock be increasing, i.e., $clock(i+1) \geq clock(i)$ for all i. For PFO-domain-independent constraints, the method produces also a bounded history encoding. The last fact is easy to see because the distance attribute can assume only finitely many values for a given set of integrity constraints. Here, we are relying on the fact that the clock assumes integer values. The existence of bounded history encoding cannot be guaranteed for clocks with values coming from a dense domain.

	α	$r_\alpha^{i+1}(x_1,\ldots,x_n)$
E	$\bullet_{\sim c}\beta$	$\beta^{i,i} \wedge clock(i+1) \sim clock(i)+c$
	α	$r_\alpha^{i+1}(x_1,\ldots,x_n,z)$
F	β **since**$_{\sim c}$ γ	$(0 \le z \le c \wedge$ $(\beta^{i,i+1} \wedge \gamma^{i,i} \wedge z = clock(i+1) - clock(i)$ $\vee \beta^{i,i+1} \wedge (\exists y)(r_\alpha^i(x_1,\ldots,x_n,y) \wedge z = y + clock(i+1) - clock(i))))$
	α	$s_\alpha^{i+1}(x_1,\ldots,x_n)$
G	β **since**$_{>c}$ γ	$(\beta^{i,i+1} \wedge \gamma^{i,i} \wedge clock(i+1) - clock(i) > c$ $\vee \beta^{i,i+1} \wedge (\exists y)(r_\alpha^i(x_1,\ldots,x_n,y) \wedge y + clock(i+1) - clock(i) > c)$ $\vee \beta^{i,i+1} \wedge s_\alpha^i(x_1,\ldots,x_n))$

Figure 4: Auxiliary relations for real-time connectives

Notation:

- $(x_1,\ldots,x_n)$ is the variable vector of α.
- the variables y and z are different from $x_1,\ldots,x_n$.
- the variable z stands for the distance attribute of the appropriate auxiliary relation.

4.3 Real-time constraint violations

We have mentioned that real-time integrity constraints may be violated just by the flow of physical time, without any database updates. To model this situation we need to allow *clock-only* transactions whose only effect is an increase in the clock value. In fact, it is advisable to distinguish between such transactions and *database-only* transactions that do not increase the clock. The latter can be aborted while the former cannot (the flow of time cannot be reversed).

Consequently, if a real-time constraint is violated by the flow of time, the violation is irreparable. This suggests the need to make temporal integrity checking more flexible. The approach of using rules (triggers) of the form

if *Condition*
then *Action*

looks particularly promising. The *Condition* can be any *Past MTL* formula and the *Action* – a corrective or compensating action. (We do not consider here the issue of the right action language: any such language providing a basic repertoire of relation updates, e.g., [69], fits well within our framework.)

In example 4.1, instead of the constraint C_7 we may have the trigger:

if $(\neg return(x)$ **since**$_{>7}$ $(recall(x) \wedge book(x)))$
then $insert(overdue(x))$.

In this way the integrity violation is detected and recorded. The trigger fires only once (see the discussion of triggers at the end of section 3).

4.4 Automatic data aging

Real-time constucts introduced above provide not only a way of modeling physical time but also help in addressing the issue of *data aging.* Data that is too old becomes often irrelevant for integrity checking. In the naive approach where the whole database history is stored, the solution to this problem is crude but simple: the irrelevant data is simply deleted and thus made unavailable for integrity checking. In our approach, only an encoding of the history is stored. The same encoding may be produced by different histories and thus truncating a history does not necessarily determine a unique update to the encoding. Rebuilding the whole encoding from scratch is not a practical option because past states may no longer be available. However, if real-time constructs are used, deletion of irrelevant data happens automatically.

Example 4.3 Consider an auxiliary relation r_α for $\alpha = \blacklozenge\, p(x)$. This relation contains all values x for which $p(x)$ was true at any time in the past. Clearly, this relation never shrinks and if the database history is sufficiently long, the size of r_α may exceed available storage space. However, if we reformulate α as $\blacklozenge_{<c}\, p(x)$, only the values from the states at most c time instants in the past will be stored. Older values will automatically disappear during the recomputation of auxiliary relations.

Clock values are not even necessary for automatic data aging. One can identify the flow of time with the sequence of committing transactions and assume that

$$clock(i+1) = clock(i) + 1$$

for all i. Then, the definitions in Figures 3 and 4 will be considerably simplified.

5 General framework for constraint satisfaction

In this section we define a general framework for temporal constraint satisfaction. This will make it possible to relate our approach to other work in this area in section 6.

We consider now temporal integrity constraints defined in full *FOTL*, not only in its past fragment. Moreover, we define the semantics of such constraints with respect to temporal databases with infinitely many states.

5.1 Full *FOTL*

The language of full *FOTL* contains, in addition to the connectives defined in section 2, the following future-tense temporal connectives: $\bigcirc$ and **until** . *FOTL* formulas with future-tense connectives but not past-tense ones will be called *future formulas*; those with past-tense connectives but not future-tense ones – *past formulas*.

An *infinite-time temporal database* is a sequence

$$D = (D_0, D_1, D_2, \ldots)$$

of first-order structures over the same vocabulary and with the same universe $|D| = |D_0| = |D_1| = \ldots$. As before, we refer to the structure D_i as to the *database state* at time instant i and assume that a constant symbol c has the same interpretation in each database state (i.e. $c^{D_i} = c^{D_0}$, for each i) which we shall denote simply by c^D.

Let $D = (D_0, D_1, D_2, \ldots)$ be an infinite-time temporal database. The semantics of future-tense temporal connectives is defined in the following way:

- $D, v, i \models \bigcirc A$ iff $D, v, i+1 \models A$,
- $D, v, i \models A$ **until** B iff for some $j > i$, $D, v, j \models B$ and for all k, $i \leq k < j$, $D, v, k \models A$,

The semantics of othe syntactic constructs of *FOTL* (including past-tense temporal connectives) is defined for infinite-time temporal databases in the same way as for finite-time databases.

The well-known temporal connectives $\Diamond$ (*"sometime in the future"*) and $\Box$ (*"always in the future"*) are defined in terms of **until** as follows:

$\Diamond A = True$ **until** A

$\Box A = \neg \Diamond \neg A.$

5.2 Temporal integrity checking

Temporal integrity constraints are imposed on the *current history* of a database, i.e., the sequence of states up to the current one. Such a history is of course a finite-time temporal database. However, the semantics of a temporal integrity constraint expressed in full *FOTL* is defined with respect to an infinite structure representing a possible future evolution of the database. The notion of *potential constraint satisfaction* reconciles those two views.

For a closed *FOTL* formula C, the *set* $Pref(C)$ *of prefixes of models of* C is defined as follows:

> $D \in Pref(C)$ if there is a model $D^* = (D_0, \ldots, D_k, \ldots)$ of C such that $D = (D_0, \ldots, D_k)$.

A constraint C is *potentially satisfied* at instant k if the current history $(D_0, \ldots, D_k) \in Pref(C)$. In other words, a constraint is potentially satisfied after an update if the history ending in the state resulting from the update has an (infinite) extension to a model of the constraint. Thus, the current history provides a *partial model* of the constraint.

We consider infinite-time databases here because we want to capture the intuition that the database should be *infinitely updatable*. Although every practical database has only a finite lifetime, infinity provides a convenient mathematical abstraction. The situation is similar to the one in concurrent programming where nonterminating programs are studied despite the fact that every practical program eventually terminates.

The notion of potential constraint satisfaction provides a different perspective on the issue of constraint satisfaction for *Past FOTL* formulas. In section 3, we defined the latter as the satisfaction of the constraint formula in every prefix of the current (finite) database history. Now we additionally require that the prefix be extendible to an infinite model of the constraint. Potential constraint satisfaction is the stronger of those two notions as evidenced by the following proposition.

Proposition 5.1 *The following properties hold:*

- *$Pref(C) \subseteq Sat(C)$ for any closed Past FOTL formula C.*
- *there is a closed Past FOTL formula C_0 and a finite-time temporal database D such that $D \in Sat(C_0)$ and $D \notin Pref(C_0)$.*

Proof: Assume that $D \in Pref(C)$ and $D = (D_0, \ldots, D_i)$. Thus, there is a temporal database $D^* = (D_0, \ldots, D_i, \ldots)$ such that $D^* \models C$. Consequently, $D^*, j \models C$ for all $0 \leq j \leq i$ and thus $D, j \models C$ for all $0 \leq j \leq i$ because for all those states D and D^* are identical. Thus $D \in Sat(C)$. For the second part, consider $C_0 \equiv \neg \bullet True$ and $D = (D_0)$, $D_0 = \emptyset$. $\Box$

As we have demonstrated in section 3 the notion of constraint satisfaction in finite histories is practical and can be efficiently implemented using bounded history encoding. The following proposition shows that this notion provides an *approximation* of potential constraint satisfaction.

Proposition 5.2 *Let $D = (D_0, \ldots, D_i)$ and C be a closed Past FOTL formula. If $D \notin Pref(C)$ and all the proper prefixes of D are in $Pref(C)$, then for any extension $D^* = (D_0, \ldots, D_i, \ldots)$ of D, there exists k, $k \geq i$, $(D_0, \ldots, D_i, \ldots, D_k) \notin Sat(C)$.*

Proof: Assume that $D \notin Pref(C)$. Thus, for every temporal database D^* that is an extension of D, D^* is not a model of C. Consequently, for every such database there is a $k \geq 0$ such that $D^*, k \not\models C$ and thus $(D_0, \ldots, D_k), k \not\models C$ (as C is a *Past FOTL* formula). Hence $(D_0, \ldots, D_k) \notin Sat(C)$. Thus, by Proposition 5.1 $(D_0, \ldots, D_k) \notin Pref(C)$ and thus $k \geq i$. $\Box$

The intuition behind this proposition is as follows: every violation of potential constraint satisfaction will be detected but not necessarily at the earliest possible instant. In most practical cases our integrity checking method catches all such violations without delay. However, this cannot be guaranteed for all *Past FOTL* formulas. Theorem 5.1 below implies that the delay in detecting integrity violations cannot be bounded by a recursive function.

Theorem 5.1 *There is a closed Past FOTL formula C_0 such that $Pref(C_0)$ is not recursively enumerable.*

Proof: See Appendix A. $\Box$

The set $Sat(C)$ constitutes the semantics of our integrity checking method. Thus different methods can be compared with respect to their violation detecting power: the smaller the set of histories they allow, the better the method. Of course, the least such a set is $Pref(C)$ itself.

We conclude this section by noting that the notion of potential constraint satisfaction is applicable not only to full *FOTL* but also to any temporal constraint language with model-theoretic semantics. For example, one can consider extending *FOTL* with fixpoint operators the generalize the propositional fixpoint operators [67].

6 Related work

6.1 Temporal logic

Propositional temporal logic has been widely used for the specification and verification of concurrent programs [24, 46, 47, 51]. Properties of temporal logic connectives on which the definitions in Figure 2 are based are well known. The idea of encoding the database history using bounded space corresponds to the idea of language recognition using a finite-state automaton. As pointed to us by Pierre Wolper, our construction is similar to the construction of a Büchi automaton corresponding to a temporal formula [66]. However, the latter was proposed only for the propositional case. We are not aware of a generalization of Büchi automata to the first-order case. Also, the implementation issues arising in the first-order case are new. Gabbay et al. [27] show that in the propositional case past-tense connectives do not increase the expressive power of temporal logic. It is an open question whether this property generalizes to the first-order case.

First-order temporal logic (*FOTL*) has received much less attention in the computer science community, mainly because the satisfiability of *FOTL* formulas is a highly undecidable problem [33]. *FOTL* has been proposed as an appropriate language for expressing *temporal queries* [64, 26, 17, 18]. The exact expressive power of *FOTL* as a query language is an open problem. Two-sorted first-order logic (in which one of the sorts is the set of integers with the standard ordering and every relation is extended with an additional, explicit temporal argument) is at least as expressive as *FOTL*. This follows from the definition of the semantics of temporal connectives. However, the containment in the other direction has not been proved or disproved. Kamp [38] established that propositional *TL* is equally expressive as monadic first-order logic. Many temporal query languages [60] are not based on logic at all but are still subsumed by *FOTL* in terms of expressive power, as shown by Clifford et al. [17, 18]. For example, TQuel [58] uses a set of interval relationships proposed in [3]. These relationships can all be expressed in *Past FOTL*. Expressing logical negation in nonlogical temporal query languages is often quite awkward (or restricted) and in this respect *FOTL* has a clear advantage. Some representative temporal query languages are studied from a formal point of view in a recent survey [13].

For efficient integrity checking using bounded history encoding the following property of *FOTL* is essential: Each subformula of a *FOTL* formula is associated with a single temporal context. This property makes possible associating auxiliary (nontemporal) relations with subformulas and updating those relations according to the flow of time. Two-sorted first-order logic does not have this property and thus it seems unlikely that it will prove useful as a practical temporal constraint language.

Future FOTL appears to be too expressive as an integrity constraint language. There are many *Future FOTL* formulas that are useless as temporal integrity constraints, because they cannot be violated by finite histories.

Example 6.1 Consider the formula $\Diamond \exists x\, p(x)$. This formula is potentially satisfied (see section 5 for the definition of potential constraint satisfaction) in every finite history.

Formally, temporal integrity constraints should express *safety properties* [51]. This issue is discussed at greater length in [15].

A comprehensive recent reference for temporal logic is [25].

6.2 Temporal integrity constraints

Recent research shows that *FOTL* is the preferred language for the specification of temporal integrity constraints (the following list is by no means complete): [10, 11, 12, 20, 32, 36, 42, 43, 44, 54, 56, 57].

Lipeck, Saake, and their students [32, 36, 43, 44] propose several constraint checking methods for a subclass of future *FOTL* with very restricted quantification (called *biquantified* in [15]). Only implicit universal quantification is allowed outside temporal operators. This restriction makes it impossible to formulate constraints involving *quantifier alternation* outside temporal operators.

Example 6.2 Consider the constraint: *"a student can enroll in the graduate program only if he has taken an 800-level course"* expressed as the following *Past FOTL* formula:

$$(\forall s)(\exists y)(GradEnroll(s) \rightarrow Level(y, 800) \wedge \blacklozenge\, Take(s, y)).$$

This constraint cannot be expressed as a biquantified formula. Lipeck and Saake [44] introduce the notion of *potential constraint satisfaction* (under a different name). Subsequently, Chomicki and Niwiński [15] show that for the class of biquantified formulas potential constraint satisfaction is undecidable. Thus by necessity Lipeck and Saake's methods implement a weaker notion of constraint satisfaction. However, those methods provide bounded history encoding (without formally defining the concept). In the presence of infinite domains like integers (see the salary example in section 2), the approach of Lipeck and Saake may require the storage of infinitely many substitutions. Hülsmann and Saake [36] deal with this problem by introducing finite decriptions for infinite sets. A similar problem does not usually arise in our framework, because we allow explicit quantification outside temporal operators. Gertz and Lipeck [32] propose to compile biquantified formulas to STARBURST triggers. The approach is, however, different from that of Chomicki and Toman [16], as first-order logic is not used as an intermediate language. As far as the expressive power is concerned, Chomicki and Niwiński [15] show that *Past FOTL* and the class of biquantified formulas are incomparable. Schwiderski et al.[56] applies the method of [44] to a subset of *Past FOTL*.

Qian and Waldinger [54] also recognize the need for efficient checking of temporal constraints using bounded history encoding. That paper presents many compelling examples but no general method. De Castilho et al. [20] and Kung [42] deal mainly with the issue of consistency of temporal constraints with action specifications and do not study integrity checking in the presence of arbitrary updates.

Sistla and Wolfson [57] discuss triggers whose conditions are (past or future) temporal formulas. However, instead of standard first-order quantifiers, they use nonstandard *freeze* quantifiers (a freeze quantifier binds a variable to a single value or a relation instance in a given temporal context). The expressive power of the trigger language of [57] (termed SW from now on) depends on two factors: the query language of the underlying DBMS and the ability to test equality of relation instances. For the purpose of this comparison, we assume that the DBMS supports a first-order query language like relational calculus or algebra. In this case, SW is subsumed by two-sorted first-order logic. If relation instances cannot be tested for equality, SW with future-tense connectives only is the same as the class of negations of biquantified formulas of [44] and SW with past-tense connectives only is a proper subset of *Past FOTL*. However, if relation instances can be tested for equality, SW can express two-sorted first-order logic formulas (in which relations have an explicit temporal argument) which cannot be formulated in *FOTL* like

$$(\exists s)(s < t \wedge (\forall x)(p(s,x) \Leftrightarrow p(t,x))).$$

Even with this provision freeze quantifiers cannot express the standard first-order quantifiers and thus SW and *Past FOTL* are incomparable in terms of expressive power. SW is capable of expressing real-time conditions using a distinguished variable *time*. Sistla and Wolfson give in [57] an example of a real-time condition and a a PFO-domain-dependent *Past FOTL* formula (for the definition of PFO-domain-independence, see section 3) expressing it. This condition, called PAST-OVERLOAD, is formulated as follows: *"the traffic on link1 has doubled within ten minutes"*. The same condition can in fact be expressed as the following PFO-domain-independent *Past FOTL* formula

$$((\exists x,y,t') \blacklozenge (\mathit{traffic}(\mathit{link}1) \approx y \wedge \mathit{clock}(t')) \wedge \mathit{traffic}(\mathit{link}1) \approx x \wedge x \geq 2y \wedge \mathit{clock}(t) \wedge t \leq t'+10)$$

or, more succinctly, as a *Past Metric FOTL* formula (which is also PFO-domain-independent)

$$(\exists x,y)((\blacklozenge_{\leq 10}\ \mathit{traffic}(\mathit{link}1) \approx y) \wedge \mathit{traffic}(\mathit{link}1) \approx x \wedge x \geq 2y).$$

Therefore, it is not clear whether the language of [57] with past-tense connectives and without the ability to test equality of relation instances can in fact express trigger conditions that are not enforceable using the frameworks of sections 3 and 4. As far as the implementation is concerned, Sistla and Wolfson do not provide bounded history encoding and the amount of historical information kept in their method grows with the age of the database. This may not be a serious problem because they postulate to discard all this information after a trigger fires. The method of [57] implements a notion weaker than potential constraint satisfaction. In addition, Sistla and Wolfson discuss some issues specific to active databases like trigger activation and deactivation.

In the object-oriented context, Gehani et al. [30, 31] discuss triggers fired by event occurrences. Because triggers are associated with individual objects, only the histories of those objects are constrained. There is no notion of a global database history. Gehani et al. [30, 31] introduce a language for specifying composite events which is equivalent to regular expressions. Events have attributes but the language does not allow quantification. So it corresponds to *quantifier-free Past FOTL*, except for the fact that the temporal connectives used are slightly more powerful than those of *Past FOTL*. This follows from a well known fact [51] that propositional *TL* has the same expressive power as star-free regular expressions (a proper subset of regular expressions). Gehani et al. [30, 31] propose to implement their event language by compiling it into finite automata. This provides an analogue of bounded history encoding in an object-oriented framework.

6.3 Real-time properties

A great many formalisms have been proposed for the specification of real-time properties of programs [4, 5]. Many of those are various extensions of temporal logic. They are usually restricted, however, to the propositional case. Propositional Metric Temporal Logic (with future- and past-tense connectives) was proposed and studied in [40, 41]. A first-order version of this logic was used in [23, 48] for software system specification.

We discuss now several possible approaches to the specification and evaluation of real-time constraints and conditions, distinct from the one presented in section 4. First, are the real-time connectives necessary at all? Real-time conditions can be expressed in *Past FOTL* without any special real-time constructs, provided formulas can refer to the special *clock* predicate (Alternatively, the logic can use a special nonrigid variable to refer to time [34].) The resulting logic does not, however, seem to admit bounded history encoding. Below, we analyze *Past FOTL* with the *clock* predicate but similar comments apply to *FOTL* with restricted quantification of [36, 44], the first-order version of Explicit Clock Temporal Logic [34], and the trigger language of [57].

For every condition expressed in *Past FOTL* with the *clock* predicate, one can derive the definitions of appropriate auxiliary relations using the definitions in Figure 2.

Example 6.3 Consider the following *Past Metric FOTL* condition:

$$(\blacklozenge_{>5}\, p(x)) \Rightarrow \neg q(x).$$

It can be expressed in *Past FOTL* as:

$$\neg(\exists t_1, t_2)(q(x) \wedge clock(t_2) \wedge t_2 - t_1 > 5 \wedge (\blacklozenge\,(p(x) \wedge clock(t_1)))).$$

This formula has a single temporal subformula

$$\blacklozenge\,(p(x) \wedge clock(t_1))$$

with two free variables. It is easy to see that the auxiliary relation corresponding to this formula will have unbounded size, because it will store past information about p indexed by different clock values.

The difficulty stems from the fact that the clock, by nature, draws its values from an unbounded domain. We have been able to avoid this pitfall because *Past Metric FOTL* formulas talk about *relative*, not *absolute*, time which can be represented more succinctly. Moreover, we treat formulas of the form β **since**$_{>c}$ γ in a special way, ensuring that the distance attribute can assume only finitely many values.

The second option is to use *Past Metric FOTL* as syntactic sugar and translate it to *Past FOTL* [68]. Nassor and Vidal-Naquet [49] shows such a translation in the propositional case. The main idea is to introduce a propositional variable *tick* that models the increment of the clock. In the states that result from a database transaction (c.f., section 4), *tick* is false, while it is true in the states resulting from a clock transaction. Any increase in the clock value can be modeled as a sequence of unit increments. The advantage of this approach is that the implementation of condition evaluation does not have to account for the real-time connectives. On the other hand, the translation of a *Past Metric FOTL* formula $\alpha = \beta$ **since**$_{\sim c}$ γ to *Past FOTL* multiplies the number of temporal subformulas of α by $O(c)$ leading to the analogous increase in the number of materialized views. Thus it is impractical for any but small values of c. In the approach presented in section 4 the number of materialized views is at most twice that of temporal subformulas of a real-time condition and does not depend on the value of c.

The specification and evaluation of real-time conditions (or constraints) addresses just one aspect of real-time databases. Equally important is guaranteeing real-time response using appropriate scheduling and transaction processing algorithms [55].

7 Conclusions and future work

We have presented a complete framework for efficient checking of temporal integrity constraints formulated as *Past FOTL* formulas. We have analyzed the computational cost of our method and showed how it can be implemented using existing database technology. We also discussed an extension of the basic framework to the case of real-time constraints and conditions.

Here, we sketch several possible extensions of this work.

7.1 Constraint evolution

It is natural to allow the set of integrity constraints to change over time. In such a case, there are several possible semantics. One could require that a constraint can only be added if the entire database history satisfies it. This will make it necessary to store the entire history and also compute from scratch all the past instances of auxiliary relations associated with the new constraint, which may be too costly. However, constraint changes correspond to policy changes and are probably much less frequent than database updates. Therefore, they don't have to be processed fast and the history does not have to be stored on-line [2]. Moreover, this semantics is probably unsuitable for most applications, because it would make most policy changes (expresssed as an addition of a new constraint) impossible.

Another semantics, admittedly more natural, allows the addition of a new constraint only if the current state of the database treated as the *first* state with no history satisfies the constraint (the constraint holds "from now on"). This semantics is easily implementable using our method. It is enough to distinguish auxiliary relations corresponding to identical subformulas in different constraints by, for example, labeling them with the appropriate constraint identifiers. However, the second approach may lead to some semantic problems, absent from the first one.

Example 7.1 Consider the constraints $C_1 = (\neg \bullet True \Rightarrow p)$ and $C_2 = (\neg \bullet True \Rightarrow \neg p)$. They are clearly inconsistent. However, if C_1 holds from the first state of the database on and C_2 from the second state on, this inconsistency will not be discovered, as C_1 is trivially satisfied from the second state on. Similarly, consider $C_3 = (\bullet True \Rightarrow p)$. If C_3 holds from the first state of the database on and C_2 from the second state on, an inconsistency will be discovered in the second state, although the constraints are consistent.

These semantic difficulties do not appear if the constraints are logically independent. Note finally that constraint deletions do not require the introduction of new auxiliary relations and can therefore be easily handled by our method under both semantics.

7.2 Language extensions

Another direction for future work concerns addressing the expressive inadequacies of temporal logic. For example, temporal logic is unable to express even the simplest forms of temporal aggregation like counting modulo 2 [70]. Various kinds of aggregation (`count`, `sum`, `average`) are often used in temporal constraints or conditions. For example, consider the constraint *"the number of orders accepted that have not been fulfilled should not be greater than Max"*. An adequate language for expressing such constraints has yet to be designed. Some aggregation operators like `max` can in fact be expressed in *Past FOTL*. For those, it may be worthwhile to provide special entries in the tables in Figure 2 in order to optimize the evaluation of constraints that use them.

The relations mentioned in the integrity constraints may be defined intensionally using views or rules (even recursive rules). The constraint checking method described in section 3 remains the same. If the intensional definitions are function-free and range-restricted, the method still provides bounded history encoding.

It is more interesting to consider the relationship between the inductive definitions of auxiliary relations and some extensions of *Datalog*.

Example 7.2 The definitions from Example 3.5 can be expressed as the following rules:

$$r_{\alpha_1}(T+1, X) \leftarrow r_{\alpha_1}(T, X), \neg \mathit{reinstated}(T+1, X).$$
$$r_{\alpha_1}(T+1, X) \leftarrow \mathit{dropout}(T, X), \neg \mathit{reinstated}(T+1, X).$$

In general, all the necessary inductive definitions can be written as rules in $Datalog_{1S}$[14, 7], an extension of *Datalog* allowing one unary function symbol in one argument of predicates, augmented with stratified negation [6]. This language can in fact express properties not expressible in *FOTL* like counting modulo 2. However, its implementation requires a more powerful, deductive DBMS.

7.3 Domain independence

In section 3 we pointed out that some *Past FOTL* constraints cannot be handled by our method because they lead to domain-dependent pure first-order formulas. A possible solution to this problem is to use *generalized relations* [39] that are sets of conjunctions of atomic formulas. However, no DBMS supports generalized relations yet and the implementation issues involved are quite formidable.

7.4 Implementation and performance evaluation

It would be very interesting to measure performance improvements in constraint evaluation offered by history encoding and analyze the tradeoffs arising in its different realizations. However, designing appropriate benchmarks or building an analytical model appear quite difficult and are left for future research. The following topics, outlined in section 3, also merit further study: tradeoffs in implementing the hybrid method proposed there, space-saving optimizations of various kinds.

7.5 Generalizations

From a more general point of view, our approach makes possible systematic *garbage collection* of historical databases. Only some historical information is kept around, the rest can be discarded (or moved to slower and cheaper storage). Consequently, we think that this approach can be generalized beyond historical databases and integrity constraints formulated in temporal logic.

8 Acknowledgments

The traffic example is due to Alex Brodsky. I am grateful to Jennifer Widom for stimulating discussions and timely comments, for making STARBURST available, and for giving me a copy of [40]. Thanks go to Damian Niwiński and David Toman for many conversations and useful comments. The anonymous reviewers are thanked for suggestions that helped to considerably improve the presentation, in particular for proposing the term "bounded history encoding".

References

[1] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases.* Addison-Wesley, 1995.

[2] Ahn, I., and Snodgrass, R. Partitioned Storage for Temporal Databases. *Information Systems 13*, 4 (1988), 369–391.

[3] Allen, J. Maintaining Knowledge about Temporal Intervals. *Commun. ACM 26*, 11 (November 1983), 832–843.

[4] Alur, R., and Henzinger, T. Logics and Models of Real-Time: A Survey. In *Real-Time: Theory in Practice*. Springer-Verlag, LNCS 600, 1992, pp. 74–106.

[5] Alur, R., and Henzinger, T. Real-Time Logics: Complexity and Expressiveness. *Information and Computation 104* (1993), 35–77.

[6] Apt, K., Blair, H., and Walker, A. Towards a Theory of Declarative Knowledge. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, 1988, pp. 89–148.

[7] Baudinet, M., Chomicki, J., and Wolper, P. Temporal Deductive Databases. In Tansel et al. [60], pp. 294–320.

[8] Bernstein, P., Blaustein, B., and Clarke, E. Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data. In *International Conference on Very Large Data Bases* (1980).

[9] Ceri, S., and Widom, J. Deriving Production Rules for Incremental View Maintenance. In *International Conference on Very Large Data Bases* (1991).

[10] Chomicki, J. History-less Checking of Dynamic Integrity Constraints. In *IEEE International Conference on Data Engineering* (Phoenix, Arizona, February 1992).

[11] Chomicki, J. Real-Time Integrity Constraints. In *ACM Symposium on Principles of Database Systems* (San Diego, California, June 1992).

[12] Chomicki, J. Temporal Integrity Constraints in Relational Databases. *IEEE Data Engineering Bulletin 17*, 2 (June 1994), 33–37.

[13] Chomicki, J. Temporal Query Languages: A Survey. In *Temporal Logic, First International Conference* (1994), D. Gabbay and H. Ohlbach, Eds., Springer-Verlag, LNAI 827, pp. 506–534.

[14] Chomicki, J., and Imieliński, T. Temporal Deductive Databases and Infinite Objects. In *ACM Symposium on Principles of Database Systems* (Austin, Texas, March 1988).

[15] Chomicki, J., and Niwinski, D. On the Feasibility of Checking Temporal Integrity Constraints. In *ACM Symposium on Principles of Database Systems* (Washington, D.C., May 1993). Full version to appear in Journal of Computer and System Sciences.

[16] Chomicki, J., and Toman, D. Implementing Temporal Integrity Constraints Using an Active DBMS. *IEEE Transactions on Knowledge and Data Engineering* (August 1995). To appear in the special section on Temporal and Real-Time Databases. Early version in Proc. 4th IEEE International Workshop on Research Issues in Data Engineering: Active Database Systems, Houston, Texas, February 1994.

[17] Clifford, J., Croker, A., and Tuzhilin, A. On the Completeness of Query Languages for Grouped and Ungrouped Historical Data Models. In Tansel et al. [60], pp. 496–533.

[18] Clifford, J., Croker, A., and Tuzhilin, A. On Completeness of Historical Relational Query Languages. *ACM Transactions on Database Systems 19*, 1 (March 1994), 64–116.

[19] Dayal, U., et al. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD Record 17*, 1 (March 1988), 51–70.

[20] de Castilho, J., Casanova, M., and Furtado, A. A Temporal Framework for Database Specifications. In *International Conference on Very Large Data Bases* (1982), pp. 280–291.

[21] Demolombe, R. Syntactical Characterization of a Subset of Domain Independent Formulas. *Journal of the ACM 39*, 1 (January 1992), 71–94.

[22] Di Paola, R. The Recursive Unsolvability of the Decision Problem for the Class of Definite Formulas. *Journal of the ACM 16*, 2 (1969), 324–327.

[23] Dubois, E., Hagelstein, J., and Rifaut, A. Formal Requirements Engineering with ERAE. *Philips Journal of Research 43* (1988), 393–414.

[24] Emerson, E. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. B. Elsevier/MIT Press, 1990, ch. 16, pp. 995–1072.

[25] Gabbay, D., Hodkinson, I., and Reynolds, M. *Temporal Logic: Mathematical Foundations and Computational Aspects.* Oxford University Press, 1994.

[26] Gabbay, D., and McBrien, P. Temporal Logic and Historical Databases. In *International Conference on Very Large Data Bases* (1991).

[27] Gabbay, D., Pnueli, A., Shelah, S., and Stavi, S. On the Temporal Analysis of Fairness. In *ACM Symposium on Principles of Programming Languages* (1980).

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

[29] Garson, J. Quantification in Modal Logic. In *Handbook of Philosophical Logic*, D. Gabbay and F. Guenthner, Eds. D. Reidel, 1984, ch. II.5, pp. 249–307.

[30] Gehani, N., Jagadish, H., and Shmueli, O. Composite Event Specification in Active Databases: Model & Implementation. In *International Conference on Very Large Data Bases* (1992), pp. 327–338.

[31] Gehani, N., Jagadish, H., and Shmueli, O. Event Specification in an Active Object-Oriented Database. In *ACM SIGMOD International Conference on Management of Data* (1992), pp. 81–90.

[32] Gertz, M., and Lipeck, U. Deriving Integrity Maintaining Triggers from Transition Graphs. In *IEEE International Conference on Data Engineering* (1993).

[33] Harel, D. Recurring Dominoes: Making the Highly Undecidable Highly Understandable. *Annals of Discrete Mathematics 24* (1985), 51–71.

[34] Harel, E., Lichtenstein, O., and Pnueli, A. Explicit Clock Temporal Logic. In *IEEE Symposium on Logic in Computer Science* (1990).

[35] Hsu, A., and Imieliński, T. Integrity Checking for Multiple Updates. In *ACM SIGMOD International Conference on Management of Data* (1985).

[36] Hülsmann, K., and Saake, G. Theoretical Foundations of Handling Large Substitution Sets in Temporal Integrity Monitoring. *Acta Informatica 28*, 4 (1991).

[37] International Organization for Standardization. Database Language SQL. ISO/IEC 9075:1992, 1992.

[38] Kamp, J. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[39] Kanellakis, P., Kuper, G., and Revesz, P. Constraint Query Languages. In *ACM Symposium on Principles of Database Systems* (Nashville, Tennessee, April 1990), pp. 299–313. To appear in Journal of Computer and System Sciences.

[40] Koymans, R. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. PhD thesis, Technische Universiteit Eindhoven, 1989.

[41] Koymans, R. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems 2*, 4 (1990), 255–299.

[42] Kung, C. On Verification of Database Temporal Constraints. In *ACM SIGMOD International Conference on Management of Data* (Austin, Texas, 1985), pp. 169–179.

[43] Lipeck, U., Gertz, M., and Saake, G. Transitional Monitoring of Dynamic Integrity Constraints. *IEEE Data Engineering Bulletin* (June 1994).

[44] Lipeck, U., and Saake, G. Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems 12*, 3 (1987), 255–269.

[45] Lohman, G., Lindsay, B., Pirahesh, H., and Schiefer, K. Extensions to Starburst: Objects, Types, Functions and Rules. *Commun. ACM 34*, 10 (October 1991), 94–109.

[46] Manna, Z., and Pnueli, A. The Anchored Version of the Temporal Framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, J. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds. Springer-Verlag, LNCS 354, 1989, pp. 201–284.

[47] Manna, Z., and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[48] Morzenti, A., Mandrioli, D., and Ghezzi, C. A Model Parametric Real-Time Logic. *ACM Trans. Program. Lang. Syst. 14*, 4 (October 1992), 521–574.

[49] Nassor, E., and Vidal-Naquet, G. Suitability of the Propositional Temporal Logic to Express Properties of Real-Time Systems. In *Symposium on Theoretical Aspects of Computer Science* (1992).

[50] Nicolas, J.-M. Logic for Improving Integrity Checking in Relational Databases. *Acta Informatica 18*, 3 (1982), 227–253.

[51] Pnueli, A. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends. In *Current Trends in Concurrency.* Springer-Verlag, LNCS 224, 1986.

[52] Qian, X. An Effective Method for Integrity Constraint Reformulation. In *IEEE International Conference on Data Engineering* (1988).

[53] Qian, X., and Smith, D. Integrity Constraint Reformulation for Efficient Validation. In *International Conference on Very Large Data Bases* (1987).

[54] Qian, X., and Waldinger, R. A Transaction Logic for Database Specification. In *ACM SIGMOD International Conference on Management of Data* (1988), pp. 243–250.

[55] Ramamritham, K. Real-Time Databases. *Distributed and Parallel Databases 1* (1993), 199–226.

[56] Schwiderski, S., Hartmann, T., and Saake, G. Monitoring Temporal Preconditions in a Behaviour Oriented Object Model. *Data and Knowledge Engineering 14* (1994), 143–186.

[57] Sistla, A., and Wolfson, O. Temporal Triggers in Active Databases. *IEEE Transactions on Knowledge and Data Engineering* (1995). To appear.

[58] Snodgrass, R. The Temporal Query Language TQuel. *ACM Transactions on Database Systems 12*, 2 (June 1987), 247–298.

[59] Snodgrass, R., and Ahn, I. Temporal Databases. *IEEE Computer 19*, 9 (1986).

[60] Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., and Snodgrass, R., Eds. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[61] Topor, R. Domain-Independent Formulas and Databases. *Theoretical Computer Science 52* (1987), 281–306.

[62] Tuzhilin, A. Temporal Logic as a Simulation Language. In *International Conference on Artificial Intelligence and Simulation* (1991).

[63] Tuzhilin, A. Applications of Temporal Databases to Knowledge-based Simulations. In Tansel et al. [60], pp. 580–593.

[64] Tuzhilin, A., and Clifford, J. A Temporal Relational Algebra as a Basis for Temporal Relational Completeness. In *International Conference on Very Large Data Bases* (1990).

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

[66] Vardi, M., and Wolper, P. An Automata-Theoretic Approach to Automatic Program Verification. In *IEEE Symposium on Logic in Computer Science* (1986).

[67] Vardi, M. Y. A Temporal Fixpoint Calculus. In *ACM Symposium on Principles of Programming Languages* (1988).

[68] Vardi, M. Y., June 1992. Personal communication.

[69] Widom, J., Cochrane, R., and Lindsay, B. Implementing Set-Oriented Production Rules as an Extension to Starburst. In *International Conference on Very Large Data Bases* (1991).

[70] Wolper, P. Temporal Logic Can Be More Expressive. *Information and Control 56* (1983), 72–99.

A Potential constraint satisfaction is not r.e.

We prove here Theorem 5.1 by constructing a *Past FOTL* formula C_0 such that $Pref(C_0)$ is not recursively enumerable. We adapt the construction that we gave elsewhere [15].

We use a temporal database to encode computations of a deterministic Turing machine M. The alphabet Σ of the tape symbols used by M includes the input alphabet $\{0, 1\}$ and the special blank symbol B. Let Q be the set of states of M including the initial state q_0 and the final state q_f. We encode configurations of M (i.e., sequences of the form $\alpha q \beta B^\omega$ where $\alpha, \beta \in \Sigma^*$ and $q \in Q$) by finite database states in the following way: the vocabulary will include a monadic symbol P_q for each state $q \in Q$ and a monadic relation symbol P_σ for each $\sigma \in \Sigma$ except for B. It will also be convenient to use the notation $P_B(x)$ as an abbreviation for $\bigwedge_{z \in Q \cup \Sigma - \{B\}} \neg P_z(x)$.

The *Past FOTL* formula C_0 forces a finite-time temporal database $D = (D_0, \ldots, D_i)$ to encode an unfinished computation of M.

We use a number of additional relation symbols:

- $\mathit{Zero}(x)$ to denote that x is the least element of the universe.
- $\mathit{succ}(x, y)$ to denote that y is one more than x.
- $x < y$ ($x \leq y$) to denote that x is less than (resp. less than or equal to) y.

All those symbols can be eliminated because they can be expressed using temporal connectives [29, 15]. (The construction used there involves future formulas but it can be easily adapted to past formulas.) Thus the result holds also in the absence of such symbols in the vocabulary (in fact, equality suffices).

The desired *Past FOTL* formula C_0 is a conjunction of formulas expressing the following properties:

1. for any element x of the universe, in any past database state (i.e. *always in the past* in the temporal sense), at most one of the monadic relations $P_q, q \in Q$, $P_\sigma, \sigma \in \Sigma - \{B\}$ is true about x.

2. the initial database state encodes some initial configuration of M. More specifically, for any elements x, y of the universe, if $\mathit{Zero}(x)$ then $P_{q_0}(x)$, otherwise, if $x \leq y$ then either no monadic relation is true about either x or y, or $P_0 \vee P_1$ is true about both x and y, or $P_0 \vee P_1$ is true about x and no monadic relation is true about y.

3. subsequent database states encode subsequent configurations. More specifically, in any past database state and for any elements x, y, z of the universe, if $P_{z_1}(x), P_{z_2}(y)$ and $P_{z_3}(z)$ hold in the previous state where $z_1, z_2, z_3 \in Q \cup \Sigma$, and $\mathit{succ}(x, y)$ and $\mathit{succ}(y, z)$ (remember the convention about P_B), then *in the current state* $P_{z'}(y)$ holds, where z' is the symbol determined by z_1, z_2, z_3 according to the transition table of M. A similar statement must be added for the case when i represents the leftmost cell of the tape (i.e., $\mathit{Zero}(i)$ holds).

4. in any past database state and for all x in the universe, $\neg P_{q_f}(x)$ holds.

Now $D_0 \in \mathit{Pref}(C_0)$ iff the computation of M does not terminate with the input encoded by D_0. By the properties of Turing machines, we can choose M in such a way that the latter problem (which is the complement of the halting problem) is not recursively enumerable.