Temporal Integrity Constraints in Relational Databases *

Jan Chomicki

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

1 Introduction

As historical databases become more widely used in practice [9], the need arises to address database integrity issues that are specific to such databases. In particular, it is necessary to generalize the standard notion of *static integrity* (involving single database states) to *temporal integrity* (involving sequences of database states). Temporal integrity constraints are used to support *time-related* policies, e.g., *"once a student drops out of the Ph.D. program, she should not be readmitted"*. In this paper, we survey our work on *temporal integrity constraints* in relational databases, done in the context of First-Order Temporal Logic (called *FOTL* hereafter). We have addressed both foundational and practical issues.

2 First-order temporal logic

Syntax. In addition to the constructs of first-order logic, *FOTL* has a number of temporal connectives: past ($\bullet$ and **since**) and future ($\circ$ and **until**). *Past* formulas contain only past temporal connectives, *future* formulas – only future ones. An *integrity constraint* is a closed *FOTL* formula.

Semantics. We assume that time is isomorphic to natural numbers, i.e., time instants form an infinite sequence $0, 1, 2, \ldots$ A corresponding infinite sequence $D = (D_0, D_1, D_2, \ldots)$ of *database states* with the same schema is called a *temporal database.* Every state contains a relation (set of tuples) for every relation symbol in the schema. Thus every state is just a relational database instance.

The truth value of a closed *FOTL* formula at a given time i in a temporal database D is defined in the following way:

1. an atomic formula A is true at time i if A is true in D_i.
2. if the main connective of the formula is non-temporal, then the semantics is the standard first-order semantics. For example, $\neg A$ is true at time i if A is not true at time i.
3. $\bullet$ A is true at time i iff $i > 0$ and A is true at time $i-1$.
4. A **since** B is true at time i iff for some j, $0 \leq j < i$, B is true at time j, and for every k, $j < k \leq i$, A is true at time k.

*Research supported by NSF grant IRI-9110581.

5. $\bigcirc A$ is true at time i iff A is true at time $i+1$.

6. A **until** B is true at time i iff for some j, $j > i$, B is true at time j and for every k, $i \leq k < j$, A is true at time k.

A formula ϕ is satisfied in D (D is a *model* of ϕ) if ϕ is true in the *first* state of D. We also consider finite temporal databases, in which the satisfaction of past formulas is defined as above and future formulas are not considered. Although temporal integrity constraints are *FOTL* formulas, the above notion of formula *satisfaction* is distinct form the notion of integrity constraint *fulfillment* defined in Section 3. The temporal connectives $\blacklozenge$ (meaning *"sometime in the past"*) and $\blacksquare$ (*"always in the past"*) can be defined using **since**, while $\Diamond$ (*"sometime in the future"*) and $\Box$ (*"always in the future"*) can be defined using **until**. In temporal logic parlance we interpret constant symbols as *rigid* (having the same meaning in different states) and relation symbols as *flexible* (corresponding to different relations in different states). This interpretation seems well-suited to database applications.

We give now several examples of how *FOTL* can be used for the specification of temporal integrity constraints.

Example 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:

$$\Box \neg(\exists x)(\mathit{Dropout}(x) \wedge \Diamond \mathit{Admitted}(x))$$

or as the formula C_0:

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

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:

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

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

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

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.

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.

3 Temporal integrity

For an integrity constraint C, the *set* $\mathit{Pref}(C)$ *of prefixes of models of* C is defined as follows:

$\eta \in \mathit{Pref}(C)$ if there is a model $D = (D_0, \ldots, D_i, \ldots)$ of C such that $\eta = (D_0, \ldots, D_i)$.

A constraint C is *fulfilled* at time i if the current history $(D_0, \ldots, D_i) \in Pref(C)$. In other words, a constraint is fulfilled after an update if the history ending in the state resulting from the update has an (infinite) extension to a model of the constraint [7].

Unfortunately, even for restricted classes of *FOTL* formulas constraint fulfillment is in general not a computationally feasible notion. Thus, we have also proposed a weaker notion of *eventual* constraint fulfillment [3].

A set P of finite temporal databases is *a set of superprefixes* of C if it satisfies the following conditions:

1. *every constraint violation is eventually detected:* for all D, η, and i such that $D = (D_0, \ldots, D_i, \ldots)$, $\eta = (D_0, \ldots, D_i)$, $\eta \notin Pref(C)$ implies $\exists k$, $(D_0, \ldots, D_k) \notin P$ (k can be less, equal to, or greater than i).

2. *no false alarms are raised:* $Pref(C) \subseteq P$.

C is *eventually fulfilled* under a set of superprefixes P at time i if the current history $(D_0, \ldots, D_i) \in P$.

For the purposes of efficient implementation and computational complexity analysis several restricted classes of *FOTL* formulas have been identified. Reference [7] proposed the class of *biquantified formulas* (without using this specific term). Biquantified formulas allow only future temporal operators and restricted quantification in the following sense: the quantifiers can be either *external* (not in the scope of any temporal connective) or *internal* (no temporal connective in their scope). Moreover, all external quantifiers are universal. We have proposed in [1] the class of *conservative formulas* which are of the form $\Box\, \phi$ where ϕ is a past formula. The constraints C_0, C_1, and C_2 are all conservative formulas. (We should point out that only *safety* formulas [8], those whose violations can be detected by considering a finite prefix of a temporal database, make sense as integrity constraints.)

Our main theoretical results are as follows:

- for biquantified safety formulas with no internal quantifiers (called *universal*), constraint fulfillment is decidable (in exponential time)[4],

- for biquantified safety formulas with a single internal quantifier (existential or universal), constraint fulfillment is undecidable [4],

- for conservative formulas constraint fulfillment is undecidable [3] (every conservative formula is a safety formula).

As far as we know those are the only known characterizations of the computational complexity of checking temporal integrity constraints formulated in *FOTL*.

4 History-less constraint checking

For a conservative formula $\phi = \Box\, \psi$, define:

$$P_\phi = \{(D_0, \ldots, D_i) : \psi \text{ is true at } i \text{ in } (D_0, \ldots, D_i)\}.$$

The set P_ϕ is a set of superprefixes of ϕ [3] and provides the formal underpinning of our temporal integrity checking method. The membership in P_ϕ, in contrast to $Pref(\phi)$, can be efficiently checked.

The satisfaction of the past formula ψ in the current history can be checked in two distinct ways: a history-based or a history-less one. Essentially, in a history-based approach the whole current history is stored and used for checking constraints. In a history-less approach every state D_i is augmented with new *auxiliary* relations to form an *extended state* D'_i. Only the last extended state is stored and used for checking constraints. For a fixed set of constraints, the number of auxiliary relations should be fixed and their size should depend only on the set of domain values that appear in database relations. History-less constraint checking is possible at the price of forgetting some information contained in past databases states. Only the information relevant for checking constraints is kept. Consequently, general temporal queries can not be precisely answered in this approach because some of the past information may be missing.

Below we describe a specific history-less method [1, 3], applicable to integrity constraints that are closed conservative formulas. Considering every constraint $\phi = \Box\, \psi$ in turn, the extended database schema contains, in addition to database relations, an *auxiliary* relation r_α for every temporal subformula α of ψ, 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.

Example 2: Consider the constraint C_1. This constraint has a single temporal subformula $\alpha_1 = (\neg \mathit{Reinstated}(x)\ \textbf{since}\ \mathit{Dropout}(x))$. Thus the extended schema contains a unary auxiliary relation symbol r_{α_1} (α_1 has one free variable) such that:

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

This new relation has a very intuitive meaning: it contains all students that dropped out and have not been subsequently reinstated.

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

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

Clearly, the number of auxiliary relation symbols is fixed for a fixed set of constraints. Also, the size of auxiliary relations depends only on the set of values that appear in database relations. (It is in fact polynomial in the cardinality of this set.) Thus, our method is indeed *history-less*.

An auxiliary relation r_α at time i should contain exactly those domain values that make α true at i. Thus, auxiliary relations are defined *inductively*. First, their instances at time 0 are defined, and then it is shown how to obtain the instances at time $i+1$ from those at time i. The inductive definitions are automatically obtained from the syntax of a constraint [1, 3].

Example 3: Consider the constraint C_1 from Example 1. We obtain for the first state $r^0_{\alpha_1}(x)$ which is identical to *False*, and 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 extended 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.

We have built a prototype implementation of our method [10]. The constraints are compiled into a set of STARBURST rules [11] that update 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.

5 Related and further work

A framework similar to ours, based on the notion of a *relational automaton*, was recently proposed in [6] to overcome the limitations of the *composite events* model of [5]. Reference [6] addresses also the issue of incremental evaluation and extends the language of past formulas with a limited form of recursion. Their ideas can be easily incorporated into our framework. To handle the notion of a *clock*, it is necessary to extend *FOTL* with *real-time* constructs. Such an extension, implementable using the approach described in Section 4, was proposed in [2].

For temporal triggers of the form "**if** C **then** A", trigger *firing* can be defined as a notion dual to constraint fulfillment [3]. The issue of controlling such firings and the operational semantics of real-time triggers require, however, further study.

Another important issue is *temporal aggregation*. For example, the constraint *"the number of submitted orders that have not been processed should not exceed K"* is expressible as a different *FOTL* formula for different values of K. It would be better to have a single formula with K as a parameter. Moreover, other aggregation operators like *sum* or *average* may also be useful. One possible solution is to design a temporal extension of SQL and try to generalize the history-less method of Section 4 to that context.

Still another important issue concerns dealing with evolving sets of constraints. It is not even clear what kind of semantics is appropriate there. For example, should past database states satisfy integrity constraints introduced later?

References

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

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

[3] J. Chomicki. History-less Checking of Temporal Integrity Constraints. Technical Report TR-CS-93-16, Kansas State University, February 1994. Submitted.

[4] J. Chomicki and D. Niwinski. On the Feasibility of Checking Temporal Integrity Constraints. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Washington, D.C., May 1993.

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

[6] H.V. Jagadish, I.S. Mumick, and O. Shmueli. Incremental Evaluation of Sequence Queries. Manuscript, February 1994.

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

[8] A. Pnueli. 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.

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

[10] D. Toman and J. Chomicki. Implementing Temporal Integrity Constraints Using an Active DBMS. In *Proc. 4th IEEE International Workshop on Research Issues in Data Engineering: Active Database Systems*, Houston, Texas, February 1994.

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