Temporal Integrity Constraints

Jef Wijsen
University of Mons-Hainaut

SYNONYMS

Dynamic integrity constraints

DEFINITION

Temporal integrity constraints are integrity constraints formulated over temporal databases. They can express dynamic properties by referring to data valid at different time points. This is to be contrasted with databases that do not store past or future information: if integrity constraints can only refer to data valid at the current time, they can only express static properties. Languages for expressing temporal integrity constraints extend first-order logic with explicit timestamps or with temporal connectives. An important question is how to check and enforce such temporal integrity constraints efficiently.

HISTORICAL BACKGROUND

The use of first-order temporal logic for expressing temporal integrity constraints dates back to the early eighties (see for example [7]). Since the late eighties, progress has been made in the problem of checking temporal integrity [9, 2, 11] without having to store the entire database history. This entry deals with general temporal integrity constraints. The entry Temporal Dependencies deals with temporal variants of specific constraints, in particular with temporal extensions of functional dependencies.

SCIENTIFIC FUNDAMENTALS

Integrity constraints, whether they are temporal or not, are an important component of each database schema. They express properties that, ideally, must be satisfied by the stored data at all times. If a database satisfies all the integrity constraints, it is called *consistent*. Integrity constraints are commonly expressed in a declarative way using logic. Declarative integrity constraints generally do not specify how to keep the database consistent when data is inserted, deleted, and modified. An important task is to develop efficient procedures for checking and enforcing such constraints.

Temporal databases store past, current, and future information by associating time to database facts. An integrity constraint can be called "temporal" if it is expressed over a temporal database. By relating facts valid at different points in time, temporal integrity constraints can put restrictions on how the data can change over time. This is to be contrasted with databases that do not store past or future facts: if integrity constraints can only refer to a single database state, they cannot capture time-varying properties of data.

The following section deals with languages for expressing temporal integrity. Section 2 discusses techniques for checking and enforcing temporal integrity constraints.

1 Defining Temporal Integrity

While temporal integrity constraints can in principle be expressed as Boolean queries in whichever temporal query language, it turns out that temporal logic on timepoint-stamped data is the prevailing formalism for defining and studying temporal integrity. Section 1.1 illustrates several types of temporal constraints. Section 1.2 deals with several notions of temporal constraint satisfaction. Sections 1.3 and 1.4 contain considerations on expressiveness and on interval-stamped data.

1.1 Temporal, Transition, and Static Constraints

Since first-order logic is the lingua franca for expressing non-temporal integrity constraints, it is natural to express temporal integrity constraints in temporal extensions of first-order logic. Such temporalized logics refer to time either through variables with a type of time points, or by temporal modal operators, such as:

Operator	Meaning
$\bullet p$	Property p was true at the previous time instant.
$\blacklozenge p$	Property p was true sometime in the past.
$\bigcirc p$	Property p will be true at the next time instant.
$\lozenge p$	Property p will be true sometime in the future.

Satisfaction of such constraints by a temporal database can be checked if the question "Does (did/will) $R(a_1,\ldots,a_m)$ hold at t?" can be answered for any fact $R(a_1,\ldots,a_m)$ and time instant t. Alternatively, one can equip facts with time and ask: "Is $R(a_1,\ldots,a_m \mid t)$ true?". Thus, one can abstract from the concrete temporal database representation, which may well contain interval-stamped facts [6]. Every time point t gives rise to a *(database) state* containing all facts true at t.

The following examples assume a time granularity of days. The facts $\mathit{WorksFor}(\text{John Smith}, \text{Pulse})$ and $\mathit{Earns}(\text{John Smith}, 20\text{K})$, true on 10 August 2007, express that John worked for the Pulse project and earned a salary of 20K at that date. The alternative encoding is $\mathit{WorksFor}(\text{John Smith}, \text{Pulse} \mid \text{10 Aug 2007})$ and $\mathit{Earns}(\text{John Smith}, 20\text{K} \mid \text{10 Aug 2007})$.

Although temporal integrity constraints can generally refer to any number of database states, many natural constraints involve only one or two states. In particular, *transition constraints* only refer to the current and the previous state; *static constraints* refer only to the current state.

The first-order temporal logic (FOTL) formulas (1)–(4) hereafter illustrate different types of integrity constraints. The constraint *"An employee who is dropped from the Pulse project cannot work for that project later on"* can be formulated in FOTL as follows:

$$\neg\exists x(\mathit{WorksFor}(x,\text{Pulse}) \wedge \blacklozenge(\neg\mathit{WorksFor}(x,\text{Pulse}) \wedge \bullet\mathit{WorksFor}(x,\text{Pulse}))) \tag{1}$$

This constraint can be most easily understood by noticing that the subformula $\blacklozenge(\neg\mathit{WorksFor}(x,\text{Pulse}) \wedge \bullet\mathit{WorksFor}(x,\text{Pulse}))$ is true in the current database state for every employee x who was dropped from the Pulse project sometime in the past (this subformula will recur in the discussion of integrity checking later on). This constraint can be equivalently formulated in a two-sorted logic, using two temporal variables t_1 and t_2:

$$\neg\exists x\exists t_1\exists t_2((t_1 < t_2) \wedge \mathit{WorksFor}(x,\text{Pulse} \mid t_2) \wedge \neg\mathit{WorksFor}(x,\text{Pulse} \mid t_1) \wedge \mathit{WorksFor}(x,\text{Pulse} \mid t_1 - 1))$$

The constraint *"Today's salary cannot be less than yesterday's"* is a transition constraint, and can be formulated as follows:

$$\neg\exists x\exists y\exists z(\mathit{Earns}(x,y) \wedge \bullet(\mathit{Earns}(x,z) \wedge y < z)) \tag{2}$$

Finally, static constraints are illustrated. The most fundamental static constraints in the relational data model are primary keys and foreign keys. The constraint *"No employee has two salaries"* implies that employee names uniquely identify *Earns*-tuples in any database state; it corresponds to a standard primary key. The constraint *"Every employee in the WorksFor relation has a salary"* means that in any database state, the first column of *WorksFor* is a foreign key that references (the primary key of) *Earns*. These constraints can be formulated in FOTL without using temporal connectives:

$$\forall x\forall y\forall z(\mathit{Earns}(x,y) \wedge \mathit{Earns}(x,z) \rightarrow y = z) \tag{3}$$

$$\forall x\forall y(\mathit{WorksFor}(x,y) \rightarrow \exists z\mathit{Earns}(x,z)) \tag{4}$$

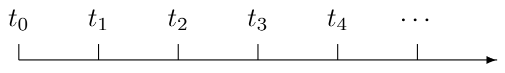
Formulas (3) and (4) show a nice thing about using FOTL for expressing temporal integrity: static constraints read as non-temporal constraints expressed in first-order logic. □

1.2 Different Notions of Consistency

The syntax and semantics of the temporal logic used in the previous section are defined next. Different notions of temporal constraint satisfaction are discussed.
Assume a countably infinite set **dom** of constants. In the following syntax, R is any relation name and each s_i is a variable or a constant:

$$C, C' ::= R(s_1, \ldots, s_m) \mid s_1 = s_2 \mid C \wedge C' \mid \neg C \mid \exists x(C) \mid \bigcirc C \mid \Diamond C \mid \bullet C \mid \blacklozenge C$$

The connectives $\bullet$ and $\blacklozenge$ are called *past operators*; $\bigcirc$ and $\Diamond$ are *future operators*. A *past formula* is a formula without future operators; a *future formula* is a formula without past operators. Other modal operators, like the past operator **since** and the future operator **until**, can be added to increase expressiveness; see the entry Temporal Logic in Database Query Languages. The set of time points is assumed to be infinite, discrete and linearly ordered with a smallest element. Thus, the time scale can be depicted as follows:



Discreteness of time is needed in the interpretation of the operators $\bigcirc$ and $\bullet$. Formulas are interpreted relative to an infinite sequence $H = \langle H_0, H_1, H_2, \ldots \rangle$, where each H_i is a finite set of facts formed from relation names and constants of **dom**. Intuitively, H_i contains the facts true at time t_i. Such a sequence H is called an *infinite (database) history* and each H_i a *(database) state*. The following rules define inductively what it means for a closed formula C to be *satisfied* by H, denoted $H \models_{\mathsf{inf}} C$.

$H, i \models_{\mathsf{inf}} R(a_1, \ldots, a_m)$	iff	$R(a_1, \ldots, a_m) \in H_i$
$H, i \models_{\mathsf{inf}} a_1 = a_2$	iff	$a_1 = a_2$
$H, i \models_{\mathsf{inf}} C \wedge C'$	iff	$H, i \models_{\mathsf{inf}} C$ and $H, i \models_{\mathsf{inf}} C'$
$H, i \models_{\mathsf{inf}} \neg C$	iff	not $H, i \models_{\mathsf{inf}} C$
$H, i \models_{\mathsf{inf}} \exists x(C)$	iff	$H, i \models_{\mathsf{inf}} C[x \mapsto a]$ for some $a \in$ **dom**, where $C[x \mapsto a]$ is obtained from C by replacing each free occurrence of x with a
$H, i \models_{\mathsf{inf}} \bullet C$	iff	$i > 0$ and $H, i-1 \models_{\mathsf{inf}} C$
$H, i \models_{\mathsf{inf}} \blacklozenge C$	iff	$H, j \models_{\mathsf{inf}} C$ for some j such that $0 \leq j < i$
$H, i \models_{\mathsf{inf}} \bigcirc C$	iff	$H, i+1 \models_{\mathsf{inf}} C$
$H, i \models_{\mathsf{inf}} \Diamond C$	iff	$H, j \models_{\mathsf{inf}} C$ for some $j > i$

Note that the truth of each subformula is expressed relative to a *single* "reference" time point i (along with H). This characteristic allows efficient techniques for integrity checking [5] and seems crucial to the success of temporal logic. Finally:

$$H \models_{\mathsf{inf}} C \quad \text{iff} \quad H, j \models_{\mathsf{inf}} C \text{ for each } j \geq 0$$

Consistency of infinite database histories is of theoretical interest. In practice, only a finite prefix of H will be known at any one time. Consider, for example, the situation where H_0 is the initial database state, and for each $i \geq 0$, the state H_{i+1} results from applying an update to H_i. Since every update can be followed by another one, there is no last state in this sequence. However, at any one time, only some *finite history* $\langle H_0, \ldots, H_n \rangle$ up to the most recent update is known, and it is of practical importance to detect constraint violations in such a finite history. It is reasonable to raise a constraint violation when the finite history obtained so far cannot possibly be extended to an infinite consistent history. For example, the constraints

$$\neg\exists x(\mathit{Hire}(x) \wedge \neg\Diamond(\mathit{Promote}(x) \wedge \Diamond\mathit{Retire}(x)))$$
$$\neg\exists x(\mathit{Retire}(x) \wedge \Diamond\mathit{Retire}(x))$$

express that all hired people are promoted before they retire, and that no one can retire twice. Then, the finite history $\langle\{\mathit{Hire}(\mathrm{Ed})\}, \{\mathit{Hire}(\mathrm{An})\}, \{\mathit{Retire}(\mathrm{Ed})\}\rangle$ is inconsistent, because of the absence of $\mathit{Promote}(\mathrm{Ed})$ in the second state. It is assumed here that the database history is append-only and that the past cannot be modified. Two different notions, denoted $\models_{\mathsf{pot}}$ and $\models_{\mathsf{fin}}$, of satisfaction for finite histories are as follows:

$\langle H_0, \ldots, H_n \rangle \models_{\mathsf{pot}} C$ if $\langle H_0, \ldots, H_n \rangle$ can be extended to an infinite history $H = \langle H_0, \ldots, H_n, H_{n+1}, \ldots \rangle$ such that $H \models_{\mathsf{inf}} C$.

2. $\langle H_0, \ldots, H_n \rangle \models_{\mathsf{fin}} C$ if $\langle H_0, \ldots, H_n \rangle$ can be extended to an infinite history $H = \langle H_0, \ldots, H_n, H_{n+1}, \ldots \rangle$ such that $H, i \models_{\mathsf{inf}} C$ for each $i \in \{0, 1, \ldots, n\}$.

Obviously, the first concept, called *potential satisfaction*, is stronger than the second one: $\langle H_0, \ldots, H_n \rangle \models_{\mathsf{pot}} C$ implies $\langle H_0, \ldots, H_n \rangle \models_{\mathsf{fin}} C$. Potential satisfaction is the more natural concept. However, Chomicki [2] shows how to construct a constraint C, using only past operators ($\bullet$ and $\blacklozenge$), for which $\models_{\mathsf{pot}}$ is undecidable. On the other hand, $\models_{\mathsf{fin}}$ is decidable for constraints C that use only past operators, because the extra states $H_{n+1}, H_{n+2}, \ldots$ do not matter in that case. It also seems that for most practical past formulas, $\models_{\mathsf{pot}}$ and $\models_{\mathsf{fin}}$ coincide [5]. Chomicki and Niwiński [3] define restricted classes of future formulas for which $\models_{\mathsf{pot}}$ is decidable.

1.3 Expressiveness of Temporal Constraints

The only assumption about time used in the constraints shown so far, is that the time domain is discrete and linearly ordered. Many temporal constraints that occur in practice need additional structure on the time domain, such as granularity. The constraint *"The salary of an employee cannot change within a month"* assumes a grouping of time instants into months. It can be expressed in a two-sorted temporal logic extended with a built-in predicate $\texttt{month}(t_1, t_2)$ which is true if t_1 and t_2 belong to the same month:

$$\neg\exists x \exists y_1 \exists y_2 \exists t_1 \exists t_2 (\texttt{month}(t_1, t_2) \wedge \mathit{Earns}(x, y_1 \mid t_1) \wedge \mathit{Earns}(x, y_2 \mid t_2) \wedge y_1 \neq y_2) \ .$$

Arithmetic on the time domain may be needed to capture constraints involving time distances, durations, and periodicity. For example, the time domain $0, 1, 2, \ldots$ may be partitioned into weeks by the predicate $\texttt{week}(t_1, t_2)$ defined by: $\texttt{week}(t_1, t_2)$ if $t_1 \backslash 7 = t_2 \backslash 7$, where $\backslash$ is the integer division operator. If $0 \leq t_2 - t_1 \leq 7$, then one can say that t_2 is *within a week from* t_1 (even though $\texttt{week}(t_1, t_2)$ may not hold). The entry Temporal Constraints discusses such arithmetic equalities and inequalities on the time domain.

Temporal databases may provide two temporal dimensions for valid time and transaction time. Valid time, used in the preceding examples, indicates when data is true in the real world. Transaction time records the history of the database itself. If both time dimensions are supported, then constraints can explicitly refer to both the history of the domain of discourse and the system's knowledge about that history [10]. Such types of constraints cannot be expressed in formalisms with only one notion of time.

Temporal logics have been extended in different ways to increase their expressive power. Such extensions include fixpoint operators and second-order quantification over sets of timepoints. On the other hand, several important problems, such as potential constraint satisfaction, are undecidable for FOTL and have motivated the study of syntactic restrictions to achieve decidability [3].

This entry focuses on temporal integrity of databases that use (temporal extensions of) the relational data model. Other data models have also been extended to deal with temporal integrity; time-based cardinality constraints in the Entity-Relationship model are an example.

1.4 Constraints on Interval-stamped Temporal Data

All constraints discussed so far make abstraction of the concrete representation of temporal data in a (relational) database. They only assume that the database can tell whether a given fact holds at a given point in time. The notion of finite database history (let alone infinite history) is an abstract concept: in practice, all information can be represented in a single database in which facts are timestamped by time intervals to indicate their period of validity. For example,

Emp	*Sal*	*FromTo*
John Smith	10K	[1 May 2007, 31 Dec 2007]
John Smith	11K	[1 Jan 2008, 31 Dec 2008]

Following [10], constraints over such interval-stamped relations can be expressed in first-order logic extended with a type for time intervals and with Allen's interval relations. The following constraint, stating that *"Salaries of employees cannot decrease,"* uses temporal variables i_1, i_2 that range over time intervals:

$$\forall x \forall y \forall z \forall i_1 \forall i_2 (\mathit{Earns}(x, y \mid i_1) \wedge \mathit{Earns}(x, z \mid i_2) \wedge \texttt{before}(i_1, i_2) \implies y \leq z) \ .$$

Nevertheless, it seems that many temporal integrity constraints can be most conveniently expressed under an abstract, point-stamped representation. This is definitely true for static integrity constraints, like primary and

foreign keys. Formalisms based on interval-stamped relations may therefore provide operators like *timeslice* or *unfold* to switch to a point-stamped representation. Such “snapshotting” also underlies the *sequenced* semantics [8], which states that static constraints must hold independently at every point in time.
On the other hand, there are some constraints that concern only the concrete interval-stamped representation itself. For example, the interval-stamped relation *Earns* shown below satisfies constraint (3), but may be illegal if there is a constraint stating that temporal tuples need to be coalesced whenever possible.

Emp	*Sal*	*FromTo*
John Smith	10K	[1 May 2007, 30 Sep 2007]
John Smith	10K	[1 Oct 2007, 31 Dec 2007]
John Smith	11K	[1 Jan 2008, 31 Dec 2008]

2 Checking and Enforcing Temporal Integrity

Consider a database history to which a new state is added whenever the database is updated. Consistency is checked whenever a tentative update reaches the database. If the update would result in an inconsistent database history, it is rejected; otherwise the new database state is added to the database history. This scenario functions well if the entire database history is available for checking consistency. However, more efficient methods have been developed that allow to check temporal integrity without having to store the whole database history.
To check integrity after an update, there is generally no need to inspect the entire database history. In particular, static constraints can be checked by inspecting only the new database state; transition constraints can be checked by inspecting the previous and the new database state. Techniques for temporal integrity checking aim at reducing the amount of historical data that needs to be considered after a database update. In “history-less” constraint checking [9, 2, 11], all information that is needed for checking temporal integrity is stored, in a space-efficient way, in the current database state. The past states are then no longer needed for the purpose of integrity checking (they may be needed for answering queries, though).
The idea is similar to Temporal Vacuuming and can be formalized as follows. For a given database schema $\mathbf{S}$, let $\mathsf{FIN\_HISTORIES}(\mathbf{S})$ denote the set of finite database histories over $\mathbf{S}$ and $\mathsf{STATES}(\mathbf{S})$ the set of states over $\mathbf{S}$. Given a database schema $\mathbf{S}$ and a set $\mathbf{C}$ of temporal constraints, the aim is to compute a schema $\mathbf{T}$ and a computable function $E : \mathsf{FIN\_HISTORIES}(\mathbf{S}) \rightarrow \mathsf{STATES}(\mathbf{T})$, called *history encoding*, with the following properties:

for every $H \in \mathsf{FIN\_HISTORIES}(\mathbf{S})$, the consistency of H with respect to $\mathbf{C}$ must be decidable from $E(H)$ and $\mathbf{C}$. This can be achieved by computing a new set $\mathbf{C}'$ of (non-temporal) first-order constraints over $\mathbf{T}$ such that for every $H \in \mathsf{FIN\_HISTORIES}(\mathbf{S})$, $H \models_{\mathsf{temp}} \mathbf{C}$ if and only if $E(H) \models \mathbf{C}'$, where $\models_{\mathsf{temp}}$ is the desired notion of temporal satisfaction (see Section 1.2). Intuitively, the function E encodes, in a non-temporal database state over the schema $\mathbf{T}$, all information needed for temporal integrity checking.

•E must allow an incremental computation when new database states are added: for every $\langle H_0, \ldots, H_n\rangle \in \mathsf{FIN\_HISTORIES}(\mathbf{S})$, the result $E(\langle H_0, \ldots, H_n\rangle)$ must be computable from $E(\langle H_0, \ldots, H_{n-1}\rangle)$ and H_n. In turn, $E(\langle H_0, \ldots, H_{n-1}\rangle)$ must be computable from $E(\langle H_0, \ldots, H_{n-2}\rangle)$ and H_{n-1}. And so on.
Formally, there must be a computable function $\Delta : \mathsf{STATES}(\mathbf{T}) \times \mathsf{STATES}(\mathbf{S}) \rightarrow \mathsf{STATES}(\mathbf{T})$ and an initial database state $J_{\text{init}} \in \mathsf{STATES}(\mathbf{T})$ such that $E(\langle H_0\rangle) = \Delta(J_{\text{init}}, H_0)$ and for every $n > 0$, $E(\langle H_0, \ldots, H_n\rangle) = \Delta(E(\langle H_0, \ldots, H_{n-1}\rangle), H_n)$. The state J_{init} is needed to get the computation off the ground.

Note that the history encoding is fully determined by the quartet $(\mathbf{T}, J_{\text{init}}, \Delta, \mathbf{C}')$, which only depends on $\mathbf{S}$ and $\mathbf{C}$ (and not on any database history).
Such history encoding was developed by Chomicki [2] for constraints expressed in Past FOTL (including the **since** modal operator). Importantly, in that encoding, the size of $E(H)$ is polynomially bounded in the number of distinct constants occurring in H, irrespective of the length of H. Chomicki’s *bounded history encoding* can be illustrated by constraint (1), which states that an employee cannot work for the Pulse project if he was dropped from that project in the past. The trick is to maintain an auxiliary relation (call it *DroppedFromPulse*, part of the new schema $\mathbf{T}$) that stores names of employees who were dropped from the Pulse project in the past. Thus, $\mathit{DroppedFromPulse}(x)$ will be true in the current state for every employee name x that satisfies $\blacklozenge(\neg \mathit{WorksFor}(x, \text{Pulse}) \wedge \bullet\, \mathit{WorksFor}(x, \text{Pulse}))$ in the current state. Then, constraint (1) can be checked by checking $\neg\exists x(\mathit{WorksFor}(x, \text{Pulse}) \wedge \mathit{DroppedFromPulse}(x))$, which, syntactically, is a static constraint. Note incidentally that the label “static” is tricky here, because the constraint refers to past information stored in the current database state. Since history-less constraint checking must not rely on past database states, the

auxiliary relation *DroppedFromPulse* must be maintained incrementally (the function Δ): whenever an employee named x is dropped from the Pulse project (i.e. whenever the tuple $\mathit{WorksFor}(x, \text{Pulse})$ is deleted), the name x must be added to the *DroppedFromPulse* relation. In this way, one remembers who has been dropped from Pulse, but forgets when.

History-less constraint checking thus reduces dynamic to static constraint checking, at the expense of storing in auxiliary relations (over the schema $\mathbf{T}$) historical data needed for checking future integrity. Whenever a new database state is created as the result of an update, the auxiliary relations are updated as needed (using the function Δ). This technique is suited for implementation in active database systems [11, 4]: a tentative database update will trigger an abort if it violates consistency; otherwise the update is accepted and will trigger further updates that maintain the auxiliary relations.

The approach described above is characterized by ad hoc updates. A different approach is operational: the database can only be updated through a predefined set of update methods (also called transactions). These transactions are specified in a transaction language that provides syntax for embedding elementary updates (insertions and deletions of tuples) in program control structures. Restrictions may be imposed on the possible execution orders of these transactions. Bidoit and de Amo [1] define dynamic dependencies in a declarative way, and then investigate transaction schemes that can generate all and only the database histories that are consistent.

Although it is convenient to use an abstract temporal representation for specifying temporal constraints, consistency checks must obviously be performed on concrete representations. Techniques for checking static constraints, like primary and foreign keys, need to be revised if one moves from non-temporal to interval-timestamped relations [8]. Primary keys can be enforced on a non-temporal relation by means of a unique-index construct. On the other hand, two distinct tuples in an interval-timestamped relation can agree on the primary key without violating consistency. For example, the consistent temporal relation shown earlier contains two tuples with the same name John Smith.

KEY APPLICATIONS

Temporal data and integrity constraints naturally occur in many database applications. Transition constraints apply wherever the legality of new values after an update depends on the old values, which happens to be very common. History-less constraint checking seems particularly suited in applications where temporal conditions need to be checked, but where there is no need for issuing general queries against past database states. This may be the case in monitor and control applications [12].

CROSS REFERENCE

Since temporal integrity constraints are usually expressed in temporal logic, the entry Temporal Logic in Database Query Languages is relevant with respect to expressing temporal integrity constraints. The entry Point-stamped Temporal Models deals with the abstract database representation used to interpret temporal logics, and is to be contrasted with Interval-stamped Temporal Models. The formalization of (bounded) history encoding resembles Temporal Vacuuming. For reasons of feasibility or practicality, certain syntactically limited classes of temporal constraints have been studied in their own right; see Temporal Dependencies. The entry Temporal Constraints deals with equations and inequalities on the time domain that allow to define sets of time points.

RECOMMENDED READING

Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.

[1] Nicole Bidoit and Sandra de Amo. A first step towards implementing dynamic algebraic dependences. *Theor. Comput. Sci.*, 190(2):115–149, 1998.

[2] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[3] Jan Chomicki and Damian Niwinski. On the feasibility of checking temporal integrity constraints. *J. Comput. Syst. Sci.*, 51(3):523–535, 1995.

[4] Jan Chomicki and David Toman. Implementing temporal integrity constraints using an active DBMS. *IEEE Trans. Knowl. Data Eng.*, 7(4):566–582, 1995.

[5] Jan Chomicki and David Toman. Temporal logic in information systems. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems*, pages 31–70. Kluwer, 1998.

[6] Jan Chomicki and David Toman. Temporal databases. In Michael Fisher, Dov M. Gabbay, and Lluis Vila, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier Science, 2005.

[7] José Mauro Volkmer de Castilho, Marco A. Casanova, and Antonio L. Furtado. A temporal framework for database specifications. In *VLDB*, pages 280–291. Morgan Kaufmann, 1982.

[8] Wei Li, Richard T. Snodgrass, Shiyan Deng, Vineel Kumar Gattu, and Aravindan Kasthurirangan. Efficient sequenced integrity constraint checking. In *ICDE*, pages 131–140. IEEE Computer Society, 2001.

[9] Udo W. Lipeck and Gunter Saake. Monitoring dynamic integrity constraints based on temporal logic. *Inf. Syst.*, 12(3):255–269, 1987.

[10] Dimitris Plexousakis. Integrity constraint and rule maintenance in temporal deductive knowledge bases. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *VLDB*, pages 146–157. Morgan Kaufmann, 1993.

[11] A. Prasad Sistla and Ouri Wolfson. Temporal conditions and integrity constraints in active database systems. In Michael J. Carey and Donovan A. Schneider, editors, *SIGMOD Conference*, pages 269–280. ACM Press, 1995.

[12] A. Prasad Sistla and Ouri Wolfson. Temporal triggers in active databases. *IEEE Trans. Knowl. Data Eng.*, 7(3):471–486, 1995.