Temporal connectives versus explicit timestamps to query temporal databases*

Serge Abiteboul, Laurent Herr
INRIA-Rocquencourt†

Jan Van den Bussche
Limburgs Universitair Centrum‡

Abstract

Temporal databases can be queried either by query languages working directly on a timestamp representation, or by languages using an implicit access to time via temporal connectives. We study the differences in expressive power between these two approaches. First, we consider temporal and first-order logic. We show that future temporal logic is strictly less powerful than past-future temporal logic, and also that there are queries expressible in first-order logic with explicit timestamps that are not expressible in extended temporal logic. Our proof technique is novel and based on communication complexity. Then, we consider extensions of first-order logic with fixpoints or while-loops. Again the explicit temporal version of these languages, using timestamps, is compared with an implicit one, using instructions for moving in time. We also compare the temporal versions of the fixpoint language with those of the while language.

*Preliminary reports on this work were presented at the International Workshop on Temporal Databases (Zürich, September 1995) and the 15th ACM Symposium on Principles of Database Systems (Montreal, June 1996).

†Address: Domaine de Voluceau, B.P. 105, F-78153 Le Chesnay Cedex, France. E-mail: serge.abiteboul@inria.fr and laurent.herr@inria.fr.

‡Address: LUC, Department WNI, B-3590 Diepenbeek, Belgium. E-mail: vdbuss@luc.ac.be.

1 Introduction

A database history can be modeled as a finite sequence of instances discretely ordered by time. We are concerned here with querying such finite sequences of database instances, also called (discrete-time) *temporal databases*. As discussed by Chomicki [5], there are two different approaches to defining temporal query languages.

One approach is to view the sequence as one single relational database of an augmented schema where a "timestamp" column is added to each relation. The new column holds the time instants of validity of each tuple. This timestamp representation can then be queried using known relational query languages, where the linear order on timestamps is given as a built-in relation. The relational query languages we will be considering are the relational calculus (first-order logic, FO) and its iterative extensions fixpoint logic (FIXPOINT), extending FO with inflationary iteration, and while logic (WHILE), offering arbitrary iteration. When applied to timestamp representations of temporal databases these languages will be denoted respectively as TS-FO, TS-FIXPOINT and TS-WHILE.

Alternatively, one can use languages providing a more "implicit" access to time. A standard example is first-order *temporal logic* [7], an extension of classical logic with the temporal operators *since*, *until*, *next*, and *previous*. Since, as observed by Wolper [20], these operators can be viewed as searching for regular events, one can be more general and supply a temporal operator for each regular language. We denote standard temporal logic by TL, and extended temporal logic (with general regular events) by ETL. The sublanguage of TL offering only the future operators *next* and *until*, called *future* TL, is denoted by FTL. We will also be considering extensions of the languages FIXPOINT and WHILE with implicit temporal access via instructions for moving in time. These languages will be denoted respectively as T-FIXPOINT and T-WHILE.

In this paper, we compare these languages with respect to expressive power. Our results are depicted in Figure 1. Note that the only new languages are T-FIXPOINT and T-WHILE. Note also the central position of T-FIXPOINT. We believe this is an important language: it can be evaluated in polynomial time; it accesses time only implicitly; and it generalizes TS-FO and ETL. Of additional interest is that going from the inflationary language FIXPOINT to the temporal language T-FIXPOINT involves adding, besides the movements in time already mentioned, some non-inflationary

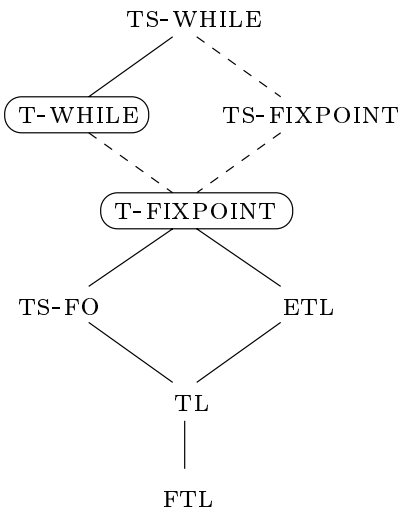

Figure 1: The relative power of temporal languages. Solid upward edges indicate strict containment. Dashed lines indicate that the strictness of the containment depends on unresolved questions in complexity theory.

language features as well.

Our results concerning FTL, TL and TS-FO should be contrasted to the extensively studied propositional case, where the three languages are equivalent [13, 10, 9]. Evidences that this equivalence fails in the predicate case already existed since 1971 [12]. Indeed, Kamp obtained results implying that TL is strictly weaker than TS-FO in the context of *densely* ordered temporal structures (rather than the discretely ordered ones we study in the present paper). Moreover, Toman and Niwinski [17] (still in the densely ordered case) showed that no finite set of first-order temporal operators can be added to TL so as to achieve expressive completeness.

The proof technique we use for separating ETL and TS-FO is novel and based on communication complexity [21, 14]. To our knowledge, this is the first time this tool is employed to analyze the expressive power of query languages.

This paper is organized as follows. In Section 2, we define temporal databases and their timestamp representations, and also briefly introduce temporal logic. In Section 3 we prove the results concerning FTL, TL and TS-FO. In Section 4, we briefly introduce the language WHILE, define T-WHILE, and

compare it with TS-WHILE and TS-FO. In Section 5, we briefly introduce the inflationary language FIXPOINT, study its augmentation with certain non-inflationary features, and define the central language T-FIXPOINT. In Section 6, we compare T-FIXPOINT to all other languages. In Section 7, we indicate special cases of temporal databases (including a notion of "local time") where the distinction between explicit versus implicit access to time largely disappears. Concluding remarks are presented in Section 8.

2 Temporal databases and temporal logic

2.1 Temporal databases and the language TS-FO

We assume some familiarity with relational databases (see, e.g., [1]). A *database schema* is a finite set of relation names, where each relation name has an associated arity. An *instance* of a schema assigns to each relation name a *finite* relation of appropriate arity over a fixed countably infinite domain of data elements. The *active domain* of an instance is the set of all data elements appearing in some of its relations.

A *temporal database* over a database schema $\mathcal{S}$ is a non-empty finite sequence $\mathbf{I} = I_1, \ldots, I_n$ ($n \geq 1$) of instances of $\mathcal{S}$. Every $j \in \{1, \ldots, n\}$ is called a *state* of $\mathbf{I}$. The *active domain* of a temporal database is the union of the active domains of its instances.

A *k-ary query* Q on temporal databases over schema $\mathcal{S}$ is a mapping assigning to each temporal database $\mathbf{I}$ over $\mathcal{S}$ a k-ary relation $Q(\mathbf{I})$ on the active domain of $\mathbf{I}$. (A 0-ary query is also called a Boolean query.)

We can identify a temporal database $\mathbf{I}$ with a two-sorted relational structure called the *timestamp representation* of $\mathbf{I}$. *Data elements* are taken from the active domain of $\mathbf{I}$, whereas *timestamps* are from the set of states $\{1, \ldots, n\}$.[1] The timestamp representation also contains the linear order on the states as an explicit binary relation $<$. Furthermore, it contains, for each relation R of arity k in the database schema, an extended relation $\bar{R}$ of arity $k+1$. The first k columns of this relation hold data elements; the last column

[1]For clarity, we assume without loss of generality that the domain of data elements is disjoint from the natural numbers. However, it is sometimes possible (and interesting) to simulate timestamps using data elements; we come back to this issue in Section 7.

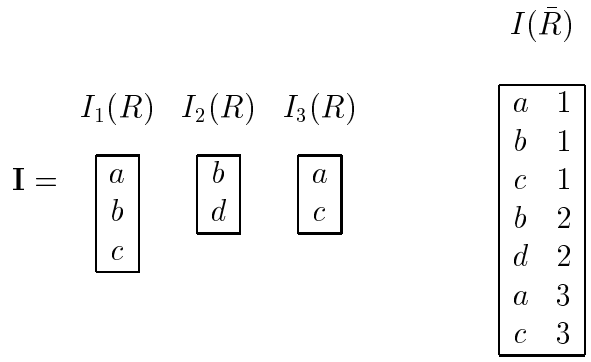

Figure 2: A temporal database and its timestamp representation.

holds timestamps. The contents of this relation, denoted $\mathbf{I}(\bar{R})$, is

$$\bigcup_{j=1}^{n} (I_j(R) \times \{j\}).$$

Example 2.1 A temporal database over a schema consisting of a single unary relation R, together with its timestamp representation are shown in Figure 2. ■

Using (two-sorted) first-order logic on the timestamp representation of a temporal database, we obtain a query language that is denoted by TS-FO. The *data-variables* in a formula range over data elements in the active domain and the *time-variables* range over states. The sorts of variables in a TS-FO formula will always be clear from the context. A formula $\varphi(x_1, \ldots, x_k)$ with k free data-variables and no free time-variables expresses a k-ary query

$$\varphi(\mathbf{I}) := \{(a_1, \ldots, a_k) \mid \mathbf{I} \models \varphi[a_1, \ldots, a_k]\}$$

in the standard way.

Example 2.2 If S is a unary relation holding employees of some company, the following TS-FO formula expresses the query returning those employees x who have been hired, later fired, and still later re-hired:

$$(\exists t_1)(\exists t_2)(\exists t_3)(t_1 < t_2 < t_3 \wedge \bar{S}(x, t_1) \wedge \neg \bar{S}(x, t_2) \wedge \bar{S}(x, t_3)).$$ ■

2.2 Temporal logic

An alternative way of providing a temporal query language is to extend first-order logic with temporal operators rather than explicit time-variables. We will use temporal operators based on regular events, leading to *extended temporal logic*, denoted by ETL [20]. The syntax of ETL over some database schema $\mathcal{S}$ is obtained by using the formation rules for standard first-order logic over $\mathcal{S}$ together with one additional formation rule:

> Let L be a regular language over the finite alphabet $(v_1, \ldots, v_p)$, and let $\varphi_1, \ldots, \varphi_p$ be formulas. Then
>
> $$L^+(\varphi_1, \ldots, \varphi_p) \qquad \text{and} \qquad L^-(\varphi_1, \ldots, \varphi_p)$$
>
> are also formulas.

The order of the letters in the alphabet $(v_1, \ldots, v_p)$ is relevant since it allows to relate these letters to the arguments $(\varphi_1, \ldots, \varphi_p)$.

The semantics of ETL is as follows. Let $\mathbf{I} = I_1, \ldots, I_n$ be a temporal database over $\mathcal{S}$. Let $\varphi(\bar{x})$ be an ETL formula with free variables $\bar{x} = x_1, \ldots, x_k$, let $\bar{a} = a_1, \ldots, a_k$ be data elements in the active domain of $\mathbf{I}$, and let $j \in \{1, \ldots, n\}$ be a state. The *truth* of $\varphi[\bar{a}]$ in $\mathbf{I}$ at time j, denoted by $\mathbf{I}, j \models \varphi[\bar{a}]$, is defined as follows:

1. If φ is an atomic formula, a conjunction, a negation, or a quantification, the definition is as usual. Quantification is always on the active domain.

2. If φ is of the form $L^+(\varphi_1, \ldots, \varphi_p)$, with L a regular language over the alphabet $(v_1, \ldots, v_p)$, then $\mathbf{I}, j \models \varphi[\bar{a}]$ if there exists a word $w = v_{w_j} \ldots v_{w_n}$ of length $(n - j + 1)$ in L such that

   $$\mathbf{I}, j \models \varphi_{w_j}[\bar{a}] \quad \text{and} \quad \ldots \quad \text{and} \quad \mathbf{I}, n \models \varphi_{w_n}[\bar{a}].$$

3. Symmetrically, if φ is $L^-(\varphi_1, \ldots, \varphi_p)$, then $\mathbf{I}, j \models \varphi[\bar{a}]$ if there exists a word $w = v_{w_j} \ldots v_{w_1}$ of length j in L such that

   $$\mathbf{I}, j \models \varphi_{w_j}[\bar{a}] \quad \text{and} \quad \ldots \quad \text{and} \quad \mathbf{I}, 1 \models \varphi_{w_1}[\bar{a}].$$

Example 2.3 The formula $L_1^+(\mathbf{true}, \varphi)$, where L_1 is the language a^*ba^* over the alphabet (a, b), is true at time j iff there is some time in the future of j

(including j itself) where φ is true. Similarly, $L_1^-(\mathbf{true}, \varphi)$ expresses that φ holds sometime in the past.

Now recall Example 2.2. The following ETL formula is true of x at some time iff x is not an employee now, has been one in the past, and will again be one in the future:

$$\neg S(x) \wedge L_1^-(S(x)) \wedge L_1^+(S(x)).$$

For another example, a formula which is true only in the last (or first) state is $L_2^+(\mathbf{true})$ (or $L_2^-(\mathbf{true})$), where L_2 is the singleton language $\{a\}$.

Finally, the formula $L_3^+(\mathbf{true})$, where L_3 is the language $(aa)^*$, is true in the first state iff the length of the temporal database is even. ■

The previous example showed how the familiar temporal operators "sometimes in the future" and "sometimes in the past" of standard temporal logic [7] can be expressed in ETL. We next show how the other temporal operators of standard temporal logic can be expressed.

The temporal connectives since and until can be expressed in ETL as follows:

$$\varphi \,\mathsf{since}\, \psi \equiv L_4^-(\varphi, \varphi \wedge \psi, \mathbf{true})$$

and

$$\varphi \,\mathsf{until}\, \psi \equiv L_4^+(\varphi, \varphi \wedge \psi, \mathbf{true}),$$

where L_4 is the language a^*bc^* over the alphabet (a, b, c). The connectives next and previous are expressed as

$$\mathsf{next}\, \varphi \equiv L_5^+(\mathbf{true}, \varphi)$$

and

$$\mathsf{previous}\, \varphi \equiv L_5^-(\mathbf{true}, \varphi),$$

where L_5 is the language aba^* over the alphabet (a, b).

Standard temporal logic, i.e., the fragment of ETL having as only temporal operators since, until, next, and previous, is denoted by TL. Future temporal logic, i.e., the fragment of TL having only the future operators next and until, is denoted by FTL.

The above examples also illustrate a subtle feature of our definition. When searching for a regular event in the future (using the L^+ connective), we require that a word in L can be found which reaches precisely the last

state of the temporal database. Similarly, when searching in the past, a word must be found which reaches precisely the first state. We refer to this as *full search*, as opposed to *partial search* which does not require the match to reach the beginning or end. As illustrated in some of the above examples, it is easy to simulate partial search using full search: it suffices to continue testing for **true** after the desired match has been found.[2]

We still have to define formally how ETL formula express queries. Let $\varphi(x_1, \ldots, x_k)$ be an ETL formula with k free variables. Then φ expresses the query

$$Q(\mathbf{I}) := \{(a_1, \ldots, a_k) \mid \mathbf{I}, 1 \models \varphi[a_1, \ldots, a_k]\}.$$

So the evaluation of an ETL query is started in the first state.

3 Comparing TS-FO with temporal logic

By the *expressive power* of a query language one means the class of queries expressible in that language. In this section, we compare the languages TS-FO, FTL, TL and ETL with respect to expressive power. Their relationship is depicted in Figure 1.

The containments FTL $\subseteq$ TL $\subseteq$ ETL are trivial. Also the containment TL $\subseteq$ TS-FO is clear; for example, to express that φ until ψ holds at t, one states that there exists $t' > t$ such that ψ holds at t' and φ holds at each t'' between t and t'. As shown in Example 2.3, the query "the length of the temporal database is even" is expressible in ETL. It is not expressible in TS-FO, since parity of a linear order is well-known not to be first-order definable.

Hence, to complete the picture provided by Figure 1, we have to prove that (i) there are queries expressible in TL but not in FTL, and (ii) there are queries expressible in TS-FO but not in ETL. These two proofs are given in the next two subsections.

3.1 TL versus FTL

Theorem 3.1 *The Boolean query* Q*:* $(\exists t > 1)(\forall x)(\bar{S}(x,t) \leftrightarrow \bar{S}(x,1))$*, is expressible in* TL *but not in* FTL*.*

[2]We leave it as an exercise for the reader to show that conversely, full search can be simulated using partial search.

Proof. We can express Q in TL as

$$\mathsf{next}\,\Diamond^+(\forall x)(S(x) \leftrightarrow \Diamond^-(\mathbf{first} \wedge S(x))),$$

where we have used the abbreviations

$$\begin{aligned} \Diamond^+\varphi &= \mathbf{true}\ \mathsf{until}\ \varphi; \\ \Diamond^-\varphi &= \mathbf{true}\ \mathsf{since}\ \varphi; \\ \mathbf{first} &= \neg\,\mathsf{previous}\ \mathbf{true}. \end{aligned}$$

(Note that **first** is only true in the first state.)

To show that Q is not expressible in FTL, we first observe that FTL-formulas can be written in a normal form, where the only way the operator until can occur is in a combination with next of the form $\mathsf{next}(\varphi\ \mathsf{until}\ \psi)$. Indeed, $\varphi\ \mathsf{until}\ \psi$ is equivalent to $(\varphi \wedge \psi) \vee (\varphi \wedge \mathsf{next}(\varphi\ \mathsf{until}\ \psi))$.

Now let θ be an FTL sentence in normal form. Let D be some arbitrary fixed finite domain of data elements, let d be the cardinality of D, and let $n \geq 1$ be some arbitrary fixed natural number. We consider temporal databases $I_1, \ldots, I_n$ on D, and define the function F on the "tails" of such databases by

$$F(I_2, \ldots, I_n) := \{I_1 \mid (I_1, I_2, \ldots, I_n), 1 \models \theta\}.$$

If θ expressed the query Q, then the cardinality of the image of F would be

$$\sum_{k=1}^{n} \binom{2^d}{k}.$$

Indeed, two sequences $I_2, \ldots, I_n$ and $I'_2, \ldots, I'_n$ have the same image by F if and only if the sets $\{I_2, \ldots, I_n\}$ and $\{I'_2, \ldots, I'_n\}$ are the same. But there are $\binom{2^d}{k}$ ways to choose a set of k distinct subsets of D.

As a particular case, if n is 2^d, the cardinality of the image of F is 2^{2^d}. However, in Lemma 3.2 we will show that the cardinality of the image of F is at most 2^{d^α}, for some integer α depending only on θ, and for sufficiently large d. We thus arrive at a contradiction. ■

Lemma 3.2 *The cardinality of the image of F is at most 2^{d^α}, for some integer α depending only on θ, and for sufficiently large d.*

Proof. Call a *temporal subformula* of θ, any subformula of the form $\mathsf{next}\,\varphi$ or $\mathsf{next}(\varphi\ \mathsf{until}\ \psi)$. A temporal subformula of θ is called maximal if it is

not a subformula of another temporal subformula of θ. Let $\theta_1, \ldots, \theta_k$ be the maximal temporal subformulas of θ. For each θ_i, the satisfaction of θ_i on a temporal database $(I_1, I_2, \ldots, I_n)$ at the first state only depends on the tail $I_2, \ldots, I_n$ of that database. So, the following function F_i on tails is well-defined and does not depend on a particular choice for I_1:

$$F_i(I_2, \ldots, I_n) := \{\bar{a} \mid (I_1, I_2, \ldots, I_n), 1 \models \theta_i[\bar{a}]\}.$$

If α_i denotes the number of free variables of θ_i, the image of F_i is a set of relations of arity α_i, and thus its cardinality is at most $2^{d^{\alpha_i}}$. But, Lemma 3.3 will imply that the cardinality of the image of F is less than the product of the cardinalities of the F_i's. Hence, if we take $\alpha > \max(\alpha_1, \ldots, \alpha_k)$, the cardinality of the image of F is less than 2^{d^α} for sufficiently large d, since 2^{d^α} dominates $2^{d^{\alpha_1}+\cdots+d^{\alpha_k}}$ for sufficiently large d. ■

Lemma 3.3 *There is an injection* $\omega : \operatorname{Im} F \to \prod_{i=1}^{k} \operatorname{Im} F_i$.[3]

Proof. Let ω be a function such that for each x in $\operatorname{Im} F$, there is a tail $(I_2, \ldots, I_n)$ with $F(I_2, \ldots, I_n) = x$ (we call such a tail an *antecedent* of x) such that:

$$\omega(F(I_2, \ldots, I_n)) = (F_1(I_2, \ldots, I_n), \ldots, F_k(I_2, \ldots, I_n)).$$

Note that the choice of the antecedent of x by F is arbitrary.

Such a function ω is injective. Indeed, if $F(I_2, \ldots, I_n)$ and $F(I'_2, \ldots, I'_n)$ have the same image by ω, the definition of ω ensures that $F_i(I_2, \ldots, I_n)$ and $F_i(I'_2, \ldots, I'_n)$ are equal for all i. But θ is a first order combination of the θ_i and of first-order formulas evaluated on I_1, so that for a given I_1, θ has the same value on $I_1, I_2, \ldots, I_n$ and $I_1, I'_2, \ldots, I'_n$. So, $F(I_2, \ldots, I_n) = F(I'_2, \ldots, I'_n)$ which yields the result. ■

3.2 ETL versus TS-FO

In this subsection, we first introduce a variant of the communication protocols of Yao [21] (see also [14]), and introduce the notion of "constant communication complexity" of binary predicates on sets of sets (of data elements). We

[3] By $\operatorname{Im} f$ we mean the image of a function f.

also introduce the class of *split* temporal databases. Each binary predicate on sets of sets gives rise to a query on split databases. We then prove that if the communication complexity of a predicate is not constant, then the corresponding query is not expressible in ETL. However, natural predicates of non-constant communication complexity exist whose corresponding queries *are* expressible in TS-FO.

3.2.1 Communication protocols

Let P be a binary predicate on sets of sets of data elements. We say that P has *constant communication complexity* if there exist fixed natural numbers k and r and a communication protocol between two parties (denoted by A and B) that, for each finite set D of data elements, can evaluate $P(X, Y)$ on any sets X and Y of non-empty subsets of D as follows:

1. A gets X and B gets Y. Both parties also know D.
2. A sends a message $a_1 = a_1(D, X)$ to B, and B replies with a message $b_1 = b_1(D, Y, a_1)$ to A. Each message is a k-ary relation on D.
3. A again sends a message $a_2 = a_2(D, X, b_1)$ to B, and B again replies with a message $b_2 = b_2(D, Y, a_1, a_2)$.
4. After r such message exchanges, both A and B have enough information to evaluate $P(X, Y)$ correctly. Formally, they apply a Boolean function

   $$a_{r+1}(D, X, b_1, \ldots, b_r) \quad \text{(for A)}$$

   or

   $$b_{r+1}(D, Y, a_1, \ldots, a_r) \quad \text{(for B)}$$

   that evaluates to true iff $P(X, Y)$ is true.

So, formally, a protocol consists of the functions a_1, ..., a_r, a_{r+1} and b_1, ..., b_r, b_{r+1}. Note that the computing power of A and B is unlimited; the functions defining the protocol can be completely arbitrary.

Example 3.4 As a simple example, let $P(X, Y)$ be true if the maximal cardinality of an element in X is larger than the maximal cardinality of an element in Y. Then P has constant communication complexity with $k = 1$ and $r = 1$. Indeed, A sends to B an element of X with maximal cardinality, and B replies with an analogous element for Y. Both A and B can then evaluate $P(X, Y)$ on their own, by a simple comparison of cardinalities. ■

We have a first lemma:

Lemma 3.5 *The equality, inclusion and disjointness predicates do not have constant communication complexity.*

Proof. Suppose there is a communication protocol for the equality predicate with r exchanges of messages of arity k. Call any such sequence $a_1 b_1 \ldots a_r b_r$ of messages a *dialogue*. Since k is fixed, for large enough D there are less dialogues than sets of non-empty subsets of D. Hence, there are two different such sets X and Y such that the protocol yields the same dialogue when evaluating $P(X, X)$ and $P(Y, Y)$. But then this same dialogue will also be used for evaluating $P(X, Y)$; a contradiction.

It follows that the inclusion and disjointness predicates are not of constant communication complexity either. Indeed, communication protocols for these predicates can be easily transformed into a communication protocol for equality. It suffices to observe that $X = Y$ iff X is included in Y and vice versa, and that $X \subseteq Y$ iff X and the complement of Y are disjoint. ■

Our notion of communication protocols is a "set-based" variant of the original bit-based one, where the predicate to be evaluated is a predicate on bit-strings, and the exchanged messages are individual bits. Yao [21] showed in this setting that the equality predicate on strings of length n requires a number of bit exchanges that is linear in n. Lemma 3.5 can also be proven from this fact.

3.2.2 Split databases

We now fix the database schema to consist of one single unary relation S. A temporal database is then a sequence of finite sets of data elements. A temporal database is called *split* if there is exactly one state whose instance is empty. This state is called the *middle* state of the split database. If $\mathbf{I} = I_1, \ldots, I_n$ is a split database with middle state m then its *right part* $I_m, \ldots, I_n$ is denoted by $\mathbf{I}_{right}$ and its *left part* $I_1, \ldots, I_m$ by $\mathbf{I}_{left}$. Observe that one can test in TL whether a temporal database is split.

We next define an auxiliary language *split*-ETL whose semantics is only defined on split databases. Syntactically, split-ETL differs from ETL only in that each temporal operator L^+ (L^-) is split into a "left" and a "right" version L^+_{left} and L^+_{right} (L^-_{left} and L^-_{right}).

Informally, the left (right) version of a temporal operator behaves roughly the same as the operator itself, except that only the left (right) part of the split database is taken into consideration. Formally, let $\mathbf{I}$ be a split database of length n with middle state m. For each state j of $\mathbf{I}$, we define

$$\mathit{left}(j) := \begin{cases} j & \text{if } j \leq m \\ m & \text{if } j \geq m \end{cases}$$

and

$$\mathit{right}(j) := \begin{cases} 1 & \text{if } j \leq m \\ j - m + 1 & \text{if } j \geq m \end{cases}$$

So, $\mathit{left}(j)$ ($\mathit{right}(j)$) is the state in the left (right) part of $\mathbf{I}$ corresponding to j, if j is indeed contained in that part; if not, the default values m and 1, respectively, are used.

The semantics of the split temporal operators is then defined as follows. For $\star$ being either $-$ or $+$, $\mathbf{I}, j \models L^{\star}_{\mathit{left}}$ if $\mathbf{I}_{\mathit{left}}, \mathit{left}(j) \models L^{\star}$, and $\mathbf{I}, j \models L^{\star}_{\mathit{right}}$ if $\mathbf{I}_{\mathit{right}}, \mathit{right}(j) \models L^{\star}$.

We now have our second lemma.

Lemma 3.6 *On split databases, each* ETL *formula is equivalent to a split-*ETL *formula.*

Proof. Consider a temporal operator L^{+} of ETL, with L a regular language over the alphabet $(v_1, \ldots, v_p)$. Then L is defined by some finite automaton M. Let the states of M be numbered $1, \ldots, q$, with 1 the initial state, and let F be the set of final states. For $z \in \{1, \ldots, q\}$ and $Z \subseteq \{1, \ldots, q\}$, let M_{zZ} be the automaton obtained from M by changing the initial state to z and the set of final states to Z, and denote by L_{zZ} the language defined by M_{zZ}. Let v_0 be a symbol not in the alphabet $\{v_1, \ldots, v_p\}$. Then the ETL formula $L^{+}(\varphi_1, \ldots, \varphi_p)$ can be expressed in split-ETL as

$$\begin{aligned} \big((\mathit{at\_right} \vee \mathit{at\_middle}) \wedge L^{+}_{\mathit{right}}(\varphi_1, \ldots, \varphi_p)\big) \vee \\ \Big(\mathit{at\_left} \wedge \bigvee_{z=1}^{q} \big((L_{1\{z\}})^{+}_{\mathit{left}}(\varphi_1, \ldots, \varphi_p) \wedge \\ (v_0 L_{zF})^{+}_{\mathit{right}}(\mathbf{true}, \varphi_1, \ldots, \varphi_p)\big)\Big). \end{aligned}$$

In the above, the language $v_0 L_{zF}$ is interpreted over the alphabet $(v_0, v_1, \ldots, v_p)$, and we have used the abbreviations

$$\begin{aligned} \mathit{at\_middle} &= \neg(\exists x) S(x); \\ \mathit{at\_left} &= K^{+}_{\mathit{left}}(\mathbf{true}, \mathit{at\_middle}); \\ \mathit{at\_right} &= K^{-}_{\mathit{right}}(\mathbf{true}, \mathit{at\_middle}), \end{aligned}$$

where K is the language a^+b over the alphabet (a, b).

The case L^- is treated similarly. ■

3.2.3 Inexpressibility

Let P be a binary predicate on sets of sets, as in Subsection 3.2.1. Consider the Boolean query Q_P on split databases defined as follows. For a split database $\mathbf{I} = I_1, \ldots, I_n$ with middle state m, $Q_P(\mathbf{I}) =$ true if $P(L, R)$ holds, where $L = \{I_j \mid 1 \leq j < m\}$ and $R = \{I_j \mid m < j \leq n\}$.

Our third lemma connects temporal queries to communication protocols:

Lemma 3.7 *If Q_P is expressible in* ETL*, then P has constant communication complexity.*

Proof. Assume Q_P is expressible in ETL. By Lemma 3.6, Q_P is expressible by a split-ETL formula θ. Consider all subformulas of θ of the form $L^\star_\delta(\ldots)$, where $\star$ is $+$ or $-$ and δ is *left* or *right*, and let $\pi_1, \ldots, \pi_r$ be a listing of these such that each subformula occurs after its own subformulas. Let k be the maximal number of free variables of any of these subformulas. We show that θ yields a communication protocol for P with r exchanges of messages of arity k.

Let X and Y be two sets of non-empty subsets of a finite set D of data elements, and consider any split temporal database $\mathbf{I}$ with middle state m, such that $X = \{I_j \mid 1 \leq j < m\}$ and $Y = \{I_j \mid m < j \leq n\}$. In order to evaluate $P(X, Y)$, it suffices to evaluate $Q_P(\mathbf{I})$, for which in turn it suffices to evaluate θ at some state of $\mathbf{I}$. To do the latter, the parties evaluate, in succession, each subformula π_i on every k-tuple of active domain elements, at the middle state. If the temporal operator of π_i is a left (right) version, then A (B) knows how to do this and he sends the resulting k-ary relation to B (A).

Note in this respect that both parties can be assumed, without loss of generality, to know the active domain of $\mathbf{I}$; if not, they can send the set of elements of D appearing in their set of sets to each other in a single exchange of messages. When the values of all the π_i are known to both parties, they have enough information to evaluate θ. ■

Putting everything together, we obtain our main result:

Theorem 3.8 *Over schemas containing at least one relation of non-zero arity, there are queries expressible in* TS-FO *but not in* ETL. *In particular, query Q "are there two different states with the same instance?" is expressible in* TS-FO *but not in* ETL.

Proof. Without loss of generality we assume the schema consists of a single unary relation S. Query Q is obviously expressible in TS-FO:

$$(\exists t)(\exists t')\big(t \neq t' \wedge (\forall x)(\bar{S}(x,t) \leftrightarrow \bar{S}(x,t'))\big).$$

On the class of split databases whose left and right parts do not contain repetitions, Q corresponds to Q_P, where P is the non-disjointness predicate. By Lemma 3.5, the complement of P (so also P itself) does not have constant communication complexity. Hence, by Lemma 3.7, Q is not expressible in ETL. ■

An important remark that can be made concerning our result is that it remains valid under the assumption that a total order on the data elements is available. Indeed, the proof of Lemma 3.5 holds regardless of *any* additional knowledge (e.g., a total order) the parties may have of the set D.

3.2.4 Infinite temporal databases

We conclude this section by extending our result to the case of infinite (but still discrete-time) temporal databases.

An *infinite temporal database* over a schema $\mathcal{S}$ is an infinite sequence $\mathbf{I} = I_1, I_2, \ldots$ of instances of $\mathcal{S}$. So, the set of states is the set of non-negative natural numbers, and the active domain may be infinite (although every individual instance is, by definition, still finite). In the present discussion, we focus on expressiveness, and not on the issue of finitely representing an infinite temporal database, or effectively computing answers to queries. References on these issues can be found in [5].

The query languages TS-FO and ETL can also be used on infinite temporal databases. For TS-FO, this is clear. For ETL, one uses ω-languages rather than ordinary languages in defining the semantics of the future temporal operators, since the future of every state is now infinite. The past of every state is, on the contrary, still finite. (Though the present discussion extends easily to the case of two-way infinite temporal databases.) An ω-language [15] is a set of infinite, rather than finite, words, and that a regular ω-language

can still be defined by a finite automaton; an infinite word is accepted by the automaton if while reading the word it enters an accepting state infinitely often.

We now argue that our techniques of the previous section extend to the infinite case. An infinite temporal database is again called *split* if there is exactly one state whose instance is empty. The *right part* of an infinite split database is itself infinite; the *left part* is finite. Syntax and semantics of *split*-ETL on infinite split databases are defined in terms of ETL exactly as before. The result that split-ETL can simulate ETL on split databases goes through in the infinite case; the only modification to the proof of Lemma 3.6 is that in the large expression for L^+, L_{zF} now becomes an ω-language. Finally, the proof of Lemma 3.7 carries over verbatim, with the condition that instead of a finite $\mathbf{I} = I_1, \ldots, I_n$ we use an infinite $\mathbf{I} = I_1, I_2, \ldots$, and instead of $\{I_j \mid m < j \leq n\}$ we use $\{I_j \mid m < j\}$. Note that this implies that party B of the protocol deals with an infinite object, but this is of no concern since his computing power is unlimited.

The result of this section can thus be summarized as follows:

Theorem 3.9 *Both on finite and on infinite temporal databases over a schema containing at least one relation of non-zero arity, there are queries expressible in* TS-FO *but not in* ETL. *As a consequence,* TL *is strictly weaker than* TS-FO.

4 Iterative queries

Let us first briefly recall how relational calculus is extended with iteration to obtain the language WHILE. (See [1] for a more detailed presentation of the languages WHILE and FIXPOINT considered in the following sections.)

An *assignment statement* is an expression of the form $X := E$, where X is an *auxiliary relation* and E is a relational calculus query which can involve both relations from the database scheme and auxiliary relations. Each auxiliary relation has a fixed arity; in the above assignment statement, the arity of the result of E must match the arity of X.

We can now build *programs* from assignment statements using sequencing $P_1; P_2$ and *while-loops*: if P is a program, then so is **while** φ **do** P **od**, where φ is a relational calculus sentence. The query language thus obtained is called WHILE. The execution of a program on a database instance is defined in the

obvious manner. The result of the query expressed by a program is the value of some designated answer relation at completion of the execution.[4]

The language WHILE on the timestamp representations of temporal databases provides a very powerful temporal query language which is denoted by TS-WHILE.

Example 4.1 The query "give the elements that belong to all odd-numbered states" is not expressible in the relational calculus with timestamps, but it is expressible in TS-WHILE as follows:

$Current := \{1\}$;
$A := \{x \mid S(x, 1)\}$;
while $(\exists t)(\exists t')(Current(t) \wedge t' = t + 2)$ **do**
  $Current := \{t' \mid (\exists t)(Current(t) \wedge t' = t + 2)\}$;
  $A := A \cap \{x \mid (\exists t)(Current(t) \wedge S(x, t))\}$
od.

In the above program, *Current* and A are auxiliary relations, and A is the answer relation. The use of the constant '1' and the addition '$t' = t + 2$' are only abbreviations which can be directly expressed in terms of the order on the timestamps. ■

An alternative temporal query language based on WHILE, not involving timestamps, can be obtained by extending WHILE with more implicit temporal features. One way to do this is to execute programs on a machine which can move back and forth over time. Formally, we provide, in addition to assignment statements, the two statements **left** and **right** which move the machine one step in the required direction.[5] Furthermore, we partition the auxiliary relations into *state relations*, which are stored in the different states, and *shared relations*, which are stored in the memory of the machine itself. So, the values of (and assignments to) state relations depend on the current state the machine is looking at, while this is not the case for shared relations. Finally, we assume two built-in nullary state relations *First* and *Last*, with *First* being true only in the first state, and *Last* being true only in the last state. The machine always starts execution from the first state.

The temporal query language WHILE extended with left and right moves just described is denoted by T-WHILE.

[4] If the execution loops indefinitely, the result is undefined. Infinite loops can always be detected at run time in WHILE [2].

[5] In the first state, **left** has no effect; in the last state, **right** has no effect.

Example 4.2 The query from Example 4.1 can be expressed in T-WHILE as follows:

shared $A(1)$, $Even(0)$;
$A := \{x \mid S(x)\}$; $Even := \{()\}$;
while $\neg Last$ **do**
  right;
  $Even := \{()\} - Even$;
  if $Even \neq \emptyset$ **then** $A := A \cap \{x \mid S(x)\}$
od.

In the above program, A and $Even$ are both shared relations. Note how they are "declared" as variables in the beginning of the program, indicating their status of shared relation and their arity; we will always use such declarations when presenting T-WHILE programs in the sequel. The if-then construct is only an abbreviation and can be expressed in the relational calculus. ■

We next study the expressive power of T-WHILE. We will see in the next section that it strictly encompasses TS-FO, and hence TL as well. We now show:

Proposition 4.3 T-WHILE *is strictly contained in* TS-WHILE.

Proof. The simulation of T-WHILE by TS-WHILE is done using a *Current* relation as in Example 4.1 which holds the current temporal position of the machine. The state relations are simulated by their time-stamped version, whereas no special transformation is needed for shared relations. The retrieval of a state relation is simulated by a join between its time-stamped version and *Current*. *First* is simulated by the formula

$$\neg(\exists t)(\exists t')(Current(t) \wedge t' < t)$$

Last is simulated symmetrically. A left move is simulated by updating the *Current* relation (a right move is simulated symmetrically):

$Current :=$ **if** $\neg First$ **then** $Current$
else $\{t' \mid (\exists t)(Current(t) \wedge t' = t - 1\}$,

where $t' = t - 1$ is an abbreviation for $(t' < t) \wedge \neg(\exists t'')(t' < t'' < t)$.

The argument for strictness is based on complexity. If we restrict our attention to propositional databases (having only relations of arity 0), the complexity of TS-WHILE programs in terms of the length n of the temporal database only is precisely PSPACE. Indeed, on propositional databases, TS-WHILE reduces to the language WHILE on an ordered relational (non-temporal) database consisting of a number of unary relations on timestamps. WHILE is well-known to coincide with PSPACE on ordered databases [1]. However, the space complexity of T-WHILE programs in terms of n is linear: we only have to store the state relations at each state. The proposition then follows from the space hierarchy theorem [11]. ■

5 Fixpoint queries

General WHILE programs can only be guaranteed to run in polynomial space (PSPACE) and hence their computational complexity is probably intractable in general. However, there is a well-known restriction of WHILE which runs in polynomial time (PTIME). This restriction consists of allowing only *inflationary* assignment statements, of the form $X := X \cup E$ (abbreviated $X \mathrel{+}= E$). Before execution of an inflationary WHILE program all auxiliary relations are initialized to the empty set. In such an execution, a while-loop whose stopping condition is never fulfilled, and thus seemingly loops forever, will repeat a configuration after an at most polynomial number of steps.[6] The computation has then "reached a fixpoint" and the result of the query can be determined as well as if the program execution would have ended normally. The query language thus obtained is therefore called FIXPOINT.[7]

On *ordered* databases (where a linear order on the active domain is available in a database relation), a query is in PTIME if and only if it is expressible in FIXPOINT. It is an open question whether FIXPOINT is strictly weaker than WHILE, but it is known [3] that this question is equivalent to the open problem in computational complexity on the strict containment of PTIME in

[6] A configuration of a program execution consists of the values of the auxiliary relations plus the position in the program.

[7] Usually [1] the language FIXPOINT is defined using "repeat-while-change" loops instead of while-loops with a stopping condition. We have chosen our definition because it yields a more flexible language when extended to a temporal context (cf. the language T-FIXPOINT to be defined later).

PSPACE.

Similarly to TS-WHILE, the language FIXPOINT on timestamp representations of temporal databases provides a powerful yet computationally tractable temporal query language denoted by TS-FIXPOINT.

Example 5.1 The query of Example 4.1 can also be expressed in TS-FIXPOINT as follows:

$Current \mathrel{+}= \{1\}$;
$B \mathrel{+}= \{x \mid \neg S(x, 1)\}$;
while $(\exists t)(\exists t')(Current(t) \wedge \neg Current(t') \wedge t' = t + 2)$ **do**
  $Current \mathrel{+}= \{t' \mid (\exists t)(Current(t) \wedge t' = t + 2)\}$;
  $B \mathrel{+}= \{x \mid (\exists t)(Current(t) \wedge \neg S(x, t))\}$
od;
$A \mathrel{+}= \{x \mid \neg B(x)\}$.

Remember that data variables (such as x in the formula $\neg S(x, 1)$) range over the data elements in the active domain only.

Note that this query could be expressed simpler by storing all odd states in a relation, and then computing the intersection of these states. ■

As an alternative to TS-FIXPOINT, we could depart from the language T-WHILE and restrict it to inflationary assignments only, to obtain a PTIME temporal query language. However, this language would be rather inflexible, since a pure inflationary restriction is an obstacle to the inherently non-inflationary back-and-forth movements along time involved in temporal querying. (For simple temporal queries involving only one single scan, this would suffice.)

This obstacle can also be analyzed using a complexity argument. As we have seen in Proposition 4.3 for T-WHILE, the available space is linear in the length n of the sequence. In FIXPOINT, the restriction to PTIME is achieved by a careful inflationary use of space. Thus, the restriction of T-WHILE to inflationary assignments would lead to a computation that would run in time linear in n.

We propose to alleviate the problem by adding two extra features to standard FIXPOINT that allow to use non-inflationary assignments in a controlled manner: "local variables" and "non-inflationary variables".

(a) Local variables to blocks: Certain auxiliary relations can be declared as local variables to program blocks. These relations can only be assigned

to within the block, and each time the block is exited, they are emptied. (If the local variables are state relations, they are emptied in each state.) Syntactically, if P is a program then [**local** $V_1, \ldots, V_r$; P] is a program block with local auxiliary relations $V_1, \ldots, V_r$.

(b) Non-inflationary variables: Certain auxiliary relations can be declared to be non-inflationary. They can be assigned to without any inflationary restriction. However, they are not taken into account in determining whether the program has reached a fixpoint. (Hence, this remains in PTIME.) Syntactically, these variables will be declared using the keyword **noninf**.

The inflationary restriction of T-WHILE, to which the above two extra non-inflationary features are added, yields a temporal query language that we call T-FIXPOINT. Configurations of T-FIXPOINT programs now include the current temporal state of the machine, which is taken into account to see whether the computation has reached a fixpoint (i.e., repeated a configuration).

It is important to note that the extra features of local and non-inflationary variables only make a difference in the context of T-FIXPOINT: in the standard FIXPOINT language, they can be simulated as shown in the next proposition. This result is interesting in its own right, since it facilitates expressing PTIME computations in FIXPOINT. It also indicates a fundamental distinction between temporal querying and non-temporal querying.

Proposition 5.2 *Adding program blocks with local variables and noninflationary variables with the restrictions described above to* FIXPOINT *does not increase the expressive power of the language.*

Proof. We only present a sketch of the argument. The key observation is that, due to the inflationary nature of the computation, a program block can be executed only so many times as tuples are inserted in the auxiliary relations that are global (i.e., not local) to this block. Hence, the contents of the local variables can be simulated by versioning their tuples with the tuples inserted in the global variables since the previous invocation of the program block (using Cartesian product). Emptying the local variables then simply amounts to creating a new version. The old versions are accumulated in a separate relation. In this manner the process is entirely inflationary, as desired.

We can also simulate the noninflationary variables using a similar versioning technique. The version consists of the tuples inserted in the ordinary, inflationary variables since the previous non-inflationary assignment. Since the program terminates as soon as the inflationary variables reach a fixpoint, we will not run out of versions. ■

We now illustrate the use of local variables and non-inflationary variables in T-FIXPOINT by means of the following two examples. We first illustrate local variables.

Example 5.3 Assume the database scheme contains two unary relations S and T. One way to express the temporal logic query $\{x \mid S(x) \,\mathsf{until}\, T(x)\}$ in T-FIXPOINT is as follows:

state $Mark(0)$;
shared $N(1)$, $A(1)$;
$Mark \mathrel{+}= \{()\}$;
$N \mathrel{+}= \neg S$;
$A \mathrel{+}= \neg N \cap T$
while $\neg Last$ **do**
  right;
  $N \mathrel{+}= \neg S$;
  $A \mathrel{+}= \neg N \cap T$
od;
while $\neg Mark$ **do left od**.

In the above program, *Mark* is a (nullary) state relation which is used to mark the initial state. Relations A and N are shared: A is the answer relation, and N keeps track of the elements that are not in S in some state encountered so far; if x is in N the first time it is found to be in T, x does *not* satisfy $S(x) \,\mathsf{until}\, T(x)$. The final while-loop returns to the marked state (the use of this will become clear immediately).

Suppose now that we have an additional third unary database relation R, and we want to express the more complex temporal logic query $\{x \mid R(x) \,\mathsf{until}\, (S(x) \,\mathsf{until}\, T(x))\}$. A simply way to do this would be to use the above program as a subroutine. However, in doing this, care must be taken that the auxiliary relations *Mark*, A and N are cleared after each invocation of the subroutine. This is precisely the facility provided by the local variables in T-FIXPOINT. Written out in full, we can thus express the query in T-FIXPOINT as follows:

shared $N_0(1)$, $A_0(1)$;
$N_0 \mathrel{+}= \neg R$;
P;
while $\neg Last$ **do**
  right;
  $N_0 \mathrel{+}= \neg R$;
  P
od,

where P is the following program block:

[**local** state $Mark(0)$;
  local shared $N(1)$, $A(1)$;
  $Mark \mathrel{+}= \{()\}$;
  $N \mathrel{+}= \neg S$;
  $A \mathrel{+}= \neg N \cap T$;
  while $\neg Last$ **do**
    right;
    $N \mathrel{+}= \neg S$;
    $A \mathrel{+}= \neg N \cap T$
  od;
  while $\neg Mark$ **do left od**;
  $A_0 \mathrel{+}= \neg N_0 \cap A$
]. ■

We next illustrate the kind of computations that can be performed using noninflationary variables.

Example 5.4 Assume the database scheme consists of a single binary relation R. Consider the program:

noninf shared $S(2)$;
$S := R$;
while $\neg Last$ **do**
  right;
  $S := \{x, y \mid (\exists z)(S(x, z) \wedge R(z, y))\}$
od.

At the end, if the last state of the temporal database is numbered n, S contains the set of pairs (x_0, x_n) such that there exist $x_0, x_1, \ldots, x_n$ with such that (x_i, x_{i+1}) is in R in the i-th state, for each $i \in \{1, \ldots, n\}$. ■

6 Comparisons

In this section, we first show that the expressive power of T-FIXPOINT lies between TS-FO and TS-FIXPOINT. Then we show that ETL can be simulated in T-FIXPOINT. Finally, we compare T-FIXPOINT and T-WHILE.

Theorem 6.1 TS-FO *is strictly contained in* T-FIXPOINT.

Proof. Each timestamp variable is represented by a nullary state relation which is true exactly in the state numbered by the current value of the variable, plus all states to the left of that state. The simulation now proceeds by induction on the structure of the formulas. We show that for each TS-FO formula φ with free data variables $x_1, \ldots, x_k$ and free time variables $t_1, \ldots, t_l$, there is a T-FIXPOINT program which computes the relation consisting of all data variables $x_1, \ldots, x_n$ for which φ is true, when the time variables $t_1, \ldots, t_n$ are fixed. The basis consists of atomic formulas. An atomic formula $S(x, t)$ is simulated by searching for the state where t is true and returning S in that state. A comparison $t < t'$ between timestamp variables is simulated by a left-to-right scan checking whether t is true before t'.

The induction is then clear if the formula φ consists of a disjunction, negation, and existential quantification of data variables which are simulated using union, complementation, and projection as usual. Finally, existential quantification of a timestamp variable is performed by a while-loop which repeatedly sets the variable true from left to right, and computes the disjunction of all the partial results .

The inclusion is strict because we will see later that T-FIXPOINT can simulate ETL, and we already know that there are queries expressible in ETL but not in TS-FO. ■

Theorem 6.2 T-FIXPOINT *is contained in* TS-FIXPOINT.

Proof. The simulation is analogous to that of T-WHILE by TS-WHILE in the proof of Proposition 4.3. The local and noninflationary relation variables of the T-FIXPOINT program can be handled by Proposition 5.2. The only difficulty that arises is the unary relation *Current* which is used in an entirely non-inflationary manner. We cannot simply change this relation into a non-inflationary one and apply Proposition 5.2, since a T-FIXPOINT program must be able to move in time (to be simulated by the relation *Current*) without

changing any of its inflationary relation variables. However, the semantics of T-FIXPOINT guarantees that such behavior can only last for at most n steps, where n is the length of the temporal database. Hence, instead of using a unary relation for *Current*, we can use a binary one which is organized as a linear order and is versioned by the tuples inserted in the inflationary relation variables, as in the proof of Proposition 5.2. The current position is always the maximum element in the order. Initially, *Current* contains $(1, 1)$; to simulate a move to the right the tuples $(1, 2)$ and $(2, 2)$ are added, and so on. This can go on until a move in the opposite direction occurs; then a new version is created with initial contents (i, i) where i is the new current position. Now repeated moves either to the left or the right can be recorded in the same orderly fashion, again until a move in the opposite direction occurs, after which again a new version is created, and so on. ■

It is not clear whether the converse of Theorem 6.2 holds. This is again because of the linear space complexity in the number of states of T-WHILE (and hence also of T-FIXPOINT) programs already mentioned in the proof of Proposition 4.3. Indeed, we can reduce the containment of TS-FIXPOINT in T-FIXPOINT to the containment of PTIME in the complexity class PLINSPACE which we define as follows:

> A problem is in PLINSPACE if it can be solved by a Turing machine in polynomial time using only linear space.

Observe that if PTIME is included in PLINSPACE, then in particular, PTIME is included in LINSPACE which is an open question of complexity theory. We observe:

Lemma 6.3 Every PLINSPACE query on ordered temporal databases is expressible in T-FIXPOINT.

Here, by an *ordered* temporal database we mean that a total order on the active domain is explicitly given by some relation, the same in all states.

Proof. The structure of the proof is the same as that of the proof presented in [1, Chapter 17.4] of the well-known fact that the language FIXPOINT can express any PTIME query on ordered relational (non-temporal) databases.

Let Q be a PLINSPACE query on ordered temporal databases. To each ordered temporal database $\mathbf{I}$, Q associates an answer relation $Q(\mathbf{I})$, of some

fixed arity, on the active domain of $\mathbf{I}$. Moreover, there is a polynomial-time, linear-space Turing machine M which, given as input an encoding of some $\mathbf{I}$, produces as output an encoding of $Q(\mathbf{I})$.

We will show there exists a T-FIXPOINT program q_M expressing Q in three phases: (1) construct an encoding of $\mathbf{I}$ that can be used to simulate M; (2) simulate M; and (3) decode the output of M.

We assume the reader is familiar with a standard way of encoding an ordinary (i.e., non-temporal) relational database on a Turing machine tape [1, Chapter 17.4]. Now recall that a temporal database is a sequence of relational databases (states) of a common schema and over a common domain of data elements. Let d be the number of data elements and let n be the length of the sequence. We assume that a temporal database is encoded on a Turing machine tape simply as the sequence of encodings of its states. The size of this encoding is $O(nd^k)$ for some fixed natural number k.

Since M uses linear space, we need to be able to represent, in q_M, a tape of length nd^k. This can be done by using several k-ary non-inflationary auxiliary state relation variables: one with name $\hat{\ell}$ for each letter ℓ of the tape alphabet, and one with name *Head*. For example, assume the $id^k + j$-th cell on the tape contains the letter ℓ, with $0 \leq i \leq n-1$ and $1 \leq j \leq d^k$. This is represented by having the tuple $(a_1, \ldots, a_k)$ in the contents of $\hat{\ell}$ at the $i+1$-th state, where $(a_1, \ldots, a_k)$ is the j-th tuple in the lexicographic ordering of k-tuples of data elements according to the given total order on the active domain. The position of the Turing machine head on the tape is represented using relation *Head* in a similar manner.

Since M runs in polynomial time, the length of its computation is bounded by $(nd^k)^l$ for some fixed natural number l. To represent a clock ticking precisely this many times, we use l auxiliary state relation variables $A_1, \ldots, A_l$ of arity k. These variables will be local to nested while-loop blocks. The nested blocks encapsulate the actual simulation of M in q_M, and clock the simulation as shown schematically below for $l = 2$:

```
[ local state A_1(k);
  while change do
    in the first state where A_1 is not yet full,
       add the lexicographically first k-tuple
       not yet in A_1 to A_1;
    [ local state A_2(k);
      while change do
```

```
            in the first state where A_2 is not yet full,
              add the lexicographically first k-tuple
              not yet in A_2 to A_2;
            Simulate the next step of M's computation
          od
        ]
      od
].
```

The actual construction of the encoded database on the input tape (using the representation described above) as well as the actual simulation of M's configuration transitions and the final decoding phase, are very much standard [1, Chapter 17.4]. The only non-standard aspect is that here, the program q_M must use the T-FIXPOINT capability of moving over the time instants to access the various portions of the simulated tape. ■

Theorem 6.4 *Assuming ordered databases,* TS-FIXPOINT = T-FIXPOINT *if and only if* PTIME = PLINSPACE.

Proof. *If.* Consider a TS-FIXPOINT query Q. Then Q is in PTIME. Note that this means that Q is computable by a polynomial-time Turing machine working on an encoding of the timestamp representation of the input temporal database. However, such a machine can be readily modified so as to work on the direct encoding of the temporal database used to prove Lemma 6.3. Moreover, since we assume PTIME = PLINSPACE, the machine can be assumed to work in linear space. Lemma 6.3 then shows that PLINSPACE queries can be computed in T-FIXPOINT. Thus Q is in T-FIXPOINT.

Only if. Let Q be a set of binary words decidable in PTIME. Consider the coding of Q as a Boolean query on temporal databases over a scheme consisting of a single relation name T, of arity 0; a word $x_1 \ldots x_n$ is represented by the database $I_1 \ldots I_n$, where $I_j(T) = \emptyset$ if $x_j = 0$ and $I_j(T) = \{()\}$ (the empty tuple) if $x_j = 1$, for $j = 1, \ldots, n$. The timestamp representations of such databases are ordered relational databases, since the order on the states is given and there are no data elements. As mentioned in the beginning of Section 5, any PTIME query on ordered databases is expressible in FIXPOINT. Hence, Q can be computed by a TS-FIXPOINT-program, and thus by our assumption, also by a T-FIXPOINT-program. This program runs in polynomial time, and since the active domain of each database is empty, it uses only linear space. Thus, Q is in PLINSPACE. ■

Theorem 6.5 ETL *is strictly contained in* T-FIXPOINT.

Proof. The simulation of ETL in T-FIXPOINT is analogous to the simulation of TL in T-FIXPOINT illustrated in Example 5.3. To simulate a temporal operator associated to a regular language L, we consider a finite automaton accepting L. For each state of the automaton we use an auxiliary relation playing a role similar to N in Example 5.3, keeping track of the status of the elements during the simulation of the automaton. The state-changes of the automaton are performed while moving over the states of the temporal database. The state-changing relations must be implemented using non-inflationary variables, since the working of the automaton is not inflationary.

To show that the inclusion is strict, one may want to argue simply that in T-FIXPOINT one can compute the transitive closure of a binary relation, which is impossible in ETL (on temporal databases of length one, ETL collapses to ordinary first-order logic). However, this argument is insufficiently general because it does not apply in the case of unary or nullary relational schemas. Instead, we show that it is possible in T-FIXPOINT to check whether the length of the temporal database is a prime number. This is impossible in ETL, since ETL is known [7] to be able to express only regular properties of the length of a database (representing a number as a unary word). Actually, we will show how to express the complementary query, checking whether the length is a composite number.

Consider the algorithm shown in Figure 3, which tests whether a natural number $n > 2$ is composite. This algorithm is special in that the auxiliary variables it uses take only values between 1 and n; the only test it uses is equality between one variable and another or n; and the only operations it uses is assigning one variable to another, incrementing a variable by one, and setting a variable to one.

We can simulate the algorithm of Figure 3 by a program in T-FIXPOINT. A variable having a value i between 1 and n (the length of the temporal database) can be simulated by a nullary state relation variable whose value is $\{()\}$ (the non-empty nullary relation, used as the truth value "true") in state i and $\emptyset$ (used as the truth value "false") in all other states. The simulation is shown in Figure 4. Nullary relation variables are used as propositional variables in the obvious manner. ■

Finally, we compare T-FIXPOINT to T-WHILE. It is quite easy to see that their equality is very unlikely:

```
begin
composite := false;
factor := 1; factor := factor + 1;
while not composite and factor ≠ n do
  product := factor;
  while product ≠ n do
    counter := 1;
    product := product + 1;
    while counter ≠ factor and product ≠ n do
      counter := counter + 1;
      product := product + 1
    od;
    if counter = factor and product = n then
      composite := true
  od;
  factor := factor + 1
od
end.
```

Figure 3: A special algorithm for testing compositeness of a natural number n.

```
shared Composite(0); state Factor(0);
right; Factor := true;
while ¬Composite ∧ ¬Last do
[ local state Product(0);
  while ¬First do left od;
  Product := true;
  while ¬Last do
  [ local state Counter(0);
    local shared Counter_eq_Factor(0);
    while ¬First do left od;
    Counter := true;
    while ¬Product do right od;
    Product := false; right; Product := true;
    while ¬Counter_eq_Factor ∧ ¬Last do
      while ¬Counter do left od;
      Counter := false; right; Counter := true;
      Counter_eq_Factor := Factor;
      while ¬Product do right od;
      Product := false; right; Product := true
    od;
    Composite := Counter_eq_Factor ∧ Last ]
  od;
  while ¬Factor do left od;
  Factor := false; right; Factor := true ]
od.
```

Figure 4: Program in T-FIXPOINT for testing compositeness of the length of the temporal database.

Proposition 6.6 *If* T-FIXPOINT = T-WHILE, *then* PTIME = PSPACE.

Proof. Suppose that T-FIXPOINT = T-WHILE. Then, in particular, T-FIXPOINT equals T-WHILE on temporal databases consisting of a single state, and hence, FIXPOINT equals WHILE. As mentioned in the beginning of Section 5, this is known to imply PTIME = PSPACE. ■

It remains open whether the converse of the above proposition holds.

7 Simulating timestamps by data elements

For clarity, we have separated the data elements in a temporal database from the natural numbers used to number its states. If, however, one allows these natural numbers to be stored in the database instances, interesting cases can be indicated in which the differences between implicit and explicit access to time disappear.

A class of situations in which TS-FO is no longer more powerful than TL is given by the following general definition. Let $\phi(\bar{z})$ be an arbitrary fixed TL-formula. For each state j of a temporal database $\mathbf{I}$, ϕ defines a relation $\phi(I_j)$ on the j-th instance I_j. If $\phi(I_j)$ and $\phi(I_\ell)$ are non-empty and disjoint for any two different states j and ℓ, $\mathbf{I}$ is called *ϕ-diverse*. We observe:

Proposition 7.1 *Let ϕ be a* TL*-formula. On ϕ-diverse databases,* TL *is equivalent to* TS-FO.

Proof. We show inductively how a TS-FO-formula γ can be translated into an equivalent TL-formula γ'. For simplicity we assume the schema consists of a single relation S. We will use the abbreviation $\Diamond\varphi$ for $\mathbf{true}\ \mathsf{until}\ \varphi$.

- To each time-variable t of TS-FO we associate distinct variables $z_1^t, \ldots, z_m^t$, where m is the number of free variables of ϕ. We denote the tuple $z_1^t, \ldots, z_m^t$ by $\bar{z}_t$.
- An atomic formula $\bar{S}(\bar{x}, t)$ is translated into $\Diamond(S(\bar{x}) \wedge \phi(\bar{z}_t))$.
- An atomic formula $t < t'$ is translated into $\Diamond(\phi(\bar{z}_t) \wedge \mathsf{next} \Diamond \phi(\bar{z}_{t'}))$.
- $\varphi \wedge \psi$ and $\neg\varphi$ are translated into $\varphi' \wedge \psi'$ and $\neg\varphi'$, respectively.
- Finally, $(\exists t)\varphi$ is translated into $(\exists \bar{z}_t)\varphi'$. ■

Two examples of ϕ-diverse databases are the following:

- Assume the database schema contains a unary relation *Time*, and assume the contents of that relation at the i-th state is the singleton $\{i\}$. Temporal databases of this kind are said to have *local time*. Since local-time databases are ϕ-diverse, with ϕ simply being $Time(z)$, the above proposition yields that TL is equivalent to TS-FO on local-time databases. The local time assumption is quite realistic in practice, and has been made, e.g., by Gabbay and McBrien [8]. It also seems to be implicitly made by Tuzhilin and Clifford [18]. Proposition 7.1 thus provides an a posteriori justification of the, at first sight erroneous, expressive completeness claims on TL made in [8, 18].

- *Insert-only* databases are databases where for each j, the instance at state $j+1$ is obtained from the instance at state j by inserting a non-zero number of tuples in some of the relations. Insert-only databases are φ-disjoint with φ being $\bigvee(R(\bar{x}) \wedge \neg \mathsf{previous}\, R(\bar{x}))$ (where the disjunction is over all relations R in the schema).

By an analogous proof to that of Proposition 7.1 we also readily see that on ϕ-diverse databases, T-FIXPOINT is equivalent to TS-FIXPOINT and that T-WHILE is equivalent to TS-WHILE. Moreover, in these query languages, ϕ-diversity can sometimes be simulated, as shown in the following:

Proposition 7.2 *Let p be a natural number. On ordered temporal databases of length at most d^p, where d is the size of the active domain,* T-WHILE *is equivalent to* TS-WHILE *and* T-FIXPOINT *is equivalent to* TS-FIXPOINT.

Proof. We can turn a database satisfying the property expressed in the proposition into an $A(\bar{z})$-diverse database, where A is a p-ary auxiliary state relation, defined using a T-FIXPOINT-program which generates the p-tuples of data elements one after the other in lexicographical order while moving over the temporal database from left to right and assigning them to the state relations A. ■

We conclude this brief section by noting that a result more general than Proposition 7.2 can be proven. Indeed, the proposition remains true without the assumption that the database is ordered, if we replace d by i, where i is the number of k-types in the database for some k. (For the definition of

k-types we refer to [1, 3].) This is because the collection of k-types, with an order on them, can be computed in T-FIXPOINT, in much the same way this can be done in FIXPOINT on non-temporal databases.

8 Concluding remarks

The main technical problem left open by our work is to determine whether or not the converse to Proposition 6.6 holds. One way to approach this problem is by trying to adapt the known proof [3] that PTIME = PSPACE implies FIXPOINT = WHILE (on non-temporal databases) to the temporal setting.

Another natural open research issue is to further relate the FIXPOINT- and WHILE-based temporal query languages proposed in this paper to other temporal query languages with iteration or recursion capabilities considered in the literature. The prime example of such a languge is (the first-order version of) *fixpoint temporal logic* (denoted μTL and proposed by Vardi [19]). This language is clearly subsumed by TS-FIXPOINT, but its exact relationship to TS-FIXPOINT as well as to T-FIXPOINT remains open. Other interesting languages are *Templog*, a logic-programming language based on TL, and $Datalog_{1S}$, which extends Datalog with the successor function on timestamps. A comprehensive presentation of these two languages was given by Baudinet, Chomicki, and Wolper [4], who also showed that they are equivalent to each other, and that in the propositional case, they are equivalent to the positive fragment of μTL. An important feature of $Datalog_{1S}$ is that programs can use the successor function on timestamps in an unbounded way; it is not only given on the set $\{1, \ldots, n\}$ of states of the input temporal database, but on the whole of the natural numbers. The infinite timestamped relations that can result from this can always be finitely represented, as shown by Chomicki and Imieliński [6]. It is not difficult to simulate a bounded version of $Datalog_{1S}$, where the successor function is only defined on the finite set of states of the input, in T-FIXPOINT.

We conclude this paper with a discussion on our proof of the separation of TS-FO from TL and ETL, presented in Section 3. An alternative approach to establish this result would be to prove that TS-FO^3, the 3 time-variable fragment of TS-FO, is strictly less expressive than full TS-FO. Indeed, it is known and not difficult to verify that every TL query is already expressible by a formula in TS-FO using at most 3 distinct time-variables. Note that our

proof of Theorem 3.8 implies that TL is strictly contained in TS-FO^3; actually, the proof shows that even some TS-FO^2 queries are not expressible in TL.

More generally, one might conjecture that there is a strict hierarchy in expressive power among the fragments TS-FO^k for each k. (It is known that TS-FO^1 $\subset_{\neq}$ TS-FO^2 $\subset_{\neq}$ TS-FO^3.) A closely related question from the field of finite model theory is whether there is a strict FO^k-hierarchy on the class of ordered finite graphs. Here, FO^k denotes the k variable fragment of standard first-order logic on ordered graphs.

One might also try to separate TL and TS-FO with a proof based on Ehrenfeucht-Fraïssé style games. Segoufin [16] designed a very elegant extension of Ehrenfeucht-Fraïssé games capturing precisely the expressive power of TL. In our experience, however, it is quite hard to explicitly construct families of pairs of temporal databases that are indistinguishable in TL. Our approach based on communication complexity turned out to be more successful. Our proof is robust under built-in relations on data elements, such as total order, and at the same time separates the more powerful ETL from TS-FO.

Acknowledgment

We thank Victor Vianu and Luc Segoufin for helpful discussions and encouragements during the course of this work. The third author also thanks Frank Neven for proofreading a draft of this paper.

References

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

[2] S. Abiteboul and E. Simon. Fundamental properties of deterministic and nondeterministic extensions of Datalog. *Theoretical Computer Science*, 78:137–158, 1991.

[3] S. Abiteboul and V. Vianu. Computing with first-order logic. *Journal of Computer and System Sciences*, 50(2):309–335, 1995.

[4] M. Baudinet, J. Chomicki, and P. Wolper. Temporal deductive databases. In A. Tansel et al., editors, *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993.

[5] J. Chomicki. Temporal query languages: a survey. In D.M. Gabbay and H.J. Ohlbach, editors, *Temporal Logic: ICTL'94*, volume 827 of *Lecture Notes in Computer Science*, pages 506–534. Springer-Verlag, 1994.

[6] J. Chomicki and T. Imieliński. Finite representation of infinite query answers. *ACM Transactions on Database Systems*, 18(2):181–223, June 1993.

[7] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier, 1990.

[8] D. Gabbay and P. McBrien. Temporal logic and historical databases. In *Proceedings 17th International Conference on Very Large Databases*, pages 423–430, 1991.

[9] D.M. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic, Mathematical Foundations and Computational Aspects*, volume 1 of *Oxford Logic Guides*. Oxford University Press, 1994.

[10] D. Gabby, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Conference Record 7th ACM Symposium on Principles of Programming Languages*, pages 163–173, 1980.

[11] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[12] H. Kamp. Formal properties of 'now'. *Theoria*, 37:227–273, 1971.

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

[14] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[15] D. Perrin. Finite automata. In *Handbook of Theoretical Computer Science*, volume B. Elsevier, 1990.

[16] L. Segoufin. Temporal logic and games. INRIA, VERSO, 1995.

[17] D. Toman and D. Niwinski. First-order queries over temporal databases inexpressible in temporal logic. In P.M.G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology—EDBT'96*, volume 1057 of *Lecture Notes in Computer Science*, pages 307–324. Springer, 1996.

[18] A. Tuzhilin and J. Clifford. A temporal relational algebra as a basis for temporal relational completeness. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 13–23. Morgan Kaufmann, 1990.

[19] M.Y. Vardi. A temporal fixpoint calculus. In *Proceedings 15th ACM Symposium on Principles of Programming Languages*, pages 250–259, 1988.

[20] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–93, 1983.

[21] A. C.-C. Yao. Some complexity questions related to distributive computing. In *Proceedings 11th ACM Symposium on the Theory of Computing*, pages 294–300, 1979.