Temporal connectives versus explicit timestamps in temporal query languages

(preliminary report)

Serge Abiteboul, Laurent Herr, Jan Van den Bussche
INRIA Rocquencourt

Contact address: Laurent Herr, INRIA (Projet VERSO), Domaine de Voluceau, Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France. Tel.: +33-1-39635275. Fax: +33-1-39635674. E-mail: laurent.herr@inria.fr.

Temporal connectives versus explicit timestamps in temporal query languages
(preliminary report)

Abstract

Some temporal query languages work directly on a timestamp representation of the temporal database, while others provide a more implicit access to the flow of time by means of temporal connectives. We study the differences in expressive power between these two approaches. We first consider first-order logic (i.e., the relational calculus). We show that first-order future temporal logic is strictly less powerful than first-order past-future temporal logic, and that the latter is strictly less powerful than the relational calculus with explicit timestamps. We also consider extensions of the relational calculus with iteration constructs such as least fixpoints or while-loops. We again compare augmentations of these languages with temporal left and right moves on the one hand, and with explicit timestamps on the other hand. For example, we show that a version of fixpoint logic with left and right moves lies between the explicit timestamp versions of first-order and fixpoint logic, respectively.

1 Introduction

A simple, natural and common way of representing a temporal relational database over a finite time period is to augment each relation with a "timestamp" column holding the time instants of validity of each tuple. Such representations can then be queried using standard relational query languages with built-in linear order on the timestamps. An alternative way of providing temporal facilities to query languages is by means of constructs yielding a more implicit access to the flow of time, such as the typical temporal connectives *next*, *previous*, *until*, and *since* of temporal logic [Eme90]. Illustrations of these two approaches can be found in [T$^+$93]. The question of how temporal connectives versus explicit timestamps in temporal query languages relate to each other with respect to expressive power, arises naturally [Cho94, CCT94]. In the present paper, we study this question from various angles.

First, we compare the relational calculus (i.e., first-order logic) with timestamps (TS-FO) to past-future first-order temporal logic (TL). We start by showing that *future* first-order temporal logic (FTL) is strictly weaker than TL. This should be contrasted with the conventional propositional case, where past connectives are known to be redundant [Eme90]. We then show that TL in turn is strictly weaker than TS-FO. Again this should be contrasted with the propositional case, where TL and TS-FO are known to be equivalent [Eme90].

We also look at more powerful languages, in particular, the languages WHILE and FIXPOINT which augment the relational calculus with while-loops and with inflationary fixpoint iteration, respectively [AHV94]. When used on timestamp representations of temporal databases, they give rise to powerful temporal query languages denoted by TS-WHILE and TS-FIXPOINT. Alternatively, they can be extended with a more implicit access to the flow of time. We study their extension with instructions for moving left and right in time, giving rise to temporal query languages denoted by T-WHILE and T-FIXPOINT. In the case of T-FIXPOINT, this involves extra non-inflationary language features which are interesting in their own right.

We compare TS-WHILE, TS-FIXPOINT, T-WHILE, and T-FIXPOINT. We also compare T-FIXPOINT with TS-FO, and with ETL, an "extended" temporal logic which is closely related to fixpoint extensions to temporal logic proposed in the propositional setting [Wol83, Var88, CF84]. Our results are summarized 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 TL, TS-FO, and ETL.

The paper is organized as follows. In Section 2, we define temporal databases and timestamp representations, recall TL, and compare it to TS-FO. In Section 3, we recall WHILE, introduce T-WHILE, and compare it to TS-WHILE and TS-FO. In Section 4, we recall the inflationary language FIXPOINT, study its augmentation with certain non-inflationary features, and introduce the central language T-FIXPOINT. In Section 5, we compare T-FIXPOINT to all other languages. Finally, in Section 6, we indicate special cases of temporal databases (including some notion of "local time") where the distinction between explicit versus implicit access to time in the context of WHILE and FIXPOINT disappears. In this preliminary report, the proofs of the technical results are only sketched.

2 First-order temporal queries

A *temporal database* is a non-empty finite sequence $\mathbf{I} = I_0, \ldots, I_n$ of instances of a common relational database scheme. Each element of the sequence is called a *state*. We can represent this sequence $\mathbf{I}$ as one single instance $\bar{\mathbf{I}}$ of an extended scheme, where each relation is extended with an additional column holding for each tuple the numbers of the states at which the tuple belongs to the relations. We call $\bar{\mathbf{I}}$ the *timestamp representation of* $\mathbf{I}$. More formally, for each relation R,

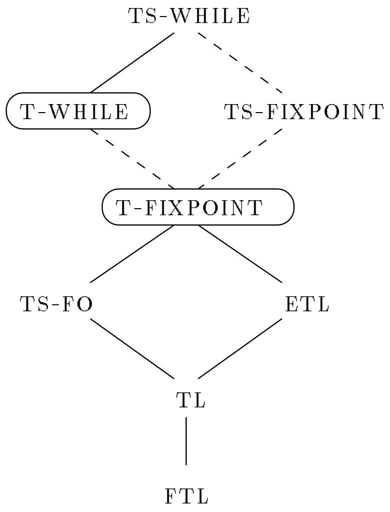

Figure 1: The relative power of temporal languages. Solid upward edges indicate strict containment. Dashed lines indicate that the strictness of the containment depends on unresolved questions in complexity theory.

we have an extended relation $\bar{R}$ whose value in $\bar{\mathbf{I}}$ is:

$$\bar{\mathbf{I}}(\bar{R}) := \bigcup_{t=1}^{n} I_t(R) \times \{t\}.$$

Furthermore, the total order on the timestamps is given in $\bar{\mathbf{I}}$ in a binary relation $<$.

Example 1 Consider the scheme consisting of a single unary relation S. Temporal databases over this scheme are basically sequences of sets. An example and its timestamp representation are shown in Figure 2. ■

A very direct way to express temporal queries on $\mathbf{I}$ is to use first-order logic (i.e., relational calculus) on the timestamp representation $\bar{\mathbf{I}}$. We denote this temporal query language by TS-FO.

Example 2 On databases as in Example 1, the query "give the elements that have been deleted but have later been re-inserted" can be expressed in TS-FO as

$$\{x \mid (\exists t)(\exists t')(\exists t'')(t < t' \wedge t' < t'' \wedge S(x,t) \wedge \neg S(x,t') \wedge S(x,t''))\}.$$

On the database of Figure 2, the answer to this query is $\{a, c\}$. ■

$$\mathbf{I} = \begin{array}{ccc} I_0 & I_1 & I_2 \\ \boxed{\begin{array}{c} a \\ b \\ c \end{array}} & \boxed{\begin{array}{c} b \\ d \end{array}} & \boxed{\begin{array}{c} a \\ c \end{array}} \end{array} \qquad \boxed{\begin{array}{cc} a & 0 \\ b & 0 \\ c & 0 \\ b & 1 \\ d & 1 \\ a & 2 \\ c & 2 \end{array}}$$

Figure 2: A temporal database and its timestamp representation.

Note that queries expressed in TS-FO can also return timestamps. For example, the query $\{t \mid (\forall x)(S(x,t) \leftrightarrow S(x,0))\}$ returns the numbers of all states that are equal to the first state. However, in the remainder of this discussion, we focus on queries returning data elements only.

An alternative way of expressing temporal queries is by using first-order temporal logic (abbreviated TL). This language is defined on the original database scheme, and does not involve timestamps. Instead, it extends the relational calculus by adding the binary connectives **since** and **until**. Syntactically, they are used to build formulas in the same way as the connective $\wedge$: if φ and ψ are formulas, then so are φ **since** ψ and φ **until** ψ.

Semantically, TL formulas, when evaluated on a given temporal database $\mathbf{I} = I_0, \ldots, I_n$, are evaluated with respect to some given time instant $t \in \{0, \ldots, n\}$. Formally, let θ be a TL formula, and let v be a valuation assigning domain values in $\mathbf{I}$ to the variables that are free in θ. The satisfaction of θ on $\mathbf{I}$ at t under v, denoted $\mathbf{I}, t \models \theta[v]$, is defined as follows:

- If θ is a relational atom $R(x_1, \ldots, x_k)$, then $\mathbf{I}, t \models \theta[v]$ if $(v(x_1), \ldots, v(x_k)) \in I_t(R)$.
- If θ is an equality predicate $x = y$, or is of one of the forms $(\exists x)\varphi$, $(\forall x)\varphi$, $\neg\varphi$, or $\varphi \wedge \psi$, satisfaction is defined in the standard way.[1]
- If θ is of the form φ **until** ψ, then $\mathbf{I}, t \models \theta[v]$ if
$$\exists t'' > t : \mathbf{I}, t'' \models \psi[v] \wedge \forall t' : t < t' < t'' \Rightarrow \mathbf{I}, t' \models \varphi[v].$$
- Finally, if θ is of the form φ **since** ψ, then $\mathbf{I}, t \models \theta[v]$ if
$$\exists t'' < t : \mathbf{I}, t'' \models \psi[v] \wedge \forall t' : t'' < t' < t \Rightarrow \mathbf{I}, t' \models \varphi[v].$$

In the above, the quantifiers on t' and t'' naturally range over $\{0, \ldots, n\}$.

Conventionally, queries expressed in TL are evaluated in the first state. So, if θ is a TL formula with free variables $x_1, \ldots, x_k$, then the result of the query $\{x_1, \ldots, x_k \mid \theta\}$ on a temporal database $\mathbf{I}$ is the relation $\{v \mid \mathbf{I}, 0 \models \theta[v]\}$.

[1] Quantifiers range over the active domain.

Example 3 The query of Example 2 can be expressed in TL as

$$\{x \mid \textsf{sometimesfuture}(S(x) \wedge \textsf{sometimesfuture}(\neg S(x) \wedge \textsf{sometimesfuture}\, S(x)))\},$$

where $\textsf{sometimesfuture}\,\theta$ is an abbreviation for $\theta \vee (\mathit{true}\ \textsf{until}\,\theta)$ (i.e., "θ is true now or some time in the future"). The other usual temporal connectives $\textsf{next}\,\theta$ ("θ holds in the next state"), $\textsf{alwaysfuture}\,\theta$ ("θ holds now and always in the future"), and the duals for the past (**previous**, **sometimespast** and **alwayspast**) are also easily definable in terms of **since** and **until**.

Since TL queries are evaluated in the first state, one may wonder if the past connective **since** is really necessary. Actually, in *propositional* TL (corresponding to the case where the database scheme consists of nullary relations[2] only), it is well-known that every formula is equivalent, with respect to the first time instant, to a pure future formula (i.e., using only **until** as temporal connective). We now show that this does not carry over to the setting of temporal databases:

Theorem 4 *The yes/no query Q: $(\exists t > 0)(\forall x)(S(x,t) \leftrightarrow S(x,0))$ is expressible in* TL *but not in pure future* TL*.*

Proof. (Sketch) We can express Q in TL as

$$\textsf{next sometimesfuture}(\forall x)(S(x) \leftrightarrow \textsf{sometimespast}(\mathbf{first} \wedge S(x))),$$

where **first** is the formula $\neg(\mathit{true}\ \textsf{since}\ \mathit{true})$ (this formula is only true in the first state).

To show that Q is not expressible in pure future TL, we use a combinatorial argument. Let θ be an arbitrary closed pure future TL formula. Let D be some arbitrary fixed finite domain of data elements, let d be the cardinality of D, and let n be some arbitrary fixed positive natural number. We consider temporal databases $I_0, \ldots, I_n$, and define the function F on the "tails" of such databases by

$$F(I_1, \ldots, I_n) := \{I_0 \mid (I_0, I_1, \ldots, I_n), 0 \models \theta\}.$$

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

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

However, it can be shown that the cardinality of the image of F is at most 2^{d^α}, for some integer α depending only on θ. The theorem then follows by elementary asymptotic calculus. ■

[2] A nullary relation can either contain the empty tuple or be empty, and can thus be used to represent a proposition (i.e., True or False).

Note that, by symmetry, pure past TL (evaluated in the last state) is also strictly weaker than TL. Pure past TL is often used to specify dynamic integrity constraints [Cho92].

We now turn to the question of how full past-future TL relates to TS-FO. Clearly, TS-FO is at least as expressive as TL. We were able to show this inclusion is strict:

Theorem 5 *The* TS-FO*-query* Q*:* $(\exists t)(\exists t')(\forall x)(S(x,t) \leftrightarrow S(x,t'))$ *is not expressible in* TL*.*

Proof. (Sketch) The proof focuses on a special kind of databases called *split databases*. A split database contains one state, called the "middle state", which is marked in some way so as to be easily recognizable among all other states. Hence, in a split database, the middle state splits the database in a "left part" and a "right part". We further require of a split database that both the left and the right part do not contain two states that are equal. Hence, on a split database, the query Q amounts to checking whether there is some state at the left that is equal to some state at the right. In other words, it must be checked whether the left and the right parts are disjoint, when viewed as sets rather than sequences of states.

This latter decision problem can also be viewed as a communication problem between two parties who both have a set of sets over a finite domain D. They do not know each other's set, but have to find out whether their sets are disjoint in a constant number of message exchanges. Each message is a k-ary relation on D, for some fixed k. Using a combinatorial argument, it can be seen that there is no communication strategy which can achieve this.

The proof can be concluded by observing that if Q would be expressible by a TL formula, then a successful strategy for the above communication problem would exist. Indeed, on split databases, temporal logic formulas can be rewritten into a formula where the temporal connectives **since** and **until** are split up in left and right versions. For instance, $\varphi\, \mathsf{until}_{right}\, \psi$ evaluated at a time instant t behaves like the normal $\varphi\, \mathsf{until}\, \psi$ if t is a position on the right, but when t is a position on the left, it behaves as $\varphi\, \mathsf{until}\, \psi$ when evaluated in the middle state. Similarly, we have since_{right}, until_{left}, and since_{left}. Now such a "split-TL" formula θ expressing query Q yields a successful communication strategy as follows. Proceeding in a bottom-up fashion, the parties evaluate all temporal subformulas of θ in the middle state. The party having the left (right) part of the split database evaluates all left (right) formulas. The results of the evaluations (being relations on the active domain of the database) are exchanged as messages. After a constant number of message exchanges (namely, the number of temporal subformulas of θ), θ can be evaluated in the middle state and the parties can thus determine whether their sets are disjoint or not. ∎

3 Iterative queries

Let us first briefly recall how relational calculus is extended with iteration to obtain the language WHILE. (See [AHV94] 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 a *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 with 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 output relation at completion of the execution[3].

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

Example 6 The query "give the elements that belong to all even-numbered states" is not expressible in the relational calculus with timestamps, but it is expressible in TS-WHILE as follows:

$Current := \{0\};$
$A := \{x \mid S(x, 0)\};$
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, A is the answer relation, and $t' = t + 2$ is only an abbreviation 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[4]. Furthermore, we partition the auxiliary relations into *state relations*, which are stored in the different states, and *shared relations*, which are stored in

[3]If the execution loops indefinitely, the result is defined to be empty by default. Such loops can always be detected [AS91].

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

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.

Example 7 The query from Example 6 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 $Even$ **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 8 T-WHILE *is strictly contained in* TS-WHILE.

Proof. (Sketch) The simulation of T-WHILE by TS-WHILE is straightforward, using a *Current* relation as in Example 6 holding the current temporal position of the machine.

The argument for strictness is based on complexity. The complexity of TS-WHILE programs in terms of the length n of the temporal database only is precisely PSPACE. 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 [HU79]. ■

4 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 $X \mathrel{+}= E$). Executing an inflationary WHILE program with all auxiliary relations initialized to the empty set will either finish or repeat a configuration after an at most polynomial number of steps.[5] The computation has then "reached a fixpoint" and the result of the query is determined. The query language thus obtained is therefore called FIXPOINT.

Actually, on *ordered* databases[6], 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 shown in [AV91] that this question is equivalent to the renowned open problem in computational complexity on the strict containment of PTIME in 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 9 The query of Example 6 can also be expressed in TS-FIXPOINT as follows:

$$
\begin{array}{l}
\mathit{Current} \mathrel{+}= \{0\};\\
B \mathrel{+}= \{x \mid \neg S(x,0)\};\\
\textbf{while } (\exists t)(\exists t')(\mathit{Current}(t) \wedge \neg \mathit{Current}(t') \wedge t' = t+2) \textbf{ do}\\
\quad \mathit{Current} \mathrel{+}= \{t' \mid (\exists t)(\mathit{Current}(t) \wedge t' = t+2)\};\\
\quad B \mathrel{+}= \{x \mid (\exists t)(\mathit{Current}(t) \wedge \neg S(x,t))\}\\
\quad \textbf{od};\\
A \mathrel{+}= \{x \mid \neg B(x)\}.
\end{array}
$$

■

Alternatively, 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 analysed using a complexity argument. As we have seen in Proposition 8 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.

This problem can be alleviated by adding two extra, non-inflationary features to standard FIXPOINT that allow to use non-inflationary variables in a controled 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

[5] As for WHILE, infinite loops can always be detected.

[6] This means that a linear order on the active domain is available.

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. Note that configurations of T-FIXPOINT programs now include the position of the machine in time, 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 standard querying.

Proposition 10 *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. (Sketch) 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 11 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 $N(1)$, $A(1)$;
$Mark \mathrel{+}= \{()\}$;
while $\neg Last$ **do**
  right;
  $A \mathrel{+}= \neg N \cap T$;
  $N \mathrel{+}= \neg S$
  od;
while $\neg Mark$ **do left**.

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

We next illustrate the kind of computations that can be performed using non-inflationary variables.

Example 12 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 $i \in \{0, \ldots, n-1\}$. ∎

5 Comparisons

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

We first show:

Theorem 13 TS-FO *is strictly contained in* T-FIXPOINT.

Proof. (Sketch) 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. 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'. Disjunction, negation, and existential quantification of data variables 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. Nested quantifiers are simulated using a nested program block for each quantifier, with the marking relation as a local variable to that block.

The inclusion is strict because TS-FO cannot compute the transitive closure of a graph stored in one of the states. ∎

Since TL is subsumed by TS-FO, as an immediate corollary we obtain that TL is strictly contained in T-FIXPOINT.

We next show:

Theorem 14 T-FIXPOINT *can be simulated in* TS-FIXPOINT.

Proof. (Sketch) The simulation is analogous to that of T-WHILE by TS-WHILE of Proposition 8. The relation *Current* used there is non-inflationary, but by Proposition 10 we know that this does not pose a problem. ∎

It is not clear whether the converse of Theorem 14 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 8. Indeed, we can reduce the containment of TS-FIXPOINT in T-FIXPOINT to the containment of PTIME in the following class of complexity:

> 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:

Theorem 15 *Assuming ordered databases,*[7] TS-FIXPOINT = T-FIXPOINT *if and only if* PTIME = PLINSPACE.

Proof: (Sketch) Suppose that PTIME = PLINSPACE, and consider a TS-FIXPOINT query Q. Then Q is in PTIME, so in PLINSPACE. It is possible (although somewhat intricate) to show that PLINSPACE queries can be computed in T-FIXPOINT. The linear tape of the Turing machine is simulated by splitting it into n pieces (where n is the number of states) and assigning one piece to each state of the database. Non-inflationary variables are used to simulate the non-inflationary nature of the Turing machine computation. Local variables are used to "count" the number of steps (polynomial) that the machine is allowed to perform. Thus Q is in T-FIXPOINT.

Conversely, suppose that TS-FIXPOINT = T-FIXPOINT. Let Q be a PTIME problem. Consider the coding of this problem as a query on a propositional temporal database (each letter in the input word is represented by one state). As mentioned in the beginning of Section 4, any PTIME query on ordered databases is expressible in FIXPOINT. Hence, Q can be computed by a TS-FIXPOINT-program. So Q can be computed by a T-FIXPOINT-program. This program runs in PTIME, and since the database is propositional, it uses only linear space. Thus, Q is in PLINSPACE. ∎

Fixpoint extensions of temporal logic have been studied extensively in the propositional case [Eme90]. One of these extensions is the *extended temporal logic* ETL [Wol83, Var88]. This language offers general temporal connectives expressed in terms of regular expressions. Indeed, the standard connective φ **until** ψ of TL corresponds to searching for the regular expression ab^*c, where the letter a stands

[7] This assumption means that a linear order is available on the active domain.

for *True*, b stands for φ, and c stands for ψ. It is not difficult to define ETL in the context of first-order predicate logic (i.e., databases) rather than propositional logic (e.g., [CF84]). We can show:

Theorem 16 ETL *is stricly contained in* T-FIXPOINT.

Proof. (Sketch) The simulation of ETL in T-FIXPOINT is analogous to the simulation of TL in T-FIXPOINT illustrated in Example 11. We now consider the finite automaton corresponding to the regular expression, and for each state of the automaton we use an auxiliary relation playing a role similar to N in Example 11, 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.

The inclusion is strict because TS-FO cannot compute the transitive closure of a graph stored in one of the states. (Actually, also on propositional databases the inclusion is strict: ETL can only recognize regular properties [Eme90], while it is possible to write a T-FIXPOINT program checking whether the length of the temporal database is a prime number.) ∎

Finally, we compare T-FIXPOINT to T-WHILE. It turns out that their equality is very unlikely:

Proposition 17 *If* T-FIXPOINT = T-WHILE, *then* PTIME = PSPACE.

Proof: (sketch) 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 4, this is known to imply PTIME = PSPACE. ∎

It remains open whether the converse of the above proposition holds. In some sense, the equality of T-FIXPOINT and T-WHILE could even be more unlikely that the equality of PTIME and PSPACE.

6 Local time

A temporal database $I = I_0, \ldots, I_n$ is said to have *local time* if at each state, the number of that state is stored in some relation. Formally, assume the database scheme contains a unary relation *Time*. Then I has local time if for each $t \in \{0, \ldots, n\}$, $I_t(\mathit{Time}) = \{t\}$. We naturally assume that the linear order on timestamps is available to query languages working on temporal databases with local time.

In practice, local time will often be present. It can be shown that on temporal databases with local time, T-WHILE is equivalent to TS-WHILE and T-FIXPOINT is equivalent to TS-FIXPOINT.

We will obtain this result as a corollary of the following much stronger result stating that, in some cases, it is possible to simulate local time using the data elements. Thereto, we need to assume that the temporal databases are ordered, i.e., that a linear order is available on the active domain. (We will remove this restriction later.)

Theorem 18 *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. (Sketch) First assume that local time is present. We already know that TS-WHILE can simulate T-WHILE and that TS-FIXPOINT can simulate T-FIXPOINT. To show the converse simulations, it suffices to show that the timestamp representation of a temporal database with local time can be constructed in T-FIXPOINT, since T-FIXPOINT is a sublanguage of T-WHILE. This is easily done using the following program (for simplicity assuming that the database scheme consists of a single relation S):

$$\textbf{while } \neg \mathit{Last} \textbf{ do } R \mathrel{+}= S \times \mathit{Time}; \textbf{ right od}$$

It now suffices to observe that local time can be simulated using the tuples in D^p. This is done by a straightforward T-FIXPOINT program which generates them one after the other in lexicographical order while moving over the temporal database from left to right. ■

Note that in order to prove Theorem 19 we do not even need the facilities of local and noninflationary variables in T-FIXPOINT.

In the above, we assumed that the domain of the database is ordered. Using a similar argument, the theorem remains true without the ordering assumption if we replace d^p by i^p, where i the *k-type index* of the database for some k. For the formal definition of type index we refer to [AHV94]; we simply recall that it is a polynomial in d on ordered databases, and that the k-type index[8] can be computed in FIXPOINT. Now for temporal databases, one can show that it can be computed in T-FIXPOINT; and it is easy to demonstrate that for databases with local time, the k-type index is always larger than the number of states. From these observations, it follows:

Corollary 19 *On temporal databases with local time,* T-WHILE *is equivalent to* TS-WHILE *and* T-FIXPOINT *is equivalent to* TS-FIXPOINT.

References

[AHV94] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1994.

[8] Actually, the collection of k-types with an order on them.

[AS91] S. Abiteboul and E. Simon. Fundamental properties of deterministic and nondeterministic extensions of Datalog. *Theoretical Computer Science*, 78:137–158, 1991.

[AV91] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings 23rd ACM Symposium on Theory of Computing*, pages 209–219, 1991.

[CCT94] J. Clifford, A. Croker, and A. Tuzhilin. On completeness of historical relational query languages. *ACM Transactions on Database Systems*, 19(1):64–116, 1994.

[CF84] A. Casanova and A. Furtado. On the description of database transition constraints using temporal constraints. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Data Base Theory*, pages 211–236. Plenum Press, 1984.

[Cho92] J. Chomicki. History-less checking of temporal integrity constraints. In *Proceedings 8th International Conference on Data Engineering*. IEEE, 1992.

[Cho94] 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.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16. Elsevier science publishers, 1990.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[T+93] A. Tansel et al., editors. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[Var88] M.Y. Vardi. A temporal fixpoint calculus. In *Proceedings 5th ACM Symposium on Principles of Programming Languages*, pages 250–259, 1988.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–93, 1983.