Temporal Query Languages Expressive Power: μTL vs. T-WHILE

Nicole Bidoit, Matthieu Objois
Laboratoire de Recherche en Informatique
L.R.I. UMR 8623 CNRS
Batiment 490 - Université Paris Sud
91405 Orsay Cedex - FRANCE
{bidoit,objois}@lri.fr

Abstract

We investigate the expressive power of implicit temporal query languages. The initial motivation was refining the results of [1] and enrich them with comparison to μTL *[17]. Thus, we address two classes of temporal query languages:* μTL*-like languages based on* TL *and* T-WHILE*-like languages based on* WHILE*. We provide a two-level hierarchy (w.r.t. expressive power) for these temporal query languages. One of the contributions solves an open problem: the relative expressivity of* μTL *and* T-FIXPOINT *[1, 11].*

1 Introduction

In this paper we focus our attention on temporal databases and temporal query languages. Two different representations of time are classically considered: the implicit framework defines a temporal database as a finite sequence of instances; in the explicit framework, tuples in relations are timestamped. It is now well-known that as far as representing time is concerned, the implicit and explicit representations of time are equally expressive. However, as far as temporal query languages are concerned, the choice between implicit and explicit frameworks leads to different languages w.r.t. expressive power. With implicit time, the linear temporal logic TL and its extensions are the basic formalisms underlying query languages [8, 7]. When time is explicitly represented, queries are specified using the language TS-FO (or extensions) which is simply the relational calculus FO with timestamps.

As pointed out in [7], temporal logic and their extensions are especially attractive as query languages for temporal databases because of their simplicity and computational advantages. Indeed, because the references to time are hidden, queries are formulated in an abstract, representation-independent way. As far as computational complexity is concerned, [15] shows that the satisfiability problem for propositional TL is PSPACE-complete (whereas the same problem for first-order theory is non elementary [13]).

The work of [1] (see also [5]) gives new insight to the relative expressive power of TL and TS-FO as query languages: surprisingly enough[1], first order TL is strictly less expressive than TS-FO. Subsequent research work [1, 11, 4] investigate query languages more expressive than TL. More recently, [16] has proved (based on the techniques of [5]) that TS-FO is strictly more expressive than any fixed-dimensional first-order temporal logic.

This paper focuses on the implicit languages introduced in [1] where the authors investigate a classical way to extend FO in order to build implicit temporal languages. These languages rely on traditional imperative mecanisms namely assignements, iterations (while statements) and temporal moves. These languages further denoted T-WHILE-like languages are also extensions of WHILE introduced in [6] to capture recursive queries in the static framework. In this paper, we address another class of temporal query languages called μTL-like languages. These languages can be viewed as extensions of TL [17, 11] or alternatively as extensions of FO with fixpoint formulas [14, 6, 9, 10, 2].

[1] provides an expressive power hierarchy of temporal languages including TL, TS-FO, T-WHILE-like languages and their explicit counterparts. Surprisingly enough [1, 11] do not address μTL-like languages in their expressive power comparative study.

The work presented in this paper aims at contributing to the following open problem pointed out in [7]: *"Fixpoint temporal logic* μTL *has been extensively used in program verification, although only in the propositional case. The first order version of* μTL *remains to be studied. In particular, its relationship to* T-WHILE *(...) needs to be elucidated."*

One of the main contributions of our paper is to enrich the expressive power hierarchy of [1] showing that the first order inflationary extension of μTL is strictly less expressive than T-FIXPOINT [2] which is roughly the inflationary T-WHILE-like language with some non-inflationary features.

[1]This result stands in contrast with the propositional case.

[2]In the paper, this language is renamed T-WHILE^+_u.

We also show that T-FIXPOINT is equivalent to the non inflationary T-WHILE which entails that T-FIXPOINT is not PTIME as claimed in [1].

The expressive power results presented in this paper shed new light on the impact of the features of the two classes of languages. In the static case, each result holding for a WHILE-like language also holds for its FO+fixpoint counterpart. Surprisingly enough, this symmetry is broken in the temporal framework: for instance, the inflationary T-WHILE-like language with some non-inflationary features (already mentionned above as T-FIXPOINT) is strictly more expressive than its μTL-like counterpart. Another interesting impact of our results is that they highlight the frontier between querying static databases and querying temporal databases. More precisely, part of our results relies on the assumption that the temporal instances under consideration are composed of at least two states as opposed to static ones composed of a single state.

The paper is organized as follows. Section 2 is devoted to preliminaries and section 3 to the presentation of the implicit temporal languages whose comparative expressive power are investigated in section 4. We conclude the paper in section 5 by discussing our results and addressing some open problems.

2 Preliminaries

We assume the reader familiar with both first-order logic FO and temporal logic TL and with the usual definitions of *relation schema*, *database schema* and *instances*. We denote the arity r of a relation schema R by R/r. In the whole paper, we consider the database schemas $\mathcal{R} = \{R_1, ..., R_r\}$ and $\mathcal{S} = \{S_1, ..., S_s\}$ and we assume a unique domain Dom.

An implicit temporal instance $\mathbb{I}$ over $\mathcal{R}$, is a finite sequence $I_1, ..., I_n$ of (finite) instances over $\mathcal{R}$. For each $i \in [1, n]$, I_i is called the state of $\mathbb{I}$ *at time point* i. The instance of the relation schema R at time point i is denoted by $\mathbb{I}(R)[i]$. The active domain of $\mathbb{I}$, denoted $adom(\mathbb{I})$, is the union of the active domains of the states I_i. The well known "unsafe query" problem is solved here by restricting variables to range over the active domain of the input temporal database.

In the paper, $\mathbb{I} = I_1, ..., I_n$ is a temporal instance of size n over $\mathcal{R}$, $\vec{x}$ represents an unspecified tuple of variables whose arity is clear from the context, and ν is a valuation of $\vec{x}$ ranging over $adom(\mathbb{I})$.

Let $\mathbb{J} = J_1, ..., J_n$ be a temporal instance of size n over the database schema $\mathcal{S}$ where $\mathcal{R} \cap \mathcal{S} = \emptyset$. Merging the instances $\mathbb{I}$ and $\mathbb{J}$ leads to the instance denoted $\mathbb{I} + \mathbb{J}$ over the database schema $\mathcal{R} \cup \mathcal{S}$, defined as $K_1, ..., K_n$ where for all $i \in [1, n]$: 1) for all $j \in [1, r]$, $K_i(R_j) = I_i(R_j)$ and 2) for all $j \in [1, s]$, $K_i(S_j) = J_i(S_j)$.

We denote $\varnothing_{\mathcal{R}}$ any temporal instance over $\mathcal{R}$ whose states are all empty.

3 Temporal query languages

In this section, we revisit the specification of some of the languages introduced in [17, 1, 11] in order to better identify their features, and we also introduce new languages. We proceed by introducing two classes of implicit temporal query languages, each one sharing the same paradigm: μTL-like languages presented in section 3.1 are both extensions of TL and of FO+fixpoint and T-WHILE-like languages presented in section 3.2 are temporal extensions of WHILE. Languages in a class differ by some features (e.g. *inflationism* vs. *non inflationism*) leading to alternative query languages. We now proceed to the presentation of the languages. The main expressivity results are developed and discussed in section 4.

3.1 μTL-like languages

All query languages presented in this section are based on the μTL language, introduced in the propositional case by Vardi in [17] and extended to the first order case by Herr in [11]. These languages can also be seen as temporal extensions of the static query language FO+fixpoint, which has a long history: it has first been considered over infinite stuctures in [14], and afterward studied in the finite case (which is relevant to the database context) in [6, 9, 10, 12].

Syntax The syntax of μTL-like languages over the database schema $\mathcal{R}$ is obtained by the formation rules for FO, together with the following additional rules:
• if φ is a formula then $prev(\varphi)$ and $next(\varphi)$ are formulas.
• suppose that $\mathcal{R} \cap \mathcal{S} = \emptyset$. We call *auxiliary schema* any schema in $\mathcal{S}$. We define a (simultaneous) inductive operator for the auxiliary schemas. Let $\varphi_{S_1}, ..., \varphi_{S_s}$ be formulas over the database schema $\mathcal{R} \cup \mathcal{S}$ [3]. Each formula φ_{S_p} has as many free variables as the arity of S_p. For all $m \in [1, s]$, the *μ-expression* $\mu fp[\varphi_{S_1}, ..., \varphi_{S_m}][\varphi_{S_{m+1}}, ..., \varphi_{S_s}](S_p)$ can be used like a relation schema to build a formula $\mu fp[\varphi_{S_1}, ..., \varphi_{S_m}][\varphi_{S_{m+1}}, ..., \varphi_{S_s}](S_p)(\vec{x})$. The schemas $S_1, ..., S_m$ (resp. $S_{m+1}, ..., S_s$) are called the *checked* (resp. *unchecked*) auxiliary schemas of the μ-expression. Note that μ-expressions can be nested.

Whether an auxilliary schema is checked or unchecked has an impact on the semantics of the language (see below). Intuitively, the former (resp. the latter) enforces the instance of the auxilliary schema to be used (resp. ignored) when computing the semantics of a μ-expression.

EXAMPLE 3.1 Let G (resp. S_1, S_2) be a binary relation schema in $\mathcal{R}$ (resp. $\mathcal{S}$) and consider the formulas [4]:
$\varphi_{S_1}(x, y) = (first \wedge G(x, y) \wedge x = a) \vee \exists z(prev(S_1(x, z)) \wedge G(z, y))$,
$\varphi_{S_2}(x, y) = (last \wedge S_1(x, y)) \vee \exists z(G(x, z) \wedge next(S_2(z, y)))$.

[3] Both $R_1, ..., R_r$ and $S_1, ..., S_s$ can appear in the formula φ_{S_p} for all $p \in [1, s]$.
[4] $first$ (resp. $last$) stands for $\neg prev(true)$ (resp. $\neg next(true)$).

The formula $\mu fpt[\varphi_{S_1}, \varphi_{S_2}][](S_2)(u,v)$ is referred to as $\psi(u,v)$ in the rest of the section. Intuitively, $\psi(u,v)$ computes a "back-and-forth temporal path" along the temporal instance, beginning with the value a (in the first state).

Semantics The *satisfaction* of a formula θ over $\mathbb{I}$ at time point $i \in [1,n]$ given a valuation ν, denoted $[\mathbb{I}, i, \nu] \models \theta$ is defined as follows:

- if θ is obtained by a first-order rule, $[\mathbb{I}, i, \nu] \models \theta$ is defined as usual
- if θ is $prev(\varphi)$, $[\mathbb{I}, i, \nu] \models \theta$ iff $i > 1$ and $[\mathbb{I}, i-1, \nu] \models \varphi$
- if θ is $next(\varphi)$, $[\mathbb{I}, i, \nu] \models \theta$ iff $i < n$ and $[\mathbb{I}, i+1, \nu] \models \varphi$
- if θ is $\mu fpt[\varphi_{S_1}, ..., \varphi_{S_m}][\varphi_{S_{m+1}}, ..., \varphi_{S_s}](S_p)(\vec{x})$, we first need some preliminaries: let $(\mathbb{S}_k)_{k \in \mathbb{N}}$ be the sequence of temporal instances of size n over $\mathcal{S}$ defined by:

– $\mathbb{S}_0$ is the empty temporal instance $\varnothing_{\mathcal{S}}$

– $\mathbb{S}_{k+1}$ is the temporal instance defined for each time point $j \in [1,n]$ and each auxiliary schema S_q for $q \in [1,s]$ by:

$$\mathbb{S}_{k+1}(S_q)[j] = \{\rho(\vec{x}) \mid [\mathbb{I} + \mathbb{S}_k, j, \rho] \models \varphi_{S_q}(\vec{x}),\ \rho \text{ a valuation}\}.$$

Intuitively, $\mathbb{S}_{k+1}(S_q)[j]$ is the answer of the query φ_{S_q} evaluated over $\mathbb{I} + \mathbb{S}_k$ at time point j.

Next, the sequence $(\mathbb{S}_k^{ch})_{k \in \mathbb{N}}$ is the one obtained from $(\mathbb{S}_k)_{k \in \mathbb{N}}$ by projection over the checked auxiliary schemas $\{S_1, ..., S_m\}$. Suppose that $(\mathbb{S}_k^{ch})_{k \in \mathbb{N}}$ reaches a repetition [5], and consider the least $\mathbf{r} \in \mathbb{N}$ such that $\mathbb{S}_{\mathbf{r}}^{ch} = \mathbb{S}_{\mathbf{r}+1}^{ch}$. Intuitively, the unchecked auxiliary relations are "hidden" when checking for a repetition.
Thus when θ is $\mu fpt[\varphi_{S_1}, ..., \varphi_{S_m}][\varphi_{S_{m+1}}, ..., \varphi_{S_s}](S_p)(\vec{x})$, we have $[\mathbb{I}, i, \nu] \models \theta$ iff $\nu(\vec{x}) \in \mathbb{S}_{\mathbf{r}}(S_p)[i]$, that is iff the tuple $\nu(\vec{x})$ belongs to the relation over S_p of the temporal instance $\mathbb{S}_{\mathbf{r}}$ at time point i.

If $(\mathbb{S}_k^{ch})_{k \in \mathbb{N}}$ never reaches a repetition, we say that the semantics of both the μ-expression and the formula it appears in are undefined.

EXAMPLE 3.2 The table below illustrates the above semantics using the ψ formula of example 3.1. It gives a temporal instance for G and the sequence $(\mathbb{S}_k)_{k \in \mathbb{N}}$.

t	1		2		3	
$\mathbb{I}$	G	(a,b) (e,f)	G	(b,c) (d,e)	G	(c,d)
	S_1	S_2	S_1	S_2	S_1	S_2
$\mathbb{S}_0$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\mathbb{S}_1$	(a,b)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\mathbb{S}_2$	(a,b)	$\emptyset$	(a,c)	$\emptyset$	$\emptyset$	$\emptyset$
$\mathbb{S}_3$	(a,b)	$\emptyset$	(a,c)	$\emptyset$	(a,d)	$\emptyset$
$\mathbb{S}_4$	(a,b)	$\emptyset$	(a,c)	$\emptyset$	(a,d)	(a,d)
$\mathbb{S}_5$	(a,b)	$\emptyset$	(a,c)	(a,e)	(a,d)	(a,d)
$\mathbb{S}_6$	(a,b)	(a,f)	(a,c)	(a,e)	(a,d)	(a,d)
$\mathbb{S}_7$	(a,b)	(a,f)	(a,c)	(a,e)	(a,d)	(a,d)
	...		...		...	

The above computation reaches a repetition at iteration 6, because $\mathbb{S}_6 = \mathbb{S}_7$. Thus the evaluation of the instance over auxiliary schema S_1 (resp. S_2) is $\mathbb{S}_6(S_1)$ (resp. $\mathbb{S}_6(S_2)$); hence the evaluation of ψ over the above temporal instance is $\mathbb{S}_6(S_2) = (\{(a,f)\}, \{(a,e)\}, \{(a,d)\})$.

Let ψ' be obtained from ψ by declaring S_2 unchecked: $\psi'(u,v) = \mu fpt[\varphi_{S_1}][\varphi_{S_2}](S_2)(u,v)$. The computation of $(\mathbb{S}_k)_{k \in \mathbb{N}}$ is the same for both ψ and ψ', but such is not the case for the sequence $(\mathbb{S}_k^{ch})_{k \in \mathbb{N}}$. Indeed, $(\mathbb{S}_k^{ch})_{k \in \mathbb{N}}$ reaches a repetition at iteration 3, since $(\mathbb{S}_k^{ch})_{k \in \mathbb{N}}$ only contains the instance over the auxiliary schema S_1, and $\mathbb{S}_3(S_1) = \mathbb{S}_4(S_1)$. Thus the evaluation of ψ' over the above temporal instance is $\mathbb{S}_3(S_2) = (\emptyset, \emptyset, \emptyset)$.

Query languages We define queries based on the formulas defined above exactly like for TL. A query q is specified by a formula φ and the answer of q over $\mathbb{I}$ is obtained by evaluating φ at time point 1:

$$q(\mathbb{I}) = \{\nu(\vec{x}) \mid [\mathbb{I}, 1, \nu] \models \varphi,\ \nu \text{ a valuation}\}$$

Next, we use the notion of inflationary formulas. A formula φ_{S_p} is *inflationary* if, considering the sequence $(\mathbb{S}_k)_{k \in \mathbb{N}}$ defined above, $\mathbb{S}_k(S_p)[j] \subseteq \mathbb{S}_{k+1}(S_p)[j]$ for all $j \in [1,n]$. Hence an inflationary formula is intuitively a "cumulative" one.

We now define four μTL-like query languages based on the previous syntax and semantics.

1. μTL^+ denotes the language without unchecked auxiliary schemas and where all formulas of μ-expressions are *inflationary*.
2. μTL denotes the language without unchecked auxiliary schemas and with arbitrary formulas.
3. μTL_u^+ denotes the language where auxiliary schemas can be checked or unchecked, and where all formulas of μ-expressions "defining" some checked relations are *inflationary*.
4. μTL_u denotes the language where auxiliary schemas can be checked or unchecked, and with arbitrary formulas.

Note that, in all four languages, μ-expressions must contain obviously at least one auxiliary schema and moreover at least one auxiliary schema must be checked.

Repetition vs. limit The semantics of μ-expressions is defined using the repetition of the checked sequence $(\mathbb{S}_k^{ch})_{k \in \mathbb{N}}$. Another classical way to define such a semantics is by using the *limit* of the sequence. These two definitions lead to different semantics for μTL_u^+ and μTL_u, and they lead to equivalent query languages for μTL^+ and μTL. Indeed, μTL_u^++repetition and μTL_u^++limit (resp. μTL_u+repetition and μTL_u+limit) are probably not equivalent query languages. Hence, we beleive that only μTL^+ and μTL would deserve to be called « fixpoint query languages ».

It is easy to show that in the static case, the repetition and limit definitions lead to equivalent query languages for both the inflationary and non inflationary FO+fixpoint.

[5] We say that a sequence $(u_n)_{n \in \mathbb{N}}$ "reaches a repetition" iff there exists $r \in \mathbb{N}$ such that $u_r = u_{r+1}$.

Partial query languages There exists formulas belonging to μTL and μTL_u whose semantics is undefined, but every μTL^+ and μTL_u^+ formula admits a defined semantics. The facts that on the one hand μTL and μTL_u are partial languages and on the other hand μTL^+ and μTL_u^+ are total languages generalize the static case: recall that non inflationary FO+fixpoint is a partial query language and inflationary FO+fixpoint is a total query language.

3.2 T-WHILE-like languages

All query languages presented in this section are based on the T-WHILE language introduced in [1]. T-WHILE is a temporal extension of the well-known WHILE query language for static databases [6]

Syntax We assume w.l.o.g. that the database schema $\mathcal{R}$ includes (implicitly) two 0-ary schemas $First$ and $Last$ [6]. A *program* over $\mathcal{R}$ is specified by a sequence of *declarations* followed by a sequence of *instructions*.

• Declarations enable to specify some new relation schemas for programs, i.e. schemas not in $\mathcal{R}$, called *auxiliary schemas*. An auxiliary schema can be declared:

– **shared** or **private**,

– **checked** or **unchecked**.

Instances of auxiliary shared schemas are identical in all states of the temporal instance. Intuitively, such a relation is common to all states of the database. Instances of auxiliary private schemas can be different in the states of the temporal instance.

Declaring an auxiliary schema checked or unchecked has an impact on the semantics of the language. Intuitively, the former (resp. the latter) enforces the instance of the auxiliary schema to be used (resp. ignored) when computing the semantics of a **while** statement.

From now on, we suppose that $\mathcal{S}$ is the set of all auxiliary schemas.

• The possible instructions of a program are:

– temporal moves **left** and **right**,

– assignment $A(\vec{x}) := e(\vec{x})$, where $A \in \mathcal{S}$ and e is a FO formula over $\mathcal{R} \cup \mathcal{S}$ with free variables $\vec{x}$.

– iterator **while** $Cond$ **do** $InstrList$ **end**, where $Cond$ is a closed FO formula and $InstrList$ a sequence of instructions. Clearly, **while** iterators can be nested.

EXAMPLE 3.3 Consider the following programs $\mathcal{P}_1$ (over $\mathcal{R} = \emptyset$) and $\mathcal{P}_2$ (over $\mathcal{R} = \{G/2\}$) whose semantics are explained later on:

```
$\mathcal{P}_1$: **unchecked shared** $T/0$ ;
    **while** $\neg Last$ **do** { **right** ; $T() := \neg T()$ } **end**.
```

```
$\mathcal{P}_2$: **checked shared** $S/2$ ;
    $S(x,y) := G(x,y) \wedge x = a$ ;
    **while** $\neg Last$ **do**
        { **right** ; $S(x,y) := \exists z(S(x,z) \wedge G(z,y))$ } **end** ;
    **while** $\neg First$ **do**
        { **left** ; $S(x,y) := \exists z(G(x,z) \wedge S(z,y))$ } **end**.
```

[6]$First$ (resp. $Last$) is a boolean whose value is $true$ only in the first (resp. last) state of any temporal instance.

Semantics The evaluation of a program over $\mathbb{I}$ is obtained considering a *current time point*, which is simply one of the time points of the instance $\mathbb{I}$.

A *configuration* of a program $\mathcal{P}$ is composed of:

– the current time point,

– the instances over all **checked** auxiliary schemas of $\mathcal{P}$ in all states.

Note that when the evaluation starts, the current time point is the first one and the instances of auxiliary schemas are empty in all states [7].

• **left** (resp. **right**) decreases (resp. increases) the current time point by 1. If the current time point is the first (resp. last) one of the instance, then **left** (resp. **right**) has no effect.

• $A(\vec{x}) := e(\vec{x})$ changes the value of the instance over A. If A is **private** then A only changes in the current state (i.e. the state at current time point); if A is **shared** then A changes in all states of the temporal instance. When A "changes", its new value is set to the answer of the query $e(\vec{x})$ evaluated at the current time point.

• **while** $Cond$ **do** $InstrList$ **end** executes all instructions of $InstrList$ until either $Cond$ becomes false, or until the sequence of configurations observed after each execution of $InstrList$ reaches a repetition. When both the current time point and the checked relations repeat, the computation terminates. Recall that unchecked instances of auxiliary schemas do not belong to configurations, and thus are not taken into account when checking for a repetition.

If none of the conditions above is ever fulfilled (i.e. $Cond$ never becomes false and the sequence of configurations never reaches a repetition), we say that the semantics of both the **while** statement and the program it belongs to are undefined.

EXAMPLE 3.4 We consider the programs of example 3.3. For $\mathcal{P}_1$, the auxiliary schema T is boolean (i.e. of arity 0). When evaluated, the program executes $T() := \neg T()$ in all states of the temporal instance, from left to right, starting from the first state. Thus the instance over T, whose initial value is $false$, returns the parity of the size of the temporal instance. Note that there is no checked auxiliary schema in this program.

For $\mathcal{P}_2$, it can be shown that when the computation is finished the instance over S is the same as the evaluation of formula ψ from example 3.2.

[7]Thus, the initial value of boolean auxiliary schemas is $false$ in all states. In this paper, we take advantage of this fact by never initializing booleans with $false$.

Query languages We define queries based on the programs defined above exactly like for the WHILE language, i.e. by giving both a program $\mathcal{P}$ and an auxiliary schema A. The answer of a query on $\mathbb{I}$ is the instance over A at time point 1 output by the evaluation of $\mathcal{P}$ over $\mathbb{I}$. Note that A can be either **checked** or **unchecked** in $\mathcal{P}$.

Next, we use the notion of inflationary assignment. An assignment $A(\vec{x}) := e(\vec{x})$ is *inflationary* iff it *adds* all tuples which satisfy e to A. An inflationary assignment $A(\vec{x}) := e(\vec{x})$ is intuitively a "cumulative" one and is equivalent to $A(\vec{x}) := e(\vec{x}) \vee A(\vec{x})$.

We now define four query languages based on the previous syntax and semantics.

1. T-WHILE$^+$ denotes the language without unchecked auxiliary schema and where all assignments are inflationary.
2. T-WHILE denotes the language without unchecked auxiliary schema and with arbitrary assignments.
3. T-WHILE$_u^+$ denotes the language where auxiliary schemas can be unchecked or checked, and where all assignments whose left-hand-side is a checked schema are inflationary.
4. T-WHILE$_u$ denotes the language where auxiliary schemas can be unchecked or checked, and with arbitrary assignments.

Note that one can specify programs of T-WHILE$_u^+$ and T-WHILE$_u$ with no checked auxiliary schemas. Indeed one may write T-WHILE-like programs with no auxiliary schema at all. This is the case of the following program which ends up setting the current time point to the last time point n of the temporal instance: **while** $true$ **do right end**.

About language names For the sake of our investigation and presentation, we choose different names for some of the languages introduced in [1]. The names given by us to the languages identify the features which are the key of the expressive power results provided in this paper. Thus:

– the language T-FIXPOINT of [1] is T-WHILE$_u^+$ here,

– the language T-WHILE of [1] is the same here,

– the language called T-WHILE$^+$ here was briefly referred to as the "purely inflationary restriction" of T-WHILE in [1].

Repetition vs. limit The semantics of the **while** statement is defined using the repetition of the sequence of configurations. Another classical way to define such a semantics is by using the *limit* of the sequence. These two definitions lead to different semantics for the languages T-WHILE$_u^+$ and T-WHILE$_u$, and they lead to equivalent query languages for T-WHILE$^+$ and T-WHILE. Indeed, T-WHILE$_u^+$+repetition and T-WHILE$_u^+$+limit (resp. T-WHILE$_u$+repetition and T-WHILE$_u$+limit) are probably not equivalent query languages. Hence, we beleive that only T-WHILE$^+$ and T-WHILE would deserve to be called « fixpoint query languages ».

It is easy to show that in the static case, the repetition and limit definitions lead to equivalent query languages for both WHILE and WHILE$^+$.

Partial query languages There exists programs, belonging to T-WHILE$^+$, T-WHILE, T-WHILE$_u^+$ and T-WHILE$_u$ whose semantics is undefined. Below we provide a program belonging to all four languages whose semantics is undefined when the size of the temporal instance is greater than 3.

```
right ;
while $true$ do
   left ;
   while $First$ do { right ; right } end ;
end.
```

As opposed to μTL-like languages, the fact that all T-WHILE-like languages are partial query languages (even T-WHILE$^+$) stands in contrast with the static case: recall that WHILE$^+$ is a total query language.

4 Expressive power results

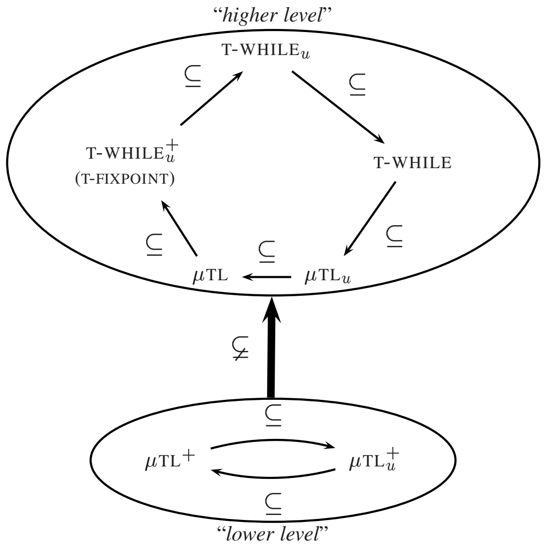
This section begins with an introductory discussion. Note that all μTL-like and T-WHILE-like languages discussed in this section are strictly more expressive than TL.

T-WHILE$_u^+$ vs. T-WHILE In [1], the authors show that the equivalence T-WHILE$_u^+ \equiv$ T-WHILE is "improbable". Their proof proceeds to suppose that T-WHILE$_u^+ \equiv$ T-WHILE. Then in particular, T-WHILE$_u^+$ is equivalent to T-WHILE on temporal databases consisting of a single state, and hence WHILE$^+ \equiv$ WHILE (because it is shown in [1] that "unchecked" schemas do not increase the expressive power of WHILE$^+$). However WHILE$^+ \equiv$ WHILE is proved equivalent to PTIME $=$ PSPACE in [3], which is unlikely.

The fact that in this paper we show T-WHILE$_u^+ \equiv$ T-WHILE should not be interpreted as a contradiction. Indeed our proof considers temporal instances containing *at least two states*, whereas as pointed out above, the result in [1] is obtained by restricting temporal instances to static ones. These two results (the unlikelyness of T-WHILE$_u^+ \equiv$ T-WHILE for single state temporal instances and the proof of T-WHILE$_u^+ \equiv$ T-WHILE for temporal instances of size at least 2) highlight an essential difference between static and temporal querying.

Data complexity of T-WHILE$_u^+$ As mentioned above, we show in this paper that T-WHILE$_u^+ \equiv$ T-WHILE over really temporal instances. This entails that T-WHILE$_u^+$ is PSPACE which disproves the claim of [1] that T-FIXPOINT is PTIME. Intuitively, this is due to the non inflationary behavior of the current time point in T-WHILE-like languages.

New expressive power results The expressive power results that are now presented lead to a two-level hierarchy, as shows the figure below. Inside each level, we display equivalent query languages (the figure indicates the way the equivalences are proved by displaying the inclusions we establish). In order to obtain the hierarchy, we first focus on the lower level, then we show the separation between the two levels, and finally turn to the higher one. For the sake of space, the proofs are sketched and sometimes even not presented.



Lower level of the hierarchy Obviously from the definitions, $\mu\text{TL}^+ \subseteq \mu\text{TL}_u^+$. Moreover:

LEMMA 4.1 $\boxed{\mu\text{TL}_u^+ \subseteq \mu\text{TL}^+}$

This result shows that unchecked auxiliary relations, though not inflationary, do not increase the expressive power of the inflationary language μTL^+. The proof is done by means of a *translation* of μTL_u^+ μ-expressions into equivalent μTL^+ formulas. The key argument is that, due to the inflationary nature of the computation, formulas belonging to μ-expressions of μTL_u^+ are only evaluated as long as tuples are inserted in the **checked** auxiliary relations. Hence, the contents of the **unchecked** relations can be simulated by *versioning* their tuples with the tuples inserted in the checked relations since the previous iteration. The versioning is done using cartesian product. The "old" versions are cumulated in a separate relation, so that the process is fully inflationary, as needed for the query language μTL^+. This technique is sketched in [1] to show that unchecked relations do not increase the expressive power of the static query language WHILE$^+$.

THEOREM 4.2 $\boxed{\mu\text{TL}^+ \equiv \mu\text{TL}_u^+}$

Relation between the two levels of the hierarchy

THEOREM 4.3 $\boxed{\mu\text{TL}^+ \subsetneq \mu\text{TL}}$

To prove this result, we consider propositional temporal instances (i.e. instances having only relations of arity 0). In this context, we show that $\mu\text{TL}^+ \in$ LINTIME and μTL expresses the NP-complete problem SAT.

The complexity measure we use here is the size n of the temporal instance [8]. As a matter of fact, we have $\mu\text{TL} \in$ LINSPACE: the only space needed for the computation of μ-expressions is the one to store the auxiliary relations [9]. We then have $\mu\text{TL}^+ \in$ LINTIME, because μTL^+ is an inflationary language: during the "evaluation" of a μ-expression, each auxiliary relation in each state of the temporal instance, can either stay empty or become full and stay full thereafter. Finally, it is rather straightforward to encode the well-known problem SAT (satisfiability of a propositional formula) in μTL, and SAT is NP-complete. The encoding works by considering a formula in conjonctive normal form ; it encodes each clause of the formula in a state of the temporal instance, and computes the potential models of the formula by using as many auxiliary schemas as there are variables in the formula.

Higher level of the hierarchy

LEMMA 4.4 $\boxed{\mu\text{TL}_u \subseteq \mu\text{TL}}$

This result shows that unchecked auxiliary relations do not increase the expressive power of μTL. The proof structure is roughly the same as the one for $\mu\text{TL}_u^+ \subseteq \mu\text{TL}^+$, though more simple. Indeed, here we do not have to deal with "inflationary vs. not inflationary" formulas, hence the versioning technique is unnecessary.

LEMMA 4.5 $\boxed{\mu\text{TL} \subseteq \text{T-WHILE}_u^+}$

This result is shown only over temporal instances whose size is at least 2. The proof is done by means of a *translation* of μTL formulas into T-WHILE$_u^+$ programs. This translation makes use of the fact that the size of the temporal instances is at least 2 as highlighted below.

We first introduce some macros for T-WHILE$_u^+$. They use special auxiliary schemas called macro schemas: 1) all macro schemas are boolean and **unchecked**, 2) when writing a program, it is assumed that each time a macro is used new names are given to its macro schemas. Let $Cond$ be a closed FO formula.

• A conditional assignment **if** $Cond$ **then** $A(\vec{x}) := e(\vec{x})$ is defined by

$$A(\vec{x}) := (Cond \wedge e(\vec{x})) \vee (\neg Cond \wedge A(\vec{x}))$$

• A conditional temporal move **if** $Cond$ **then** $depl$ (where $depl \in \{\textbf{left}, \textbf{right}\}$) is defined by:

[8]Note that the complexity measure for query languages is usually the number of tuples rather than the number of states of the instance.

[9]In the propositional context, each relation is either empty (i.e $false$) or "full" (i.e. $true$, when its content is the empty tuple).

```
$Aux() := true$ ;
while $Cond \wedge Aux()$ do { $depl$ ; $Aux() := false$ } end
```

where Aux is a **shared** macro schema.

• The macro **goto**$(Bool)$ with $Bool()$ a boolean schema is defined by:

```
while $true$ do left end ;
while $\neg Bool()$ do right end ;
```

Intuitively, **goto**$(Bool)$ sets the current time point to the leftmost time point where $Bool()$ is true.

• The macro **Flipflop** is defined by:

```
if $First$ then $Aux() := true$ ;
if $First$ then right ; if $\neg Aux()$ then left ;
$Aux() := false$ ;
```

where Aux is a **shared** macro schema. Intuitively, **Flipflop** changes the current time point as long as the temporal instance contains at least two states.

• The macro **scan**$(Cond, InstrList)$, where $InstrList$ is a sequence of instructions is defined by:

```
$Mark() := true$ ;
while $true$ do left end ;
while $Cond$ do { $InstrList$ ; right } end ;
goto$(Mark)$ ;
$Mark() := false$ ;
```

where $Mark$ is a **private** macro schema. Intuitively **scan**$(Cond, InstrList)$ executes $InstrList$ in all states of the temporal instance from left to right starting from the first state, as long as $Cond$ is true. The initial instruction of **scan** marks the current state (using $Mark$) in order to be able "go back there" using **goto**. Hence, **scan** preserves the current time point.

• A global assignment $A(\vec{x}) \,\widehat{:=}\, e(\vec{x})$ is defined by:

```
scan$(true, A(\vec{x}) := e(\vec{x}))$ ;
```

Intuitively, a global assignement enforces A to behave like a **shared** auxiliary schema for the assignment $A(\vec{x}) := e(\vec{x})$.

• The macro **Repetition**(X, Y) is defined by:

```
$BRep() := true$ ;
scan$(BRep, BRep() := (\forall \vec{x}(X(\vec{x}) \longleftrightarrow Y(\vec{x}))))$ ;
```

where $BRep$ is a **shared** macro schema. Intuitively, **Repetition**(X, Y) checks that the instances over X and Y are the same in all states.

Sketch of proof: We now proceed to the translation. Let φ be a μTL formula ; we inductively give a program $\mathcal{P}_\varphi$ and an auxiliary schema A_φ such that for all instance $\mathbb{I} = I_1, ..., I_n$ and for all $i \in [1, n]$, the evaluation over $\mathbb{I}$ at time point i of φ is equal to the instance over A_φ at time point i produced by the evaluation of $\mathcal{P}_\varphi$. In order to build $\mathcal{P}_\varphi$, we inductively build an intermediate program $\mathcal{D}^{int}_\varphi$; $\mathcal{I}^{int}_\varphi$ (where $\mathcal{D}^{int}_\varphi$ are declarations and $\mathcal{I}^{int}_\varphi$ are instructions) and a FO formula F_φ.

The program $\mathcal{P}_\varphi$ is given by:

```
$\mathcal{D}^{int}_\varphi$ ; checked private $A_\varphi/|\varphi|$ ;
$\mathcal{I}^{int}_\varphi$ ; $A_\varphi(\vec{x}) \,\widehat{:=}\, F_\varphi(\vec{x})$.
```

where $|\varphi|$ denotes the number of free variables of a φ.

• If φ is obtained through an FO formation rule, then F_φ is essentially φ itself, and $\mathcal{D}^{int}_\varphi$ and $\mathcal{I}^{int}_\varphi$ are straightforward.

• If $\varphi(\vec{x}) = next(\psi(\vec{x}))$, let:

```
– $\mathcal{D}^{int}_\varphi = \mathcal{D}^{int}_\psi$ ; unchecked shared $Temp/|\psi|$ ;
            unchecked private $N/|\psi|$ ;
– $F_\varphi(\vec{x}) = N(\vec{x})$
– $\mathcal{I}^{int}_\varphi$ =
    $\mathcal{I}^{int}_\psi$ ;
    $N(\vec{x}) \,\widehat{:=}\, false$ ;                /* resets $N$ in case of nesting */
    scan$(\neg Last, InstrList_{next})$ ;         /* sets $N$ */
```

where $InstrList_{next}$ is:

```
{ right ; $Temp(\vec{x}) := F_\psi(\vec{x})$ ; left ; $N(\vec{x}) := Temp(\vec{x})$ }
```

Intuitively, it is "not possible" in T-WHILE^+_u to grab, at the current time point, anything from the next state or any other state. Thus in order to translate $next(\psi)$ we use $Temp$ whose purpose is to move $F_\psi(\vec{x})$ from the next state to the current state. Hence $Temp$ needs to be shared. It also needs to be unchecked because we compute $next(\psi)$ in all states and this computation is not inflationary.

• the case $\varphi(\vec{x}) = prev(\psi(\vec{x}))$ is similar to the previous one and is omitted here.

• If $\varphi(\vec{x}) = \mu fpt[\psi_S][](S)(\vec{x})$ [10], where S is an auxiliary schema of ψ_S, let:

```
– $\mathcal{D}^{int}_\varphi = \mathcal{D}^{int}_\psi$ ;
  unchecked shared $BRep_S/0, Stop/0$ ;
  unchecked private $S/|\psi_S|, S^{old}/|\psi_S|, Mark_\varphi/0$ ;
– $F_\varphi(\vec{x}) = S(\vec{x})$
– $\mathcal{I}^{int}_\varphi$ =
    $Mark_\varphi() := true$ ;                 /* marks the current state */
    $S(\vec{x}) \,\widehat{:=}\, false$ ;               /* resets $S$ to $\emptyset$ in case of nesting */
    while $\neg Stop()$ do
      Flipflop ;                       /* enforces current time point to change */
      $S^{old}(\vec{x}) \,\widehat{:=}\, S(\vec{x})$ ;              /* sets $S^{old}$ to $S$ in all states */
      $\mathcal{I}^{int}_{\psi_S}$ ;
      $S(\vec{x}) \,\widehat{:=}\, F_\psi(\vec{x})$ ;
      Repetition$(S, S^{old})$ ;        /* sets $BRep_S$ to $true$ iff $S = S^{old}$ */
      if $BRep_S()$ then $Stop() := true$ ;
    end ;
    goto$(Mark_\varphi)$ ;
    $Mark_\varphi() := false$ ; $Stop() := false$ ;
```

Intuitively, since ψ_S is not necessarily an inflationary formula, it is translated with the **unchecked** auxiliary schema S. This implies that the evaluation of the **while** block proceeds with configurations whose single component is the current time point. Recall that the evaluation of **while** loops stops either because their condition is false or because of configuration repetition. Here our goal is to force the evaluation of the **while** block to stop when "S reaches a repetition". Because S is unchecked, repetition check over S is encoded in the **while** block. Morerover **Flipflop** enforces

[10]Here we consider w.l.o.g. μTL restricted to single-formula μ-expressions.

two consecutive configurations to differ by changing the current time point and thus the evaluation of **while** block only stops when its condition becomes false. Note that **Flipflop** works properly only if the temporal instance size is at least 2. Of course, if the semantics of the μ-expression $\mu fpt[\psi_S][\,](S)$ is undefined, so is the semantics of the above program. □

Obviously T-WHILE$_u^+ \subseteq$ T-WHILE$_u$. Moreover:

LEMMA 4.6 $\boxed{\text{T-WHILE}_u \subseteq \text{T-WHILE}}$

This result shows that unchecked relations do not increase the expressive power of T-WHILE.

Sketch of proof: Let $\mathcal{W}$ be a **while** statement of a T-WHILE$_u$ program $\mathcal{P}$, such that $\mathcal{W}$ contains two auxiliary schemas S and T, where S (resp T) is declared **checked** (resp. **unchecked**). We suppose that both S and T appear as left-hand-sides of assignments within $\mathcal{W}$. Now let $\mathcal{P}'$ be obtained from the program $\mathcal{P}$ by declaring T **checked**. The evaluation of $\mathcal{W}$ in $\mathcal{P}'$ is necessarily at least as long as its evaluation in $\mathcal{P}$: if S, T and the current time point repeat, then in particular S and current time point repeat. But the evaluation of $\mathcal{W}$ in $\mathcal{P}'$ can be longer, and may not terminate (e.g. if T never reaches a repetition). Hence a way to have equivalent evaluations of $\mathcal{W}$ in both $\mathcal{P}$ and $\mathcal{P}'$ is to observe when both S and the current time point reach a repetition, and enforce that the condition of $\mathcal{W}$ becomes $false$ at that moment. □

LEMMA 4.7 $\boxed{\text{T-WHILE} \subseteq \mu\text{TL}_u}$

Sketch of proof: This proof is decomposed into two parts: 1) partial unnesting of **while** statements 2) translation of partially unnested T-WHILE programs into μTL$_u$ formulas. Both parts are rather technical and intricate, hence for the sake of space we only sketch them below.

We need to unnest **while** statements because of a fundamental difference between T-WHILE and μTL$_u$: in the former, an instance of an auxiliary schema S can be assigned by more than one instruction, and hence its contents can change at various nesting levels. This is "not possible" in μTL$_u$ because an instance of S is only defined once by a single formula φ_S. Partial unnesting of T-WHILE programs addresses this issue.

Now, let us try to outline the main ideas behind partial unnesting. Recall that WHILE programs can be fully unnested. Indeed, consider the following WHILE program:

```
while $Cond_1$ do
    $S_1(\vec{x}) := e_1(\vec{x})$ ;
    while $Cond_2$ do $S_2(\vec{y}) := e_2(\vec{y})$ end ;
end.
```

This program is equivalent to the following unnested one:

```
$S_2^{old}(\vec{y}) := S_2(\vec{y})$ ;
while $Cond_1$ do
    if $(\forall \vec{y}(S_2^{old}(\vec{y}) \longleftrightarrow S_2(\vec{y})))$ then $S_1(\vec{x}) := e_1(\vec{x})$ ;
    $S_2^{old}(\vec{y}) := S_2(\vec{y})$ ;
    if $Cond_2$ then $S_2(\vec{y}) := e_2(\vec{y})$ ;
end.
```

For WHILE programs, the main idea behind unnesting is to unfold nested **while** statements by means of "conditional" assignments which are assignments themselves, thus the process can be iterated until obtaining fully unnested programs. The above technique can be applied to unnest T-WHILE programs: assignments and temporal moves in nested **while** statements are transformed into conditional assignments and conditional temporal moves. Contrary to the case of WHILE, the process does not lead to fully unnested T-WHILE programs for the following reason: a conditional temporal move is a (simple) **while** statement [11]. Furthermore, to unnest T-WHILE programs, we need to propagate the storage instruction $S_2^{old}(\vec{y}) := S_2(\vec{y})$ on all states. This leads to a global assignment, which is also a **while** statement. To sum up, the partial unnesting of a T-WHILE program leads to a program whose **while** statements contain only conditional temporal moves, conditional assignments, conditional global assignments, and the conditional variants of the macros **Repetition** and **Time**.

Once a T-WHILE program is partially unnested, translating its conditional instructions listed above turns out to be technical but rather simple. The main problem raised by the translation of a T-WHILE program is to deal with time: a major component of the configuration used in defining the semantics of T-WHILE-like languages is the current time point which introduces some kind of explicit control of time in T-WHILE; no such notion exists for μTL-like languages. The translation manages this difficulty as follows: to each instruction we associate a formula aiming at marking its current time point. Such a formula has to be declared checked in order to take time into account the way T-WHILE does. □

THEOREM 4.8 $\boxed{\text{The languages } \mu\text{TL}, \mu\text{TL}_u, \text{T-WHILE}, \text{T-WHILE}_u, \text{T-WHILE}_u^+ \text{ are equivalent.}}$

We would like to emphasize once again that this theorem is valid under the two-states instances assumption only. The result does not hold for single state instances (see section 4).

5 Discussion

The first contribution of this paper is to enrich the expressive power hierarchy of [1] with results for μTL-like languages. In particular, our two-level hierarchy provides the proof of the claim of [11] that the first order extension

[11] We do not succeed in unnesting conditional temporal moves.

of μTL (denoted μTL^+ in our framework) is strictly less expressive than T-FIXPOINT (denoted T-WHILE_u^+ in this paper). Because T-FIXPOINT is equivalent to T-WHILE over really temporal instances, T-FIXPOINT is PSPACE.

The second important contribution of this paper is to highlight the impact of unchecked auxiliary schemas on T-WHILE-like and μTL-like languages. Allowing one to use unchecked auxiliary schemas has no impact on the expressive power of non inflationary languages of both classes and on the expressive power of the inflationary language μTL^+. In the static case, the inflationary (resp. non inflationary) WHILE language is equivalent to the inflationary (resp. non inflationary) FO+fixpoint language. The paper shows that this symmetry is broken in the temporal framework: $\mu\text{TL}_u^+ \subsetneq \text{T-WHILE}_u^+$.

We do not provide any insight w.r.t. the expressive power of T-WHILE^+. However, we have strong reasons (although not yet proofs) to beleive that:

CONJECTURE 5.1 $\boxed{\text{T-WHILE}^+ \subsetneq \text{T-WHILE}}$

As in [1] (see section 4), we can use a complexity argument to prove that in case of static instances $\text{T-WHILE}^+ \subsetneq$ T-WHILE probably holds because in that case T-WHILE^+ reduces to WHILE^+, T-WHILE reduces to WHILE and $\text{WHILE}^+ \equiv$ WHILE iff PTIME=PSPACE. This argument relying on static instances does not completely satisfy us because it does not entail that the result holds for really temporal instances (as our result about T-WHILE_u^+ and T-WHILE shows). Another direction of investigation is to show that T-WHILE^+ and T-WHILE are separated by some query. A candidate is the query denoted **twin** which checks wether there exists two identical states in the temporal instance.

To conclude this discussion, let us comment on the specific treatment of time in T-WHILE-like languages. Somehow, time is not fully implicit in these languages because of the definition of configuration. Moreover, the temporal moves **left** and **right** alone are too “poor” to simulate the $next$ and $prev$ modalities of μTL-like languages. Roughly, (see for instance the translation of $next$ in the proof of lemma 4.5), during the evaluation of a T-WHILE-like program, the computations made at some time point are only aware of the state at that time point and of the shared auxiliary relations. Thus, without shared auxiliary schemas, moving to the right does not help to compute queries dependending on data of both the current and next states because the move allows one to access the next state data and meanwhile prevents from accessing data in the current state. It is then clear that such queries need to be supported by shared and non inflationary auxiliary schemas.

References

[1] S. Abiteboul, L. Herr and J. Van den Bussche. Temporal Connectives versus Explicit Timestamps in Temporal Query Languages. In *Journal of Computer and System Science*, 58(1):54–68, 1999.

[2] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. In *Journal of Computer and System Science*, 43:62–124, 1991.

[3] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. ACM SIGACT Symp. on the Theory Of Computing*, pages 209–219, 1991.

[4] N. Bidoit and S. De Amo. Implicit temporal query languages: towards completeness, In *FST&TCS*, Chennai, India, LNCS Vol. 1738, 1999, pages 245–257.

[5] N. Bidoit, S. De Amo, and L. Segoufin. Order independent temporal properties. In *Journal of Logic and Computation*, 14(2):277–298, 2004.

[6] A. K. Chandra and D. Harel. Structure and comlexity of relational queries. In *Journal of Computer and System Science*, 25(1):99–128, 1982.

[7] J. Chomicki and D. Toman. Temporal Logic in Information Systems. In *Logics for databases and information systems*, Kluwer Academic Publishers, chapter 3, pages 31–70, 1998.

[8] E. A. Emerson. Temporal and Modal Logic, In *Handbook of Theoretical Computer Science*, Volume B: Formal Models and Semantics, Jan van Leeuwen, Ed., Elsevier Science Publishers (1990) 995–1072.

[9] Y. Gurevich. Toward a logic tailored for computational complexity. In *Computation and Proof Theory*, pages 175–216, M. M. Ritcher et al. editor, Springer Verlag, LNM 1104, 1984.

[10] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. In *Annals of Pure and Applied Logic*, 32:265–280,1986.

[11] L. Herr. Langages de Requête pour les Bases de Données Temporelles. *Ph.D thesis, Université Paris Sud*, 1997.

[12] D. Leivant, Inductive definitions over finite structures. In *Information and Computation*, 89:95–108, 1990.

[13] A.R. Meyer. Weak monadic second order theory of successor is not elementary recursive. In *Proceedings Logic Colloquium*, Lecture Notes in Mathematics, Vol. 453, pp. 132–154, Springer-Verlag, 1975.

[14] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*, North Holland, Amsterdam, 1974.

[15] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. In *Journal of the ACM*, 32(3):733–749, 1985.

[16] D. Toman. On Incompleteness of Multi-dimensional First-order Temporal Logics. In *TIME*: 99–106, 2003.

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