Relational Temporal Machines

Nicole Bidoit
Univ. Paris-Sud, UMR 8623, Orsay F-91405
CNRS, Orsay F-91405
nicole.bidoit@lri.fr

François Hantry
Rectorat de Paris
francois.hantry@voila.fr

Abstract

The paper introduces and investigates `relational temporal machine` *(*RTM*) as a general abstract model for generic temporal querying. The* RTM *devices subsume most temporal query languages that have emerged in the literature. A first contribution of the paper is to provide two simplified forms for our machines, namely extended one-tape* RTM*s and one-tape* RTM*s. Another contribution is to establish connections between* RTM*s and the* T-WHILE *and* TS-WHILE *extensions of* FO *based on complexity criteria.*

1 Introduction

Emerging applications such as the web, bioinformatics, medical applications, multimedia applications, animation, simulation, personal information management, data flow processing, streaming, are intensively managing temporal or ordered information. Thus in many computer science areas, modeling and manipulating temporal and ordered information turns out to be a central issue.

Suitable database models [12, 21] have been developed to store and support temporal information management. The languages mainly target snapshot queries (what are the data available at some time point?), history extractions (what is the evolution of some data?).

Most temporal query languages are based on extensions of first-order logic (FO) or SQL. Two central classes of formal temporal languages have been considered so far.

- Linear temporal logic TL [13, 12] and its extensions ETL [24], μTL [23], T-WHILE and T-FIXPOINT [3, 8] allow to query temporal databases represented as finite sequences of database states.
- Time-stamped first-order logic TS-FO and its extension TS-WHILE [3] allows to query temporal databases represented by time-stamped relations.

The aim of this paper is to introduce and investigate `relational temporal machines` (RTM) as a general-purpose computation model for temporal querying. The first issue is to provide an abstract model of temporal querying. Indeed, the well-known formal temporal languages TL and TS-FO have very strong limitations: they provide the ability to define mappings from temporal instances to relations (tables) and lack the ability to map temporal instances to temporal instances, in a general manner. The T-WHILE extension of FO shares the same limitation although the language allows for computing auxiliary temporal instances in a restricted manner: auxiliary temporal instances in T-WHILE programs need to have a number of states equal to that of the input temporal instance. Indeed, RTMs have the ability to map temporal instances to temporal instances and thus these devices subsume most temporal query languages. The temporal computations defined by RTM are generic in the classical sense, that is, data with identical logical properties are treated uniformly. The second issue is to demonstrate that RTMs are of special interest for studying, understanding and extending some of the features of known temporal languages by mapping such languages to subclasses of RTMs. Indeed, one contribution of our paper is to establish connections between RTMs and the T-WHILE and TS-WHILE extensions of FO.

RTMs are based on Turing machines and have been inspired by `relational machines` (RM) [5] known as a very powerful formal tool for studying relational queries and their complexity. The RTM model is a rather simple and intuitive extension of the RM model, given that we assume temporal instances to be finite sequences of database states. An RTM machine uses tapes to store the input temporal instance (input tape) and the auxiliary temporal instances (auxiliary tapes). Roughly, the contents of an RTM cell is a (set of) relation(s) of fixed arity rather than a bit in a Turing machine cell. The interactions or transitions of an RTM can modify the auxiliary tapes either by changing the contents of (relations stored in) a cell through a first order (FO) query, or by adding/removing cells. The input is accepted if the machine halts in an accepting state. The output is the temporal instance stored in one of its auxiliary tapes. Thus, a RTM is able to output a temporal instance of any (finite)

14th International Symposium on Temporal Representation and Reasoning (TIME'07
0-7695-2836-8/07 $20.00

size, and of course of a size greater than that of the input temporal instance. Note that, like a RM, an RTM uses a fixed set of FO queries.

The first results presented in the paper are dedicated to analysing the usefulness of different components of our devices. Focussing on the number of auxiliary tapes, we are able to provide two reductions for RTMs. The first reduction is rather simple and expected: it says that any RTM can be reduced to one, called extended one-tape RTM, using exactly two auxiliary tapes of which one is a simple store. A store is a one-cell tape. Although more powerful, extended one-tape RTMs features are closely related to T-WHILE features.

The second result is a stronger contribution and provides a normal form for RTMs. We prove that any RTM can be reduced to one using one and only one auxiliary tape. Indeed, the proof strongly relies on the technical tools developed in [5] for building RM normal forms.

We then investigate some complexity classes of temporal computation by RTMs. Our complexity measures are with respect to the length of the input temporal instance (tape). We focus on space complexity: we intend to measure the size (length) of the auxiliary memory needed to compute some temporal query. The paper first investigates linear-space and polynomial-space RTM. Although T-WHILE and TS-WHILE are standard temporal query languages (they output static tables), they make use of temporal auxiliary instances during their computation. Thus it makes sense to compare T-WHILE and TS-WHILE with RTM. Nicely enough, it turns out that T-WHILE (resp. TS-WHILE) is complete (i.e. expresses all queries) w.r.t. the class of linear-space (resp. polynomial-space) RTMs, when outputs are restricted to be relations.

Constant-space RTMs are devices using one auxiliary store. Recall that a store is a one-cell tape. Such devices are called `relational register machine` (RRM), because registers are needed in that case. Investigating RRM leads us to study new languages based on T-WHILE or on TL. Indeed, we first investigate queries that are computable by `relational pebble machine` (RPM), which are RRM conforming to a stack discipline for register management. It is immediate to show that, in the propositional case, the languages accepted by RPMs are exactly regular languages. This entails that RRM are strictly more powerful than RPM. We then show that RPM are strictly more expressive than TS-FO and ETL and claim the same for a natural restriction of T-WHILE called T-WHILE$_{\text{shared}}$.

Our main contribution concerning RRM provides two temporal query languages, which are complete w.r.t. RRMs. The first language is a while-loop language called T-WHILE$^{\text{bind}}_{\text{shared}}$ based on T-WHILE. The second language, called @TL, is based on TL and hybrid modal logic [10] and uses temporal variables as propositions plus a fixpoint operator.

The paper is organized as follows. After the preliminary definitions and notations presented in section 2, section 3 introduces `relational temporal machines` and then presents the two main reductions. Section 4 investigates linear-space, polynomial-space and constant-space sub-classes of RTMs as well as their relationship with known languages. The last section provides a short concluding discussion and further research directions.

2 Preliminaries

We review informally some notions and notations related to relational databases (db) and query languages. We assume the reader familiar with both first-order logic FO and with the usual definitions of *relation schema*, *database schema* and *instances*. In the whole paper, we assume a unique possibly infinite domain for all attributes. In the context of query specification, the db schema $\mathcal{R}$ (resp. $\mathcal{S}$) is a set of relation schemas and is used, in general, as the input db schema (resp. auxiliary db schema).

An implicit temporal instance $\mathcal{I}$ over the db schema $\mathcal{R}$ is a finite sequence $I_1, ..., I_n$ of finite instances over $\mathcal{R}$. The size of $\mathcal{I}$, denoted $|\mathcal{I}|$ is the size of the sequence $I_1, ..., I_n$, that is n. For each $i \in [1..n]$, I_i is called the state of $\mathcal{I}$ *at time point* i. The instance $I_i(R)$ of the relation schema R at time point i is also denoted by $\mathcal{I}[i](R)$. The active domain of $\mathcal{I}$, denoted $adom(\mathcal{I})$, is set of domain elements appearing in $\mathcal{I}$. Thus $adom(\mathcal{I})$ is a finite subset of the domain. In the paper, $\vec{x}$ represents a tuple of (data) variables whose arity is clear from the context, and ν is a valuation of $\vec{x}$ ranging over the active domain. We assume w.l.o.g. that the db schema $\mathcal{R}$ includes two 0-ary relations (propositions) $First$ and $Last$ such that $\mathcal{I}[i](First)=true$ iff $i=1$, and $\mathcal{I}[i](Last)=true$ iff $i=n$. We denote $\emptyset^n_{\mathcal{R}}$ the temporal instance over $\mathcal{R}$ of size n whose states are all empty. Thus, $\emptyset^0_{\mathcal{R}}$ denotes the temporal instance of size 0.

The temporal query language TL The first order linear temporal logic TL [13] is a well known formalism for specifying queries over implicit temporal databases [12, 3, 7]. The syntax of TL over a db schema $\mathcal{R}$ is given by the formation rules for FO over $\mathcal{R}$, together with the following additional rules: if φ_1 and φ_2 are formulas then φ_1 *until* φ_2 and φ_1 *since* φ_2 are formulas.
The *satisfaction* of a TL formula φ over a temporal instance $\mathcal{I}$ at time point $i \in [1..n]$, given a valuation ν of the free variables of φ, denoted $[\mathcal{I}, i, \nu] \models \varphi$, is defined as follows:

- If φ is $R(\vec{x})$, $[\mathcal{I}, i, \nu] \models \varphi$ iff $\nu(\vec{x}) \in \mathcal{I}(R)[i]$.
- If φ is obtained by a first order rule, $[\mathcal{I}, i, \nu] \models \varphi$ is defined as usual.
- $[\mathcal{I}, i, \nu] \models \varphi_1$ *until* φ_2 iff there exists $j > i$ such that $[\mathcal{I}, j, \nu] \models \varphi_2$ and for all k such that $i < k < j$, $[\mathcal{I}, k, \nu] \models \varphi_1$.

• $[\mathcal{I},i,\nu]\models\varphi_1$ *since* φ_2 iff there exists $j<i$ such that $[\mathcal{I},j,\nu]\models\varphi_2$ and for all k such that $i>k>j$, $[\mathcal{I},k,\nu]\models\varphi_1$.

A TL query q over the db schema $\mathcal{R}$ is specified by a TL formula $\varphi(\vec{x})$ and the answer of q over the temporal instance $\mathcal{I}$, denoted $q(\mathcal{I})$, is obtained by evaluating φ at time point 1: $q(\mathcal{I})=\{\nu(\vec{x}) \mid [\mathcal{I},1,\nu]\models\varphi(\vec{x}),\ \nu \text{ a valuation}\}$.

The temporal query language ETL ETL [24, 3] is an extension of TL build using FO and regular languages. The following formation rule is added to FO. Let L be a regular language over the finite alphabet $a_1 \dots a_p$ and let $\varphi_1 \dots \varphi_p$ be ETL formulas. Then $L^+(\varphi_1 \dots \varphi_p)$ and $L^-(\varphi_1 \dots \varphi_p)$ are also ETL formulas.

• $[\mathcal{I},i,\nu]\models L^+(\varphi_1 \dots \varphi_p)$ iff there exists a word $w=v_{w_i}\dots v_{w_n}$ in L of length $n-i+1$ such that $[\mathcal{I},k,\nu]\models\varphi_{w_k}$ for $k=i \dots n$.

The semantics of $L^-(\varphi_1 \dots \varphi_p)$ is dual and, intuitively, checks a matching word w over the past (left to i) instead of over the future (rigth to i).

For example, in order to specify in ETL the TL formula $R(x)untilS(x)$, one needs to consider the language $L=a^*b$ and the formula $L^+(\varphi_a,\varphi_b)$ with $\varphi_a=R(x)$ and $\varphi_b=S(x)$.

A ETL query is defined exactly as a TL query and evaluated at time point 1.

Another example illustrates the power of ETL. One can express in ETL the query `t-even` which returns true over $\mathcal{I}$ if $|\mathcal{I}|$ is even and false otherwise. It suffices to consider the language $L=(aa)^*$ and the formula $L^+(\varphi_a)$ with $\varphi_a=true$.

The temporal query language T-WHILE [3] is an extension of the well-known WHILE query language for static databases [11]. It is built using imperative mechanisms like assignments, loops and temporal moves. Variants of this language have been investigated in [8].

A *program* over $\mathcal{R}$ is specified by a sequence of *declarations* specifying the auxiliary db schema $\mathcal{S}$, followed by a sequence of *instructions*. An auxiliary schema $S\in\mathcal{S}$ can either be declared **shared** or **private**. Intuitively, instances of auxiliary shared schema are identical in all states of a temporal instance, that is if $\mathcal{J}$ is a temporal instance over $\mathcal{S}$ and S is shared then $\mathcal{J}[i](S)=\mathcal{J}[j](S)$ for all $i,j\in[1..|\mathcal{J}|]$. No such restriction is enforced for private schemas. The instructions of a program are:

[*temporal moves*] **left** and **right**.

[*assignment*] $S:=\varphi(\vec{x})$, where $S\in\mathcal{S}$ and φ is a FO formula over $\mathcal{R}\cup\mathcal{S}$ with free variables $\vec{x}$.

[*while loop*] **while** $Cond$ **do** $Body$ **end**, where $Cond$ is a closed FO formula over $\mathcal{R}\cup\mathcal{S}$ and $Body$ a sequence of instructions. While loops can be nested.

EXAMPLE 2.1 The program $\mathcal{P}$ below uses an auxiliary shared proposition B.

while $\neg Last$ **do** { **right** ; $B:=\neg B$ } **end** ;

Indeed, the value of B returned by $\mathcal{P}$ evaluated over $\mathcal{I}$ is $true$ iff `t-even`$(\mathcal{I}) = true$.

Intuitively, the evaluation of a program $\mathcal{P}$ over a temporal instance $\mathcal{I}$ proceeds as follows. We use a temporal cursor, called *current time point*, denoted *ctp*, which is assigned initially to 1. A temporal instance $\mathcal{J}$ over the auxiliary db schema $\mathcal{S}$ is initialized with the empty temporal instance $\emptyset_{\mathcal{S}}^{|\mathcal{I}|}$. Instructions of $\mathcal{P}$ are executed sequencially. Each instruction has some effect either on the instance $\mathcal{J}$ or on *ctp* as explained below:

• **left** (resp. **right**) decreases (resp. increases) *ctp* by 1. If $ctp=1$ (resp. $ctp=n$) before the execution of **left** (resp. **right**) then *ctp* remains unchanged.

• $S:=\varphi(\vec{x})$ changes the instance $\mathcal{J}$ only over S. Let ans be the answer of the FO query $\varphi(\vec{x})$ evaluated over the static instance $(\mathcal{I}+\mathcal{J})[ctp]$ (i.e. over the snapshot of $\mathcal{I}+\mathcal{J}$ at the current time point *ctp*).

– If S is private then $\mathcal{J}$ changes over S only at *ctp*, that is $\mathcal{J}[ctp](S)$ becomes ans;

– if S is shared then $\mathcal{J}$ changes over S everywhere, that is $\mathcal{J}[i](S)$ becomes ans for all $i\in[1..n]$.

• **while** $Cond$ **do** $Body$ **end** executes all instructions of $Body$ until either $Cond$ becomes false, or until a repetition of the effect of the execution of $Body$ is obtained. The effect of one loop iteration is measured by observing configurations. A configuration, at some point of the evaluation of a program, is the pair $(\mathcal{J},ctp)$ where $\mathcal{J}$ is the current instance of the auxiliary db schema $\mathcal{S}$ and *ctp* is the current time point. Thus a while loop stops as soon as the configuration obtained after an iteration is the same as the one at its beginning. The reader should pay attention to the fact that a repetition of the auxiliary instances over $\mathcal{S}$ is not sufficient to exit the loop: if *ctp* changes then the evaluation of the body of the loop is reactivated. Of course, if neither the condition $Cond$ of the loop becomes false nor the sequence of configurations reaches a repetition, then the semantics of both the while loop and the program it belongs to are undefined.

Finally, a T-WHILE query over $\mathcal{R}$ is specified by a T-WHILE program $\mathcal{P}$ over $\mathcal{R}$ and a distinguish shared schema S_{rep} whose instance, at the end of the evaluation of $\mathcal{P}$, is intended to collect the answer of the query. Let us emphasize that T-WHILE is a query language that maps temporal instances to static relations although it computes auxiliary temporal instances. Indeed, note that the size of these auxiliary temporal instances is exactly the size of the input temporal instance.

The temporal language TS-FO The definition of this language only makes sense when considering a temporal instance over $\mathcal{R}$ as a standard instance over the time-stamped db schema $\mathcal{R}^{est}$ where $\mathcal{R}^{est}$ is obtained from $\mathcal{R}$ by adding

one attribute T of type "time" to each relation schema $R \in \mathcal{R}$. The domain of the attribute T is the set of positive integers. It is well known that any temporal instance $\mathcal{I}$ over $\mathcal{R}$ can be transformed into a time-stamped instance over $\mathcal{R}^{est}$, and vice versa. For the sake of our presentation, the set of FO formulas over the time-stamped schema $\mathcal{R}^{est}$ is denoted FO* in order to distinghish these formulas from FO formulas over $\mathcal{R}$.

The language TS-FO is defined as a restriction of FO*: free variables in a TS-FO formula are required to be "data" variables.

EXAMPLE 2.2 Let R be a unary schema. The formula $\varphi(t_1, t_2)$ below is a FO* formula using one data variable x and two free variables t_1 and t_2 of type "time":

$not(t_1{=}t_2) \wedge (\forall x R^{est}(x, t_1) \leftrightarrow R^{est}(x, t_2))$.

The evaluation of such a formula over a time-stamped instance returns pairs of integers (time points).

$\exists t_1 \exists t_2 \varphi(t_1, t_2)$ is a TS-FO query, called `twin`, which checks whether there exists two time points holding the same instance over R.

The temporal language TS-WHILE This language also assumes that the temporal instances over $\mathcal{R}$ are given as time-stamped instances over $\mathcal{R}^{est}$. Then the language TS-WHILE over $\mathcal{R}$ is simply defined as the language WHILE over $\mathcal{R}^{est}$. Thus a TS-WHILE program is build using auxiliary relation schemas, assigments of the form $S{:=}\varphi$ where $\varphi \in$ FO*, and while loops of the form **while** $Cond$ **do** $Body$ **end** where $Cond$ is a boolean FO* formula and $Body$ is a sequence of instructions. The fact that the right part φ of an assignement is a FO* formula entails that the auxiliary schema S may have multiple attributes of type "time". Of course in this framework, there is no such things as shared or private auxiliary schemas and instructions do not include temporal moves.

3 Relational temporal machines

In this section, we introduce *relational temporal machine* (RTM) as an abstract model for general temporal db querying. Recall that here we intend to capture temporal queries where both the input and the output are temporal db instances as explained in the introduction. The model is a Turing-like machine which merges ideas from relational machines [5] and automata [22]. The input tape contains the input temporal instance and is read only. A finite number of working tapes are available for storing intermediate computing and the output. Of course, for both input and working tapes, each cell of the tape is a relational db instance. A relational db schema is associated to each tape for typing the contents of the cells. Over each tape, a cursor is used to scan the corresponding temporal instance and for accessing the contents of a cell, i.e. a relational instance. A cursor is allowed to move in both directions (left and right). A finite number of registers is introduced in order to store time points over the input tape; the only operation on registers (apart from assignment) is a comparison with the input cursor. As usual, transitions specify the changes of state, of registers and of the tape contents as well as the moves of the cursors. Transition may use a query in order to modify the contents of a cell (pointed by a cursor). Queries are FO formulas specified over the input and auxiliary schemas. They are evaluated on the relational instance composed by the contents of the cells pointed by the input and auxiliary cursors.

Next, we assume that $\mathcal{R}$ is the input schema and that the input tape stores an instance $\mathcal{I}$ over $\mathcal{R}$. Formally, a relational temporal machine $\mathcal{M}$ over $\mathcal{R}$ working with m tapes and k registers, denoted RTM$_k^m$, is specified by:

- a finite number m of db auxiliary schemas $\mathcal{S}_1 \ldots \mathcal{S}_m$ pairwise disjoint, one for each of the m working tapes; an output tape given by $out \in [1..m]$,
- a finite set of states $\mathcal{E}$ among which $start$ is the initial state; a subset $\mathcal{E}_f$ of final states,
- a finite set $\mathcal{T}$ of transitions of the form $(q, P, s) \rightarrow (ns, a)$ where:
  - q is a boolean FO query over $\mathcal{R} \cup_{i=1..m} \mathcal{S}_i$,
  - $P \subseteq [1..k]$ is a subset of the k register indices,
  - s and ns are states,
  - a is an action among $left(i)$, $right(i)$ where $i \in [0..m]$, $erase(i)$, $create(i)$ where $i \in [1..m]$, $update(S, \varphi)$ with $S \in \mathcal{S}_i$, for $i \in [1..m]$ and φ a FO query over $\mathcal{R} \cup_{i=1..m} \mathcal{S}_i$, and finally $reg(j)$ with $j \in [1..k]$.

A configuration of an RTM$_k^m$ $\mathcal{M}$ on input $\mathcal{I}$ is specified by a tuple $[[c_0, c_1 \ldots c_m], s, \theta, [\mathcal{I}_1 \ldots \mathcal{I}_m]]$ where c_i are the input and auxiliary cursor values, θ is a register assignment and $\mathcal{I}_i$ is a temporal instance over $\mathcal{S}_i$. It is required that $c_0 \in [0..|\mathcal{I}|]$ and $c_i \in [0..|\mathcal{I}_i|]$, for $i \in [1..m]$. The initial configuration is $[[0 \ldots 0], start, \theta_0, [\emptyset^0_{\mathcal{S}_1} \ldots \emptyset^0_{\mathcal{S}_m}]]$ where $\theta_0(j){=}0, \forall j \in [1..k]$. A configuration whose state s is in $\mathcal{E}_f$ is accepting and then the RTM$_k^m$ $\mathcal{M}$ outputs the temporal instance $\mathcal{I}_{out}$.

Given a configuration $[[c_0, c_1 \ldots c_m], s, \theta, [\mathcal{I}_1 \ldots \mathcal{I}_m]]$, the transition $(q, P, s) \rightarrow (ns, a)$ applies iff

- $P = \{l \mid l \in [1..k], \theta(l) = c_0\}$, and
- $q(J){=}true$ where J is the relational instance over $\mathcal{R} \cup_{i=1..m} \mathcal{S}_i$ defined by $J[\mathcal{R}]{=}\mathcal{I}[c_0]$ and $J[\mathcal{S}_i]{=}\mathcal{I}_i[c_i]$.

It leads to the new configuration $[[c'_0, c'_1 \ \ldots \ c'_m], s, \theta_n, [\mathcal{J}_1 \ldots \mathcal{J}_m]]$ where (we only mention changes):

- if a is $right(i)$ (resp. $left(i)$) then[1] if $|\mathcal{I}_i| \neq 0$ then $c'_i{=}max\{c_i{+}1, |\mathcal{I}_i|\}$ (resp. $c'_i{=}min\{c_i{-}1, |\mathcal{I}_i|\}$),

[1] for the case i=0, one should take $\mathcal{I}_0{=}\mathcal{I}$

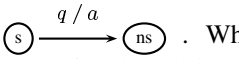
– if a is $erase(i)$ then $\mathcal{J}_i$ is obtained by removing the last cell of the tape $\mathcal{I}_i$; as side effects, if $c_i{=}|\mathcal{I}_i|$ then $c'_i{=}c_i{-}1{=}|\mathcal{J}_i|$, and if $\theta(j){=}|\mathcal{I}_i|$ then $\theta_n(j){=}\theta(j){-}1{=}|\mathcal{J}_i|$,

– if a is $create(i)$ then $\mathcal{J}_i$ is obtained by extending the tape $\mathcal{I}_i$ with a new empty cell,

– if a is $update(S,\varphi)$ with $S{\in}\mathcal{S}_i$, if $c_i{\neq}0$, then $\mathcal{J}_i[c_i](S){=}\varphi(J)$, where J is defined as above (nothing else changes),

– if a is $reg(j)$ then $\theta_n(j){=}c_0$,

Before we start the analysis of our model, let us illustrate the definition of RTM by presenting an example. Assume that the temporal database stores a sequence of picture descriptors. Each state is a picture and each picture contains circles and triangles which can be either blue or red. The RTM given in Figure 2 takes as input a sequence of pictures and split each picture in order to separate red forms from blue ones: red forms are placed in a picture that preceeds the blue ones which are placed in another picture. Thus, in general, the result of this "color based splitting" is a temporal instance whose size is greater or equal to the input size. Figure 1 gives an input sequence of pictures and the expected output. Figure 2 represents the RTM implementing the color based splitting .
A transaction $(q,P,s){\rightarrow}(ns,a)$ is drawn like this (P is omitted because the example does not require it):



. When the boolean query q is $true$, the arrow is labelled by $\_/\ a$, and when there no action, it is labelled by $q\ /\ \_$.

RTM versus relational machines In case temporal instances are represented by timestamped relations (thus in case of working under the explicit time representation), the most straigthforward model for temporal querying that one may think of, is the standard relational machines. However, the standard relational machine model is not powerful enough for captuting queries whose output length has to be greater than the input size. This is due to the fact that standard relational machines do not invent "new" value in their result and thus the temporal domain cannot be changed by the query. The previous example of "color based splitting" shows that RTM are able to define such queries.

RTM genericity Genericity is a property which captures one of the main features of database system functional architecture, the so called physical independance principle: queries are assumed to only use information provided by the abstract view of data which is, of course, independent of the internal data representation. More formally, generic computation is insensitive to automorphisms of the input. Clearly, RTM specify generic computations.

Determinism Next, we will concentrate on deterministic RTMs, ensuring that in each configuration at most one transition applies. The source of non determinism of an RTM $\mathcal{M}$ comes from the boolean queries q in a transition $(q,P,s){\rightarrow}(ns,a)$. One way to enforce determinism is to require that given a state s, there exists at most two kinds of transitions, the ones with premises of the form (q,P,s) and the others with premises of the form $(\neg q,P,s)$. In other words, s "determines" q.
Next, RTM_k^m denotes the temporal queries computable by deterministic RTM_k^m.

Register The reader should be aware about the fact that registers do not add any expressiveness to our model. Registers have been introduced in order to simplify the specification of RTMs and also because they are needed useful when considering restricted version of our model as discussed in section 4.

PROPOSITION 3.1 $\text{RTM}_k^m{=}\text{RTM}_0^{m+1}$

Sketch of proof : The RTM_0^{m+1} $\mathcal{N}$ that simulates an RTM_k^m $\mathcal{M}$ is very much like $\mathcal{M}$. Its extra tape (let us call it the reg-tape) is used to simulate the registers over the input tape. Its size is at most the size of the input tape. It uses a unary schema Reg and $k{+}1$ new constants $r_0, r_1 \ldots r_k$: the constant r_0 is meant to keep track, on the reg-tape, of the input cursor position for technical reasons, and for $j{\in}[1..k]$, the constant r_j keeps track of the value $\theta(j)$ of the $\mathcal{M}$'s registers. It is rather immediate to change the set of transitions of $\mathcal{M}$ in order to enforce that if after some step of the computation, the configuration of $\mathcal{M}$ is $[[c_0, c_1 \ldots c_m],\ s,\ \theta,\ [\mathcal{I}_1 \ldots \mathcal{I}_m]]$ then the corresponding steps of $\mathcal{N}$'s computation leads to a configuration $[[c_0, c_1 \ldots c_m, c_{m+1}],\ s,\ [\mathcal{I}_1 \ldots \mathcal{I}_m, \mathcal{I}_{m+1}]]$ such that: (i) $c_0{=}c_{m+1}$, (ii) $r_0{\in}\mathcal{I}_{m+1}[t][Reg]$ iff $t{=}c_0$, and (iii) for $j{\in}[1..k]$, $r_j{\in}\mathcal{I}_{m+1}[t][Reg]$ iff $\theta(j){=}t$. □

Auxiliary tapes The RTM model can be simplified with respect to the number of auxiliary tapes (or temporal instances) required. The first result below says that it is sufficient to consider RTMs with only two auxiliary tapes and moreover that one of these two tapes is a store. A store is a one-cell tape. Such devices are called `extended one-tape` RTMs and denoted RTM_k^{1+}.

PROPOSITION 3.2 $\text{RTM}_k^m{=}\text{RTM}_k^{1+}$

Sketch of proof : Let us show that any RTM_k^m $\mathcal{M}$ can be simulated by an extended one-tape RTM_k^{1+} $\mathcal{N}$. Intuitively, the m auxiliary tapes of the RTM_k^m $\mathcal{M}$ are concatenated on the single temporal auxiliary tape (let us call it temp-tape) of the extended one-tape RTM_k^{1+} $\mathcal{N}$. The db schema of the temp-tape is composed by the schemas $\mathcal{S}_i$, for $i{\in}[1..m]$ and

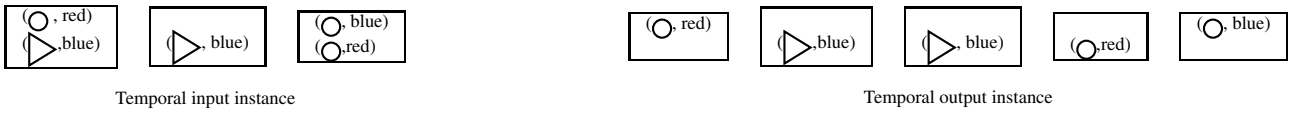
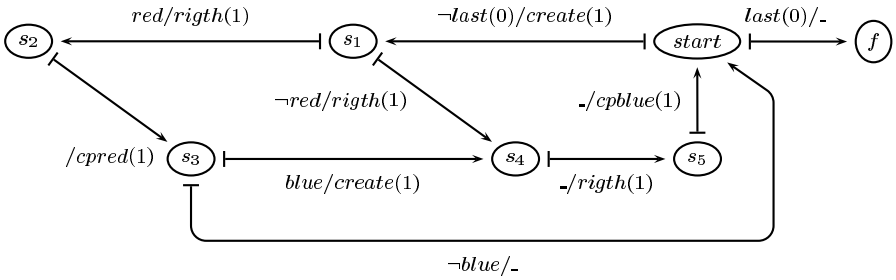


Figure 1. An example of a color based splitting.



queries and actions	
last(0)	checks end of input
red	$\exists x P(x, red)$
blue	$\exists x P(x, blue)$
cpred(1)	Update($Q.P(x,y) \wedge y{=}red$)
cpblue(1)	Update($Q.P(x,y) \wedge y{=}blue$)

Figure 2. The RTM for color based splitting.

a unary schema Cur for the purpose of keeping track, over temp-tape, of the m $\mathcal{M}$'s auxiliary cursors. Indeed, we use $m{+}1$ new constants $cur_1 \ldots cur_m$ and sep: for $i{\in}[1..m]$, the constant cur_i keeps track of the $\mathcal{M}$'s cursor over the i^{th} tape and the constant sep is used as a separator between the concatenated tapes.

Recall that only one cursor is available over each tape and that the transitions of $\mathcal{M}$ make use of queries (in the premise or/and the action) that are evaluated over relations pointed by the input and the m auxiliary cursors. The purpose of the store is to handle these relations, in order for $\mathcal{N}$ to be able to evaluate such queries. Thus the db schema associated with the store is composed by the schemas $\mathcal{S}_i$.

Now it is quite easy to change/modify the set of transitions of $\mathcal{M}$ in order to enforce that, if after some step of the computation, the configuration of $\mathcal{M}$ is $[[c_0, c_1 \ldots c_m], s, \theta, [\mathcal{I}_1 \ldots \mathcal{I}_m]]$ then the corresponding steps of $\mathcal{N}$'s computation lead to a configuration $[[c_0, c'_1, 1], s, \theta, [\mathcal{J}_1, \mathcal{J}_2]]$ such that: (i) the value of c'_1 does not really matter and it may be assumed that $c'_1{=}1$, (ii) the temp-tape $\mathcal{J}_1$ contains (nothing else than) $\mathcal{I}_i[t][R_i]$ in the cell $|\mathcal{I}_1|{+}\ldots{+}|\mathcal{I}_{i-1}|{+}i{-}1{+}t$, $Cur(cur_i)$ in the cell $|\mathcal{I}_1|{+}\ldots{+}|\mathcal{I}_{i-1}|{+}i{-}1{+}c_i$, $Cur(sep)$ in the cells $|\mathcal{I}_1|{+}\ldots{+}|\mathcal{I}_{i-1}|{+}i{-}1$ for $i{\in}[2..m]$, and (iii) $\mathcal{J}_2[1][R_i]{=}\mathcal{I}_i[c_i][R_i]$. □

The reader may have noticed the immediate relationship between extended one-tape RTMs and T-WHILE languages although extended one-tape RTMs are more powerful: the temporal auxiliary tape (resp. the store) of an extended one-tape RTM plays a role similar to the private (resp. shared) auxiliary schemas of a T-WHILE program, although, the auxiliary instance of a T-WHILE program has always the size of the input temporal instance which is not the case for a RTM. This result will serve in the next section to study complexity issues.

The next result is quite strong and may seem surprising at a first glance although it turns out to be a generalization of the normalisation of RM developed in [5]. It tells us that, indeed, any RTM^m_k $\mathcal{M}$ can be simulated by an RTM^1_k $\mathcal{N}orm$.

THEOREM 3.3 $\text{RTM}^m_k{=}\text{RTM}^1_k$

Sketch of proof : The proof strongly relies on the technics developed in [5] for exhibiting a normal form for `loosely coupled Generic Machine`. Indeed we could have stated our result more precisely and provide details about the "shape" of the RTM^0_k $\mathcal{N}orm$ as follows:

$$\mathcal{N}orm{=}\mathcal{N}_{\text{equiv}}; \mathcal{N}_{\text{load}}; \mathcal{N}_{\text{comp}}; \mathcal{N}_{\text{res}}$$

meaning that the RTM $\mathcal{N}orm$ works in four phases:

- $\mathcal{N}_{\text{equiv}}$ phase: One should keep in mind that, the ultimate goal of this phase is to encode a superset of the auxiliary relations computed when running the RTM^m_k $\mathcal{M}$ over the input temporal instance. Thus, this step, as in [5], is dedicated to the computation of a relation $R^r_<$ which is a representation of an ordered partition $\{\delta_1 \ldots \delta_d\}$ of the set $\Delta_r(\mathcal{I})$ of the r-ary tuples build from the active domain of the input temporal instance $\mathcal{I}$. For the sake of simplicity and w.l.o.g. we assume that all schemas have the same arity r and thus that all queries used by the RTM $\mathcal{M}$ have r free variables. The partition is based on the equivalence relation $\equiv^r_{\mathcal{I},\mathcal{Q}}$ and it characterizes r-tuples that cannot be distinguished by any composition of queries used by $\mathcal{M}$. Note that $\mathcal{M}$ uses a finite set of queries $\mathcal{Q}$. The closure $\mathcal{Q}^*$ of $\mathcal{Q}$ under composition is a generalization of the notion of closure provided in [5]. It should carefully take into

account the ability of the machine $\mathcal{M}$ to move its cursor over the input tape. Thus the set of formula in $\mathcal{Q}^*$ depends of the size n of $\mathcal{I}$. It is defined as the smallest set of formulas with r free variables over the db schema $\mathcal{R}$ such that: (a) for each $R{\in}\mathcal{R}$ and $t{\in}[1..n]$, $R(\vec{x})[t]$ is in $\mathcal{Q}^*$; (b) for each formula $\varphi(T_{i_1},\ldots,T_{i_p})$ written using p occurrences of schemas in $\mathcal{R}\cup_{i=1..m}\mathcal{S}_i$ and for $\varphi_1,\ldots,\varphi_p$ in $\mathcal{Q}^*$, $\varphi(\varphi_1,\ldots,\varphi_p)$ is in $\mathcal{Q}^*$ where $\varphi(\varphi_1,\ldots,\varphi_p)$ denotes the formula obtained by replacing the j-th occurrence of a schema by φ_j. Now, we have that $u{\equiv}^r_{\mathcal{I},\mathcal{Q}}v$ iff for every composition q of queries in $\mathcal{Q}^*$, we have $u{\in}q(\mathcal{I})$ iff $v{\in}q(\mathcal{I})$. The definition of $q(\mathcal{I})$ extends slightly that of [5] and is easy to infer from the definition of $\mathcal{Q}^*$. It is explained using an example. Assume that $r{=}1$ and $\mathcal{R}{=}\{R_1,R_2\}$ and let $\mathcal{I}$ be a temporal instance of size 3. Now assume that q is $R_1(x)[1]{\wedge}R_2(x)[3]$. Then given a valuation ν for x, $u{=}\nu(x){\in}q(\mathcal{I})$ iff $u{\in}\mathcal{I}[1](R_1)$ and $u{\in}\mathcal{I}[3](R_3)$.
In [5], it is showed that the relation $R^r_<$ can be computed by a fixpoint query. In our case, we can show that $R^r_<$ can be computed by a simple RTM^1_m $\mathcal{N}_{\text{equiv}}$ (indeed, it can be computed by a $\text{T-WHILE}_{\text{shared}}$ program as well). The RTM $\mathcal{N}_{\text{equiv}}$ also computes the action tables of the queries $\mathcal{Q}$ based on the partition. Roughly, for each query q, R_q describes the effect of q in terms of partition blocks: $\delta_{i_1}\times\ldots\times\delta_{i_q}\times\delta_{i_{q+1}}{\in}R_q$ iff $\delta_{i_{q+1}}\subseteq q(\delta_{i_1}\ldots\delta_{i_q})$. Note that $R^r_<$ and the action tables are stored in the first cell of the auxiliary tape.

$\underline{\mathcal{N}_{\text{load}}\text{ phase:}}$ Intuitively, during this phase, the RTM^1_k $\mathcal{N}_{\text{load}}$ encodes on the auxiliary tape (starting at the 2nd cell) the relation $R^r_<$, the action tables and the input instance $\mathcal{I}$. The auxiliary tape is then used as a Turing machine tape. The encoding relies on the fact that the relation $R^r_<$ is a representation of an ordered partition and thus an integer (or its binary encoding) may be substituted to each block of the partition.

$\underline{\mathcal{N}_{\text{comp}}\text{ phase:}}$ The RTM^1_k $\mathcal{N}_{\text{comp}}$ simulates the run of the RTM $\mathcal{M}$ on the input $\mathcal{I}$. Roughly, $\mathcal{N}_{\text{comp}}$ is a Turing machine that works on the encoding previously stored on the auxiliary tape.

$\underline{\mathcal{N}_{\text{res}}\text{ phase:}}$ The result produced by the Turing machine $\mathcal{N}_{\text{comp}}$ needs to be decoded to provide a temporal instance whose state (cell contents) are sets of tuples. This means that the integers encoding the partition blocks have to be replaced by the corresponding tuples. This is made possible using only one auxiliary tape because the computation of the relation $R^r_<$ can be (re)done and stored in any cell of the auxiliary tape.

□

In the previous proof, the one-tape RTM machine uses the same number of registers as the original one. This entails that:

COROLLARY 3.4 $\text{RTM}^m_k{=}\text{RTM}^1_0$.

The normalization of RTM is an interesting result by itself. It is a witness, once more, that the technics provided in [5] are quite powerful. Before considering complexity issues, we would like to briefly comment on the normalization of T-WHILE programs. Although we are not yet able to provide a detailed proof, we claim that T-WHILE programs can be normalized in the same way as WHILE programs are in [5]. However, this normalization cannot be realized without using shared auxiliary schemas in order to store the relation $R^r_<$ and make it available to any step of the program evaluation. Because shared auxiliary schemas of T-WHILE programs play a role closely related to the store of the extended one-tape RTM, this is in favour of working with extended one-tape RTM rather than with the one-tape machines.

4 Space complexity: RTM versus temporal languages

Now, we are going to investigate the connection, in terms of complexity and expressiveness, between our general temporal computation model RTM and some languages, namely T-WHILE and TS-WHILE, as well as some restriction of these languages. As already mentioned, we focus on space complexity and the main issue addressed in this section is to measure temporal computation complexity in terms of the auxiliary space required. Recall that the size of a temporal instance $\mathcal{I}{=}I_1\ldots I_n$ is n. We are not interested here in the size of the active domain as usually assumed.

Next, we consider extended one-tape RTMs, the ones using one temporal tape plus a store. The set of RTMs whose runs make use of an auxiliary temporal tape of size linear (resp. polynomial) in the size of the input temporal instance is denoted LinSpace-RTM (resp. PSpace-RTM).

The question that arises here is why do we consider extended one-tape RTMs rather than one-tape RTMs? Indeed, as we study linear and sublinear space complexities, the one-tape RTMs are useless because of the polynomial encoding of the input tape on the auxiliary tape.

The first result of this section establishes a strong correspondence between LinSpace-RTMs and T-WHILE queries in the one hand, and between PSpace-RTMs and TS-WHILE queries in the other hand. The reader should be aware that this result holds iff the output tape of the extended one-tape RTM is restricted to be the store.

THEOREM 4.1 *The following holds (under the above restriction):*

- LinSpace-RTM = T-WHILE, *and*
- PSpace-RTM = TS-WHILE.

Sketch of proof : LinSpace-RTM = T-WHILE: Proving that any T-WHILE program $\mathcal{P}$ can be simulated by a linear RTM does not present any difficulty. Indeed, we show that the RTM simulating $\mathcal{P}$ uses an auxiliary tape of size n when the input temporal instance is of size n. Building the RTM $\mathcal{B}$ is done by building an RTM for each instruction of the program and by concatenating these machines.
Proving that any linear RTM $\mathcal{B}$ can be simulated by a T-WHILE program $\mathcal{P}$ is not difficult either. The only technical point deserving to be mentioned concerns the fact that a T-WHILE program works with auxiliary temporal instances whose sizes are exactly that of the input temporal instance. Thus, assuming that the RTM $\mathcal{M}$ is of complexity cn where n is the size of the input, in order to simulate the auxiliary tape of $\mathcal{B}$ by the program $\mathcal{P}$, we declare c copies $\mathcal{S}_i$ of the auxiliary schema $\mathcal{S}$ associated to the temporal tape of $\mathcal{M}$: the auxiliary temporal instance $\mathcal{J}$ of $\mathcal{P}$ can be viewed as a multidimensional tape of size n and it is managed by $\mathcal{P}$ in such a way that the "dimension" R_i of $\mathcal{J}$ encodes the i^{th} segment of size n of the temporal tape of size at most cn.

PSpace-RTM = TS-WHILE: First, recall that, for TS-WHILE, (1) the input schema is time stamped and (2) the auxiliary schemas may have multiple attributes of type "time". Recall also that the queries (used in assignments) in TS-WHILE are FO* queries. For simulating a TS-WHILE program $\mathcal{P}$ by an RTM, it is more convenient wlog to target multi-tape devices. Thus now simulating a TS-WHILE program $\mathcal{P}$ by an RTM $\mathcal{B}$ essentially requires to show how to manage general time stamped relations by implicit temporal instances and how to translate FO* queries. Assume that $P^*(\vec{A}, T_v, \ldots, T_1)$ is a relation schema with v attributes T_i of type "time" ($\vec{A}$ is a vector of data attributes). We associate with P^* an auxiliary temporal tape whose schema is $P(\vec{A})$. The RTM $\mathcal{B}$ will encode an instance I^* of P^* as the temporal instance $\mathcal{I}$ over P:
- if $v{=}0$ then $|\mathcal{I}|{=}1$ and $\mathcal{I}[1][P]{=}I^*$, and
- if $v{>}0$ then $|\mathcal{I}|{=}n^v$ where $n{=}max(\cup_{i=1..v}\pi_{T_i}(I^*))$ and then the j^{th} cell $\mathcal{I}[j]$ of the tape contains (over P) the data projection of the tuples of I^* time-stamped by $(i_v, \ldots, i_1)$ with $j{=}(\sum_{u\in[1...v]}(i_u{-}1){\times}n^{u-1}){+}1$. Intuitively, the first state of $\mathcal{I}$ stores the data time-stamped by $(1\ldots1, 1)$ in I^*, the n^{th} state of $\mathcal{I}$ stores the data time-stamped by $(1\ldots1, n)$ in I^*, etc. Now, in order for the RTM $\mathcal{B}$ to compute the encoded effect of a FO* query, it needs to properly manage the cursor which does not present any difficulty. Thus we prove that any TS-WHILE program can be simulated by a RTM $\mathcal{B}$ working on ℓ temporal auxiliary tapes. The size of each tape is polynomial in the size of the input and thus we can conclude by using (the proof) of proposition 3.2.
The proof of the converse i.e. PSpace-RTM$\subseteq$ TS-WHILE is based on an encoding of the temporal auxiliary tape of $\mathcal{B}$ by a general time-stamped auxiliary relation managed by $\mathcal{P}$. This encoding is more or less the inverse of the encoding used in the first part of this proof. It takes the $\mathcal{B}$ auxiliary tape whose length is at most n^v, and maps it to an instance of a predicate R^* having v attributes of type "time". □

The last part of this section focusses on RTMs whose space complexity is constant. Indeed constant space RTMs are the devices using an auxiliary store and no auxiliary temporal tape. Such devices are next called `relational register machine` (RRM$_k$), because registers are necessary components in that case. We would like to highlight that such class of temporal queries has never been studied before.

Remark: The reader should pay attention to the fact that proposition 3.1 stating that registers do not add expressiveness to the model, does not apply when considering RRM$_k$ because registers can no more be simulated using a temporal tape.

In this section, we also use `relational pebble machine` RPM$_k$ which are straightforward restrictions of RRM$_k$ that enforces a stack discipline over the k registers. Thus, an RPM$_k$ requires to order the registers, to keep track of the top of the stack whose value is used as the cursor over the input temporal instance. A configuration is given as before by a tuple $[i, s, \theta, J]$ although now $i{\leq}k$ is the index of the top of the stack and $\theta(i)$ is the value of the cursor over the input tape. The set of actions of a transition are modified and simplified: $right, left, update(S, \varphi)$ are defined as before more or less; $push$ increments the top stack index, if possible and the new configuration $[j, s', \theta', J']$ is such that $j{=}i{+}1$ and $\theta'(j{+}1){=}\theta(i)$, and finally $lift$ decrements the top stack index. Notice that the size of the stack of a RPM$_k$ is bounded by k.

It is interesting to know that:

PROPOSITION 4.2 *Propositional* RPM *expresses exactly the regular languages over finite alphabet.*

A propositional RPM takes as input a propositional temporal instance. Wlog, we assume that the auxiliary db schema of an RPM is a set of 0-ary relation schemas (i.e. propositions).

Sketch of proof : The proof is based on mappings from words over a finite alphabet to temporal instances over 0-ary schemas and vice versa. The mapping from words to propositional temporal instances is rather obvious: each element a of the finite alphabet A is mapped to a proposition B_a (i.e. a 0-ary relation schema) and the word $w{=}a_1, a_2 \ldots a_f$ is mapped to the temporal instance $\mathcal{I}$ such that $\mathcal{I}[t]{\models}B$ iff $B{=}B_{a_t}$. The inverse mapping, from propositional temporal instances to words, assumes that an element a_s of the

alphabet is reserved for each possible subset s of propositions and a temporal instance $\mathcal{I}$ is then mapped to the word $w{=}a_1, a_2, ..., a_{|\mathcal{I}|}$ such that for $t{\in}[1..|I|]$, $a_t{=}a_s$ where s is the set of propositions "true" at $\mathcal{I}[t]$.

The transformation of a determinist automata for a regular language $\mathcal{L}$ into an RPM is straightforward.

In order to show that the languages recognized by propositional RPMs are regular, the idea is to show that any propositional RPM can be simulated by a pebble automata PA over a finite alphabet and, then we conclude by applying the following result [19]: PA $\subset$ MSO*. □

This proposition entails that:

COROLLARY 4.3 RPM $\subset$ RRM.

Sketch of proof : The strict inclusion is entailed by the fact that for any fixed integer n, the language $\mathcal{L}{=}a^n b^n$ can be "recognized" by a propositionnal RRM. □

We now are investigating languages that express RPM computable queries, resp. RRM computable queries .

THEOREM 4.4 *We have:*

- TS-FO$\subsetneq$ RPM,
- ETL$\subsetneq$ RPM, *and*
- T-WHILE$_{\text{shared}}\subseteq$ RPM. *(***claim:** *the inclusion is strict)*

First let us present the language T-WHILE$_{\text{shared}}$: it is the restriction of T-WHILE to the case where only shared auxiliary relations can be used by the programs.

Recall that the languages TS-FO and ETL are not comparable [3]: TS-FO can express the query `twin` ($\exists t \exists t' \forall x(R(x,t) \leftrightarrow R(x,t'))$) but ETL cannot; ETL can express the query `t-even` (`t-even`($\mathcal{I}$)=true iff $|\mathcal{I}|$ is even) but TS-FO cannot. On the other hand, TS-FO can express the query $\forall$-`twin` ($\forall t \exists t'[t \neq t' \wedge \forall x(R(x,t) \leftrightarrow R(x,t'))]$) but T-WHILE$_{\text{shared}}$ cannot; T-WHILE$_{\text{shared}}$ can express transitive closures which cannot be expressed neither by TS-FO nor ETL. Finally, we claim that the languages ETL and T-WHILE$_{\text{shared}}$ are incomparable. ETL queries involving nested formulas can probably not be translated into a T-WHILE$_{\text{shared}}$ program roughly because T-WHILE$_{\text{shared}}$ lacks the ability to store (or mark) any time point over the input.

Sketch of proof :

TS-FO$\subsetneq$ RPM: The proof of the inclusion involves rather technical details. The subformulas of a TS-FO formula are in FO*. Thus we need to show how to translate a FO* formula by an RPM. The main idea is to simulate the variables of type "time" of the FO* formulas via pebbles. The only critical point is to show that the stack discipline is adequate for the simulation.

The inclusion is strict because the query `t-even` is RPM computable but cannot be expressed in TS-FO.

ETL$\subsetneq$ RPM: By induction on the structure of the ETL formula φ, we show that there exists an RPM $\mathcal{M}_\varphi$ such that, (1) the auxiliary db schema of $\mathcal{M}_\varphi$ includes a relational schema R_φ whose arity is the number of free variables of φ, and such that given any temporal input instance $\mathcal{I}$, for any $t{\in}[1..|\mathcal{I}|]$, any run of $\mathcal{M}_\varphi$ starting with the configuration $[1, s_\varphi, \theta(1){=}t, I]$ with $I[R_\varphi]{=}\emptyset$ stops with configuration $[1, f_\varphi, \theta(1){=}t, J]$ with $J[R_\varphi]{=}\{\mu(\vec{x}) \mid (\mathcal{I}, \mu, t){\models}\varphi(\vec{x})\}$. Assume that φ is atomic i.e. of the form $R(\vec{x})$[2], then it is easy to see that the RPM $\mathcal{M}_\varphi$ has two states s_φ and f_φ, and one transition $(true, \emptyset, s_\varphi){\rightarrow}(f_\varphi, update(R_\varphi, (\varphi))$. The induction steps corresponding to disjunction, negation and universal quantification constructs are not difficult. Let us comment the induction step for $\varphi{=}L^\delta(\varphi_a, ...\varphi_g)$ where δ is either $+$ or $-$ and assuming that A is the automata recognizing the language L and for $\alpha{\in}[a..g]$, $\mathcal{M}_{\varphi_\alpha}$ is the RPM simulating φ_α as described above. The main idea is to build $\mathcal{M}_\varphi$ based on the automata A (the behaviour of the automata can be encoded in the store). $\mathcal{M}_\varphi$ will scan the input instance from t up to $|\mathcal{I}|$ (resp. down to 1) if δ is $+$ (resp. is $-$) and at the end will put the input cursor back at t. A relation schema R_q is associated to each state q of the automata A and will serve for storing the relevant tuples according to state q. Computing these tuples will of course be done using (slightly modified copies of) the RPM $\mathcal{M}_{\varphi_\alpha}$.

The inclusion of ETL in RPM is strict because the query `twin` is RPM computable but cannot be expressed in ETL.

T-WHILE$_{\text{shared}}\subseteq$ RPM: The proof of the inclusion is rather immediate. The translation of a T-WHILE$_{\text{shared}}$ program $\mathcal{P}$ into a RPM $\mathcal{M}$ is done (by induction) instruction by instruction. The only technicality is the translation of while loops. The level of nesting of while loops in a program will determine the size of the stack of $\mathcal{M}$. Recall that a while loop execution stops either because of the condition or because two consecutive iterations compute the same configuration. The important point is that a configuration is composed of both the current time point and the auxiliary (shared) relations. The naive approach consisting in pushing the value of the input cursor in the stack at each iteration of the loop will lead to use an unbounded number of pebbles and thus cannot be followed to build our RPM. Thus, intuitively, the idea is to "duplicate" the execution of each iteration or in other words to simulate one iteration of a loop by two executions of its body. The purpose of the first execution is to determine whether it is the last iteration of the loop: it uses a new pebble pushed on the top of the stack, it uses a copy of the auxiliary shared relations and the new pebble is lift once the decision is made. The second execution of the iteration corresponds to the "real" one: it does not use any pebble and it changes the auxiliary shared relations. We show that the RPM that simulates a T-WHILE$_{\text{shared}}$ program $\mathcal{P}$ uses a stack of size at most $k+1$ where k is the maximum

[2] Wlog we disallow constants in φ.

nesting level of while loops in $\mathcal{P}$.

Although we do not have a proof yet, we claim that the inclusion is strict because T-WHILE$_{\text{shared}}$ programs do not have the ability to register current time points since any auxiliary relation is shared. □

The last part of this section is devoted to the presentation of two languages T-WHILE$^{\text{bind}}_{\text{shared}}$ and @TL which are complete with respect to RRM. The first language, T-WHILE$^{\text{bind}}_{\text{shared}}$, is an extension of T-WHILE$_{\text{shared}}$ with temporal variables (t-variables) which are obviously meant to mimic the registers of the RRM devices.

We now define T-WHILE$^{\text{bind}}_{\text{shared}}$ essentially by describing the features that have been added to T-WHILE$_{\text{shared}}$. Thus besides shared auxiliary schemas, the declaration of a T-WHILE$^{\text{bind}}_{\text{shared}}$ program may include a finite list of t-variables $t_1 \ldots t_k$ which are all initialized to 1. The formulas φ in the right part of assignements $S{:=}\varphi$ belong to FO$^{\text{bind}}$. The language FO$^{\text{bind}}$ over $\mathcal{R} \cup \mathcal{S}$ and $t_1 \ldots t_k$ allows one for new atomic formulas of the form $t_i{=}t_j$ but restrict quantification to apply over "data" variables only. Finally, instructions of the form $\downarrow t_i$ are added in order to bind t-variables to the current time point. The semantics of a T-WHILE$^{\text{bind}}_{\text{shared}}$ program directly follows from that of T-WHILE$_{\text{shared}}$. While loop execution and termination are based on configurations: a configuration is now a triple $(\mathcal{J}, ctp, g)$ where g is a t-variable assignement. A while loop executions stops as soon as either its condition is false or the configuration produced by an iteration is equal to the configuaration at the beginning of this iteration.

EXAMPLE 4.1 The following sequences of instructions places the current time point *ctp* at the time point contained in the t-variable t. It uses two t-variables t and t'.

while not(first) **do** left **end**;
$\downarrow t'$;
while not ($t'{=}t$) **do** (right; $\downarrow t'$) **end**.

The next result states that the language T-WHILE$^{\text{bind}}_{\text{shared}}$ is complete wrt RRM.

THEOREM 4.5 T-WHILE$^{\text{bind}}_{\text{shared}}$=RRM.

Sketch of proof :

T-WHILE$^{\text{bind}}_{\text{shared}} \subseteq$ RRM: Proving this inclusion is very similar to proving that T-WHILE$_{\text{shared}} \subseteq$ RPM and we use quite the same translation, especially for while loops. We build an RRM $\mathcal{M}$ that simulates a T-WHILE$^{\text{bind}}_{\text{shared}}$ program $\mathcal{P}$ that uses a number of registers $(k{+}1)(p{+}1)$ where k is the number of t-variables of $\mathcal{P}$ and p is the maximum nesting level of while loops in $\mathcal{P}$.

RRM $\subseteq$ T-WHILE$^{\text{bind}}_{\text{shared}}$: The proof starts by showing the following lemma:

LEMMA 4.6 *Any* RRM $\mathcal{M}$ *is equivalent to a* RRM *whose transitions use two states only:* $start$ *and* $final$

Intuitively, states of the RRM $\mathcal{M}$ are encoded as boolean in the store.

Thus it remains to show that any 2-states RRM $\mathcal{M}$ can be simulated by a T-WHILE$^{\text{bind}}_{\text{shared}}$ program $\mathcal{P}$ which turns out to be rather immediate. □

The second language presented here, called @TL, is equivalent to T-WHILE$^{\text{bind}}_{\text{shared}}$ although its definition is based on TL and borrows some idea from Hybrid modal logic [10]. The main idea is similar to the one used to define T-WHILE$^{\text{bind}}_{\text{shared}}$: temporal variables (t-variables)[3] are added as propositions in order to mark time points over the input temporal instance; we also introduce a fixpoint operator.

Let us assume that $t_1 \ldots t_k$ are t-variables. Then the @TL language over $\mathcal{R}$ is inductively defined by:

• an @TL atomic formula is either a FO atomic formula over $\mathcal{R}$ or a t-variable t,

• Let φ_1 and φ_2 be @TL formulas, let x be a data variable and t be a t-variable, then $\neg\varphi_1$, $\exists x\varphi_1$ and $\varphi_1 \wedge \varphi_2$, next($\varphi_1$), prev($\varphi_1$) and $@_t\varphi_1$ are @TL formulas,

• $@fpt_S(\phi^{\vec{n}}_{\vec{t}})(\vec{x})$ is a @TL formula where $@fpt(\phi^{\vec{n}}_{\vec{t}})$ is a fixpoint expression where:
$\phi^{\vec{n}}_{\vec{t}}$ is a @TL formula using the new (auxiliary) relation schema S of arity k; this formula has k free data variables. Finaly, $\vec{n}$ and $\vec{t}$ are two vectors of t-variables of same size.

Intuitively, in the formula $\phi^{\vec{n}}_{\vec{t}}$ above, t-variables in $\vec{n}$ are viewed as nominals or constants. This means that, when evaluating the fixpoint expression $\phi^{\vec{n}}_{\vec{t}}$, at each iteration, $\vec{n}$ should be viewed as time points previously computed and $\vec{t}$ as the time points under computation.

EXAMPLE 4.2 The formula $\exists z[z{=}a \wedge @fpt_S(\psi^n_t)(\vec{x}, z)]$ where $\psi^n_t(\vec{x}, z)$ is defined by:

$$[z{=}b \wedge t \wedge \vec{x}{=}\vec{b}] \vee @_n[z{=}a \wedge S(\vec{b}, b) \wedge \varphi(\vec{x})]$$

is equivalent to the hybrid modal logic formula $\downarrow_t \varphi(\vec{x})$ whose semantics assigns the t-variable t to the evaluation (current) time before evaluating $\varphi(\vec{x})$.

We restrict the presentation of the semantics of the @TL language to its specific features. Let $\mathcal{I}$ be a temporal instance over $\mathcal{R}$, let μ be a valuation of data variables and g a valuation of t-variables, then

• $(\mathcal{I}, \mu, g, i) \models t$ iff $g(t){=}i$.

• $(\mathcal{I}, \mu, g, i) \models @_t(\phi)$ iff $(\mathcal{I}, \mu, g, j) \models \phi$ where $j{=}g(t)$,

• $(\mathcal{I}, \mu, g, i) \models (next\phi)$ iff $(\mathcal{I}, \mu, g, i+1) \models \phi$

• $(\mathcal{I}, \mu, g, i) \models (prev\phi)$ iff $(\mathcal{I}, \mu, g, i-1) \models \phi$

[3] Temporal variables are called state variables in the framework of Hybrid modal logic.

• $(\mathcal{I},\mu,g,i)\models @fpt_S(\phi_{\vec{t}}^{\vec{n}})(\vec{x})$ iff $\mu(\vec{x})\in\int$ where s is the relation defined as the limit of $(s_n)_{(n\geq 0)}$ defined simltaneously with $(g_n)_{(n\geq 0)}$ as follows:

– $s_0=\emptyset$ and $g_0(n_i)=1$ for any t-variable n_i of $\vec{n}$.

– if there exists a **single** valuation h of the t-variables such that:
(a) $h(n_i)=g_s(n_i)$ for any t-variable in $\vec{n}$,
(b) $h(u)=g(u)$ for any t-variable u occuring in $\phi_{\vec{t}}^{\vec{n}}$ but neither in $\vec{n}$ nor in $\vec{t}$, and
(c) there exists a valuation ν of the free data variables in $\phi_{\vec{t}}^{\vec{n}}$ such that $(\mathcal{I},\nu,h,i)\models\phi_{\vec{t}}^{\vec{n}}$ then
(†) $s_{s+1}=\{\nu \mid (\mathcal{I},\nu,h,i)\models\phi_{\vec{t}}^{\vec{n}}\}$ and
(†) $g_{s+1}(n_i)=h(t_i)$ for any t-variable n_i of $\vec{n}$.

Above, the semantics of a fixpoint expression relies on the unicity of the t-variable valuation h. This condition is rather strong and clearly, testing if a fixpoint expression is well defined is undecidable. One possible direction to cope with this, may be for instance to choose the least (dualy, the greatest) t-valuation.

EXAMPLE 4.3 Let us consider the fixpoint expression of the previons example. Let us assume that the formula $\varphi(\vec{x})$ is simply $R(x)$ and that the fixpoint expression is evaluated given i=8 and given a temporal instance $\mathcal{I}$ with $\mathcal{I}[8](R)=\{(@),(*)\}$. The simultaneous sequences K_n and g_n are, by definition equal to:

$s_0=\emptyset$	$g_0(n)=1$
$s_1=\{(b,b)\}$	$g_1(n)=8$
$s_2=\{(a,@),(a,*),(b,b)\}$	$g_2(n)=8$, and
$s_3=s_2$	$g_3(n)=g_2(n)$

We can now state that:

THEOREM 4.7 $@\text{TL}=\text{RRM}$ *and thus* $@\text{TL}=\text{T-WHILE}^{\text{bind}}_{\text{shared}}$.

Sketch of proof : Thanks to lemma 4.6, proving that RRM $\subseteq$ @TL can be reduced to show that any 2-state RRM $\mathcal{M}$ can be simulated by a @TL query. This will require one fixpoint expression (and no nesting of fixpoint expression).

Proving the converse, that is @TL $\subseteq$ RRM, is done by induction over the structure of @TL formulas. The difficult case is of course that of a formula build from a fixpoint expression although the underlying idea are borrowed from previous proofs. □

5 Discussion

In this paper, we have introduced a general computation model for temporal database queries. This model is the first that captures queries whose outputs are temporal instances. The results of the paper show that RTM provides an interesting formal tool for analysing standard languages and we claim that this tool opens several further research directions. Some of these directions have been already discussed during the presentation as for instance working on defining a normal form for T-WHILE. [1] investigates normal form of RM and WHILE language as an optimization tool. On the other hand, [2] provides and evaluates practical heuristics based on the normal form. Although it needs to be carrefully studied, we could reasonably expect that such results and directions extend to our framework and that our normal form could be used as well to optimize some temporal computations.

The RTM computation model provides the basis for identifying some critical features of temporal computation which are often implicit when just considering concrete languages. Investigating the features/parameters of RTM seem a promising research direction and may well benefit to more specific domains like streaming. Continuous-query processing over data streams [6] is a very challenging research topic because of its large range of applications, from stock ticks to sensor applications. Formalizing continuous query and streamable queries [15, 14] is under study but a general model still needs to be proposed. For instance, none of the proposed formal models is yet able to take into account manipulations over multiple streams like fusion or in general operations that output a stream from streams which is a critical issue in the framework of distributed sensors applications.

Let us now briefly review some of the questions that may arise when considering some variants of our RTM:

– finite versus infinite temporal instances: we have assumed all along the presentation that the input tape is finite (as well as the auxiliary tapes); it is relevant to relax this assumption in the context of streaming computation for instance.

– controling cursor moves: the general setting makes no restriction on the moves of the cursors neither on the input nor on the auxiliary tapes; studying right only moves and forbiding/restricting reversals are relevant topics, once again in the context of streaming computation but not only. Other restrictions may be of interest as well: synchronisation of the cursors (indeed, T-WHILE implements such a discipline); enforcing cursor moves at each transition especialy in the case of right only moves for capturing a loose real time behavior.

– internal query language: recall that RTM make use of (a finite set) of FO queries either to control transitions or as transition actions. Of course, the impact of restricting queries to be inflationary has to be revisited in our framework. Obviously, one may also need to use a language more powerful than FO, for instance in order to

take into account sophisticated abstract data types. One exiting and inspiring area is musical improvisation [16] whose goal is to generate new musical sequences from (live) musical performances.

Another important research direction focusses on the definition of concrete temporal languages satisfying reasonable complexity criteria or/and complying to some of the restrictions listed above. In [9], we propose a class of languages called SQTL. These languages are based on a paradigm related to windows in the context of streaming. They are based on a two phase process: first, the input temporal instance is sliced to produce a sequence of subsequences of the input (a sequence of slices); slices may overlap or not, they may cover or not the input; secondly, temporal queries are evaluated over each slice although with access to the whole input. These languages need to be refined and further studied from the complexity and the expressivity points of view.

It is not very difficult to use T-WHILE as it is for specifying temporal queries with temporal outputs. It suffices to extend the definition of a query by allowing the output schema to be a private schema. However, by definition, we know in advance that these queries defined through T-WHILE programs will always output temporal instances having exactly the size of the input temporal instance. Thus, an open and exciting issue is the definition of a P-Space temporal language a la T-WHILE.

References

[1] S. Abiteboul, K. Compton and V. Vianu. Queries are easier than you thought (probably). In *Proc. PODS*, pages 23–32, 1992.

[2] S. Abiteboul and A. Van Gelder. Optimizing active databases using the split technique. In *Proc. ICDT*, LNCS vol. 646, pages 171–187, 1992.

[3] S. Abiteboul, L. Herr and J. Van den Bussche. Temporal Connectives versus Explicit Timestamps in Temporal Query Languages. In *Proc. of the VLDB International Workshop on Temporal Databases*, pages 43–57, 1995.

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

[5] S. Abiteboul, V. Vianu. Computing with First-Order Logic. In *Journal of Computer and System Science*, 50(2):309–335, 1995.

[6] A. Arasu, B. Babcock, S. Babu, J. McAlister and J. Widom, Characterizing memory requirements for queries over continuous data streams. In *ACM Trans. Database Syst.*, (29):162–194 , 2004.

[7] N. Bidoit, S. De Amo, and L. Segoufin. Order independent temporal properties. In *Journal of Logic and Computation*, to appear.

[8] N. Bidoit and M. Objois. Temporal Query Languages Expressive Power: muTl versus Twhile, TIME 2005, 74–82, (Revised version in preparation).

[9] N. Bidoit and M. Objois. STQL : a preliminary proposal for t2t languages, Proc. TIME 2007, to appear.

[10] C. Areces and P. Blackburn and M. Marx. Hybrid logics: characterization, interpolation, and complexity, In *Journal of Symbolic Logic*, 66(3):977-1010, 2001.

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

[12] 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.

[13] 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.

[14] M. Grohe, Y. Gurevich, D. Leinders, N. Schweikardt, J. Tyszkiewicz and J. V. den Bussche. Database Query Processing Using Finite Cursor Machines. In *Proc. ICDT*, pages 284-298, 2007.

[15] M. Grohe, C. Koch and N. Schweikardt. Tight Lower Bounds for Query Processing on Streaming and External Memory Data. To appear In *Theoretical Computer Science, Special issue for selected papers from ICALP'05, Track B*.

[16] http://www.ircam.fr/.

[17] 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.

[18] M. Paterson. Tape bounds for time-bounded Turing machines. In *Journal of Computer and System Science*, 6(2):116–124, 1972.

[19] F Neven, Th. Schwentick and V. Vianu. Finite state machines for strings over infinite alphabet. In *ACM Trans comput. Logic*, 5(3):403-435, 2004.

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

[21] http://www.cs.auc.dk/TimeCenter/.

[22] W. Thomas. Languages, automata and logic. In *Handbook of Formal Languages, G. Rozenberg and A. Salomaa editors*, chap. 7, 1997.

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

[24] P. Wolper. Temporal Logic Can Be More Expressive. In *Information and Control*, pages 72–99, 1983.

0-7695-2836-8/07 $20.00 © 2007 IEEE