

SQTL: a Preliminary Proposal for a Temporal-to-temporal Query Language

Nicole Bidoit, Matthieu Objois
Univ. Paris-Sud, UMR CNRS 8623, Orsay F-91405
CNRS, Orsay F-91405
{bidoit,objois}@lri.fr

Abstract

A temporal database can be represented as a finite sequence of relational instances. Queries over such temporal instances usually map the input sequences to relation instances. Motivated by emerging applications, we are investigating a new type of queries over temporal instances, called t2t queries (temporal-to-temporal queries). The issue is to develop languages able to define mappings from temporal instances to temporal instances. The paper proposes and investigates two types of t2t query languages: point wise languages and SQTL languages.

Keywords Temporal databases, temporal query languages.

1 Introduction

Managing temporal or ordered information is one of the central issues of many Computer Science areas. Temporal and ordered information plays a crucial role in a wide range of fields and applications: the web, bioinformatics, medical applications, multimedia applications, animation, simulation, personal information management, data flow processing, streaming, etc. Suitable database models and query languages [15, 7] have been developed to store and support temporal information.

Query languages that deal with temporal information, either based on the relational model or on more sophisticated models such as XML, provide operators allowing for limited forms of information retrievals. Usually, these languages mainly target snapshot queries (what are the data available at some time point?), history extractions (what is the evolution of some data?) and combinations/variations of these extractions. Indeed, well-known formal temporal languages such as TL or TS-FO [8, 12, 7] have very strong limitations:

they give the ability to define mappings from temporal database instances (a sequence of database states) to relations (tables) and totally lack the ability to map sequences of instances to sequences of instances, in a general way.

The purpose of this paper¹ is to introduce a new type of query languages, called temporal-to-temporal (t2t) languages for specifying mappings from temporal instances to temporal instances.

We now illustrate the usefulness of t2t languages.

(1) Let us start by considering video recordings and their semantical descriptors which can be seen as temporals instances. A `Video` extraction outputs the images satisfying a given property, in the order of the input video. For example, if the video is a TV news recording, one could be interested in extracting the images related to a given topic (e.g. politics, economics).

Quite similarly to videos, musical scores or descriptors of musical digital files can also be seen as temporal instances.

(2) `Musical canon` with v voices consists in repeating a main melody v times, such that the first repetition starts with a time offset o , and each repetition also starts after the previous one with time offset o . One could be interested in building automatically canons, using a t2t query with parameters v and o .

(3) `Musical transposition` is meant to transpose k semi-tones higher or lower, each note of a score.

Aside from video and music processing, another field of application is streaming [2, 9]: a stream is a potentially infinite temporal instance.

(4) `Stream filtering`. Suppose that several traffic control devices send continuously data to a central server. One could want the server to filter the stream and build a new stream, made of all the data satisfying some property P .

The t2t languages introduced in this article are de-

¹A preliminary version of this work appeared in the informal proceedings of the French national conference on databases BDA 2006.

signed using known standard temporal query languages as elementary building blocks. For instance, we will consider two languages to illustrate our point: first order logic FO, and first order temporal logic TL. However, the results presented are stated for any temporal query language (e.g. FO, TL, ETL [17, 12], \forall TL [12], μ TL [16, 5], etc.).

We investigate two ways of defining t2t languages. The first one, denoted point-wise, exploits a simple and naive generalization of temporal query languages: given a query of a standard temporal language, the idea is to apply the query at all time points of a temporal instance (rather than at its first state). The t2t languages obtained that way are quite limited because they cannot have an impact over the length of the output instances. Thus, another mechanism called slicing-querying is explored for building t2t languages. Slicing-querying languages, called SQTL languages, are based on a two-phase process: the first step slices temporal instances into sub-instances; the second step is a query applied on each slice or sub-instance.

We investigate two mechanisms for slicing temporal instances. Cut slicing consists in gathering in a slice the consecutive states of a temporal instance as long as they return the same answer to some query. Cut slicing outputs a sequence of slices that do not overlap and that covers the input temporal instance. Regular slicing is based on a regular language, that is, given an input temporal instance a regular slicing tries to match subsequences of states of the temporal instance with words of the regular language. Regular slicing is designed in a way similar to the temporal language ETL [17], although they are quite different. We show that regular slicing subsumes cut slicing in most cases.

Let us now turn to an intuitive presentation of the querying phase. The input of the querying phase is the input temporal instance enriched by the output of the slicing phase. The querying process considers a temporal query q which is sequentially applied, for each slice, over the input temporal instance. It should be understood that, for each slice, the querying process outputs a temporal instance. Two kinds of querying are introduced and investigated: for each slice, either the query q is evaluated at the first time point of the slice and thus outputs a (static) relation or the query q is evaluated at each time point of the slice (in a local point-wise manner) and thus outputs a temporal instance.

The paper is organized as follows. Section 2 gives the notations we use throughout the article, and recalls the definitions of the temporal query language TL. Section 3 is devoted to the point-wise languages and gives their properties. Section 4 introduces SQTL languages, and their presentation is geared by some running examples. Section 5 is a discussion of some of the choices made in the design of slicing-querying languages, and

gives some concluding remarks.

2 Preliminaries

We assume the reader to be familiar with first-order logic FO and with the usual definitions of *relation schema*, *database schema* and *instances*. We match boolean relation schemas (resp. boolean instances) with relation schemas (resp. instances) of arity 0. Thus, the empty instance of arity 0 is denoted *false* and the instance of arity 0 containing the empty tuple is denoted *true*. In the whole paper, we assume a unique domain, and we consider the database schema \mathcal{R} .

An implicit temporal instance \mathbb{I} over the database schema \mathcal{R} is a finite sequence I_1, \dots, I_n of finite instances over \mathcal{R} . The size of this temporal instance \mathbb{I} , denoted $|\mathbb{I}|$ is the size of the sequence I_1, \dots, I_n , that is n . A temporal instance of size 1 is called a *static* instance. A temporal instance of size at least 2 is said to be *strictly temporal*. For each $i \in [1, n]$, I_i is called the state of \mathbb{I} at time point i . The instance $I_i(\mathcal{R})$ of the relation schema \mathcal{R} at time point i is also denoted $\mathbb{I}(\mathcal{R})[i]$. The active domain of \mathbb{I} , denoted $adom(\mathbb{I})$, is the set of domain elements appearing in \mathbb{I} . In the paper, \vec{x} represents a tuple of variables whose arity is clear from the context, and ν is a valuation of \vec{x} ranging over $adom(\mathbb{I})$.

If $\mathbb{I}' = I_f, \dots, I_\ell$ is a subsequence of states of \mathbb{I} , we denote $first-ind(\mathbb{I}') = f$ and $last-ind(\mathbb{I}') = \ell$.

The query language TL The first order linear temporal logic TL [8] is a well known formalism for specifying queries over the implicit temporal databases [7]. The syntax of TL over database 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 temporal instance \mathbb{I} at time point $i \in [1, n]$, given a valuation ν of the free variables of φ , denoted $[\mathbb{I}, i, \nu] \models \varphi$, is defined as follows:

- If φ is $R(\vec{x})$, $[\mathbb{I}, i, \nu] \models \varphi$ iff $(\nu(\vec{x})) \in \mathbb{I}(\mathcal{R})[i]$.
- If φ is obtained by a first order rule (negation, conjunction, or quantification), $[\mathbb{I}, i, \nu] \models \varphi$ is defined as usual.
- If φ is φ_1 *until* φ_2 , $[\mathbb{I}, i, \nu] \models \varphi$ iff there exists $j > i$ such that $[\mathbb{I}, j, \nu] \models \varphi_2$ and for all k such that $i < k < j$, $[\mathbb{I}, k, \nu] \models \varphi_1$.
- If φ is φ_1 *since* φ_2 , $[\mathbb{I}, i, \nu] \models \varphi$ iff there exists $j < i$ such that $[\mathbb{I}, j, \nu] \models \varphi_2$ and for all k such that $i > k > j$, $[\mathbb{I}, k, \nu] \models \varphi_1$.

It is sometimes convenient to use the following derived temporal modalities and formulas:

- $\mathcal{F}(\varphi)$ is *true until* φ and $\mathcal{P}(\varphi)$ is *true since* φ
- $next(\varphi)$ is *false until* φ and $prev(\varphi)$ is

false since φ

- *first* is $\neg \text{prev}(\text{true})$ and *last* is $\neg \text{next}(\text{true})$
- *some*(φ) is $\mathcal{F}(\varphi) \vee \mathcal{P}(\varphi) \vee \varphi$ (“sometime φ ”)
- *always*(φ) is $\neg \text{some}(\neg \varphi)$ (“always φ ”)

EXAMPLE 2.1 Let R be a unary schema in \mathcal{R} and let $\mathbb{I} = I_1, \dots, I_n$ be a temporal instance over \mathcal{R} . The boolean TL formula φ given below checks whether there is a state I_i (with $i \geq 2$) in \mathbb{I} such that I_i is identical to the first state I_1 .

$$\varphi = \mathcal{F}(\forall x [R(x) \longleftrightarrow \mathcal{P}(\text{first} \wedge R(x))])$$

Recall that the query “are there two states I_i and I_j that are identical?”, denoted *twin*, is not expressible in TL [1, 3].

Note that several temporal query languages more expressive than TL have been investigated, like Extended Temporal Logic ETL [12], the while-loop language T-WHILE [12, 5], and the fixpoint language μTL [5]. Note that ETL and μTL were introduced in the propositional case, respectively by Wolper [17] and Vardi [16].

Queries and answers A FO or TL query q over the database schema \mathcal{R} is specified by a formula $\varphi(\vec{x})$. The *evaluation* of q over temporal instance \mathbb{I} at time point i , denoted $q(\mathbb{I})[i]$, is obtained by evaluating φ at time point i : $q(\mathbb{I})[i] = \{\nu(\vec{x}) \mid [\mathbb{I}, i, \nu] \models \varphi(\vec{x}), \nu \text{ a valuation}\}$.

The *answer* of q over \mathbb{I} , denoted $q(\mathbb{I})$, is the evaluation of q at time point 1: $q(\mathbb{I}) = q(\mathbb{I})[1]$.

The reader should pay attention to the fact that the answer to a FO or a TL query is, by definition, a relation that is a static instance.

Two queries q_1 and q_2 are *equivalent*, denoted $q_1 \equiv q_2$, iff for all temporal instance \mathbb{I} we have $q_1(\mathbb{I}) = q_2(\mathbb{I})$.

Let q be a query of a temporal language over the database schema \mathcal{R} . We denote *insch*(q) the *input schema* of q , that is \mathcal{R} , and we denote *outsch*(q) the *output schema* of q .

Let \mathcal{L} be a language. Let q_1 and q_2 be queries of \mathcal{L} such that *insch*(q_2) = *outsch*(q_1). The *composition* of q_1 and q_2 is the query denoted $q_1 \circ q_2$ such that for all temporal instance \mathbb{I} , $(q_1 \circ q_2)(\mathbb{I}) = q_2(q_1(\mathbb{I}))$. The language \mathcal{L} is *closed under composition* iff any composition of queries of \mathcal{L} is expressible in \mathcal{L} .

Temporal and temporal-to-temporal languages In order to distinguish known query languages such as TL (whose queries output static instances) from the languages presented in this article (whose queries output temporal instances), we keep the usual terminology, *temporal language*, for the former and we introduce the terminology *temporal-to-temporal (t2t) language* for the latter. Note that the notions of input schema,

output schema and closure under composition are the same as above for t2t queries and t2t languages.

We now proceed to the definition of a first class of t2t languages, called point-wise languages.

3 Point-wise languages

In a first attempt to define t2t languages, a simple and naive idea is to apply a query of a given temporal language at all time points of a temporal instance. As we will see, the t2t languages thus obtained have important limitations. We now give a formal definition of point-wise languages.

If \mathcal{L} is a temporal language then $\text{PW}(\mathcal{L})$ is a point-wise language. The syntax of $\text{PW}(\mathcal{L})$ is the same as \mathcal{L} . A *query* of $\text{PW}(\mathcal{L})$ is an expression of the form $pw(q)$ where q is a query of \mathcal{L} . The *answer* of $pw(q)$ over the temporal instance $\mathbb{I} = I_1, \dots, I_n$, denoted $pw(q)(\mathbb{I})$, is the temporal instance $\mathbb{J} = J_1, \dots, J_n$ such that for all $i \in [1, n]$, J_i is the answer of q evaluated at time point i , that is $J_i = q(\mathbb{I})[i]$. Figure 1 illustrates how the answer to a point-wise query is obtained. However, the reader should not misinterpret the arrow between I_i and J_i : it represents that J_i is the result of the evaluation of q at time point i , which depends on the whole temporal instance I_1, \dots, I_n when \mathcal{L} is a temporal language like TL.

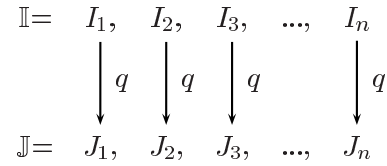


Figure 1. Evaluation of $pw(q)$ in $\text{PW}(\mathcal{L})$

We now give some examples of point-wise queries.

EXAMPLE 3.1 Transposition with $\text{PW}(\text{FO})$. We consider the musical transposition query, described in the introduction. A musical score is viewed here as a finite sequence of (sets of) notes. Let R be the unary relation schema. We consider temporal instances over $\mathcal{R} = \{R\}$ which are encoding of scores: in each state of the temporal instance, the instance over R stores notes.

Let k be an integer. The aim of the query transp_k is to compute a musical score where each note from the original score is transposed by k semitones. We assume that the function tr_k computes the k -transposition of notes.

The query transp_k is expressible in $\text{PW}(\text{FO})$. Let q be the following FO query:

$$q(x) = \exists y (R(y) \wedge \text{tr}_k(y) = x)$$

Then, it is easy to check that $pw(q)$ expresses $transp_k$.

EXAMPLE 3.2 Transposition revisited with $PW(TL)$. We use the same notations as in Example 3.1. We consider the t2t query denoted $cond-transp_k$ which aims at transposing k semi-tones higher only for notes N such that $tr_k(N)$ appears somewhere in the original score. The query $cond-transp_k$ is expressible in $PW(TL)$; let q_2 be the following TL query:

$$q_2(x) = [q(x) \wedge some(R(x))] \vee [\neg \exists z (q(z) \wedge some(R(z))) \wedge R(x)]$$

Above, q is the query from Example 3.1. Then, it is easy to check that $pw(q_2)$ expresses $cond-transp_k$.

EXAMPLE 3.3 Mirror. We consider the t2t query denoted *mirror*, specified by: for all temporal instance $\mathbb{I} = I_1, I_2, \dots, I_{n-1}, I_n$, $mirror(\mathbb{I}) = I_n, I_{n-1}, \dots, I_2, I_1$.

We claim that *mirror* is not expressible in $PW(TL)$, because if it was, we would be able to build a TL formula expressing the query *twin*. However, using a more expressive language like the fixpoint language μTL [12, 5], we can show that *mirror* is expressible in $PW(\mu TL)$.

Expressive power It is possible to obtain expressive power results for point-wise languages based on the hierarchy for temporal languages.

PROPERTY 3.1 Let \mathcal{L}_1 and \mathcal{L}_2 be temporal languages.

1. If $\mathcal{L}_1 \subseteq \mathcal{L}_2$ then $PW(\mathcal{L}_1) \subseteq PW(\mathcal{L}_2)$, and moreover,
2. If $\mathcal{L}_1 \subsetneq \mathcal{L}_2$ then $PW(\mathcal{L}_1) \subsetneq PW(\mathcal{L}_2)$.

The proof of this result is immediate from the definitions of the languages.

Point-wise versus temporal languages A temporal language \mathcal{L} can be seen as a t2t language: consider that its output is a temporal instance of size 1 (i.e. a static instance). Over static instances, it is trivial to show that \mathcal{L} and $PW(\mathcal{L})$ are equivalent. Over strictly temporal instances, \mathcal{L} and $PW(\mathcal{L})$ are incomparable: the answer to any query of \mathcal{L} (resp. $PW(\mathcal{L})$) cannot be an instance of size greater than 1 (resp. of size 1).

Properties We now show some properties of the point-wise languages. In the following, \mathcal{L} is any temporal language.

PROPERTY 3.2 $PW(\mathcal{L})$ is closed under composition.

Sketch of proof: let \mathbb{I} be a temporal instance over \mathcal{R} , and let q_1 and q_2 be queries of $PW(\mathcal{L})$ such that $q_1 \circ q_2$ is well defined. Let us assume that $insch(q_2) = outsch(q_1) = \{S\}$.

Since q_1 and q_2 are in $PW(\mathcal{L})$, there exists q'_1 and q'_2 in \mathcal{L} such that $q_1 = pw(q'_1)$ and $q_2 = pw(q'_2)$. Let q' be the query q'_2 where any occurrence of S is replaced by q'_1 , and let $q = pw(q')$. It is rather easy to see that $q(\mathbb{I}) = (q_1 \circ q_2)(\mathbb{I})$, hence $q \equiv q_1 \circ q_2$ \square

PROPERTY 3.3 For all temporal instance \mathbb{I} , for all $q \in \mathcal{L}$, $|pw(q)(\mathbb{I})| = |\mathbb{I}|$.

This property shows an important and obvious limitation of point-wise languages: they are unable to change the size of the temporal instance (i.e. if the size of the input is n then so is the size of the output). Indeed, none of the point-wise languages can express the video extraction or musical canon queries. This fact motivates the introduction of another paradigm for t2t queries.

Point-wise languages in the literature In [15], models for temporal queries are presented. The *sequenced queries* and the *non-sequenced queries* are respectively similar to point-wise queries with FO as the support language, to point-wise queries with a support language at least as expressive as TL. Although, because these languages deal with an interval based bidimensional representation of time, these queries are slightly more powerful than point-wise languages. However, the languages presented in [15] still suffer from being unable to define queries that increase the size of a temporal instance.

4 SCTL languages

In order to cope with the limitation of point-wise languages, we now consider temporal-to-temporal languages that work in two phases: the first one is called *slicing*, the second one is called *querying*.

Let us consider a temporal instance $\mathbb{I} = I_1, \dots, I_n$. Intuitively, the first step slices the temporal instance into sub-instances, called *slices*. The output of this first step is a sequence of slices. It can be such that, for instance, two slices share some common states (they may overlap), or/and such that a state of the temporal instance does not belong to any of the slices (the sequence of slices may have “holes”).

In the second step, a query q is applied over each slice, keeping available the access to the whole temporal instance \mathbb{I} as a context for q . The output of the second step is a temporal instance built by concatenating the results of querying the slices, in the order of the slicing output. We will investigate two ways to apply

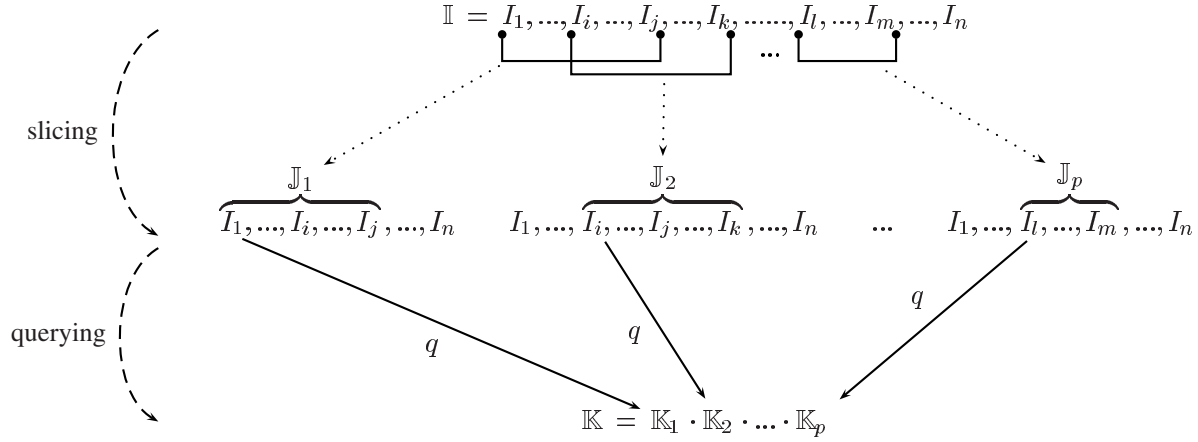


Figure 2. SQTl languages

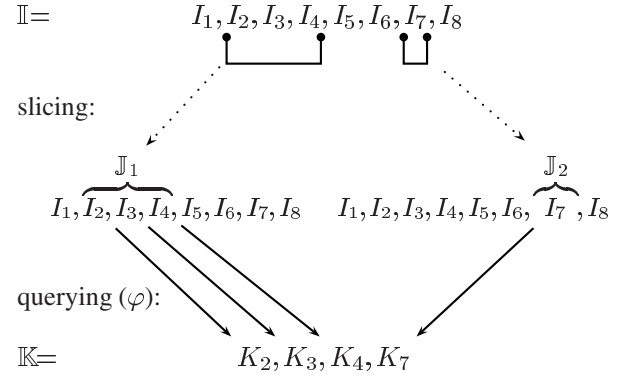
q over each slice : either q is applied once, in the first state of the slice, or q is applied in all the states of the slice, in a “local” point-wise manner.

Figure 2 illustrates the slicing and querying steps, in the case where q is applied only in the first state of every slice. The reader should pay attention to the fact that this figure and the subsequent ones are meant to illustrate the *semantics* of our slicing-querying languages, and not their physical implementation. For instance, the result of the slicing step is described by duplicating the temporal instance (once for each slice produced), but it is clear that such a replication is not needed for implementing the language.

Before defining further the slicing-querying languages, we illustrate the main ideas by some informal running examples.

EXAMPLE 4.1 Conditional point-wise query. Let $\varphi(x)$ be a query with free variable x . The aim of the query cpw is to compute a temporal instance whose states are the answers to φ iff this answer is not empty. For instance, suppose that $\mathbb{I} = I_1, \dots, I_8$ and that $K_i = \varphi(\mathbb{I})[i]$ is empty for $i \in \{1, 5, 6, 8\}$. Then $cpw(\mathbb{I}) = K_2, K_3, K_4, K_7$. Note that the video extraction and stream filtering applications given in the introduction are particular cases of conditional point-wise queries.

In order to express cpw by a SQTl query, we will first slice the input: a slice \mathbb{J}_i is a sub-instance of \mathbb{I} such that the answer to φ evaluated at any point of \mathbb{J}_i is non empty. Then, the querying phase applies the query φ in *each state* of each slice, in a point-wise manner. We illustrate this idea below:

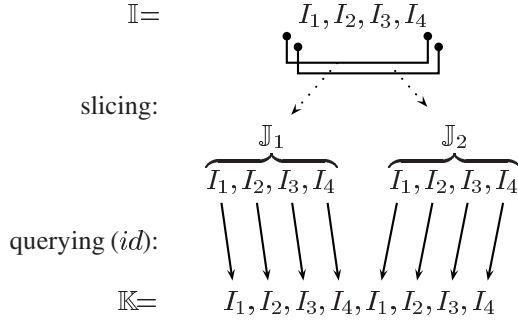


We would like to highlight the fact that if φ is a temporal query (whose evaluation at time point i depends on the whole temporal instance \mathbb{I}), then applying φ in the querying phase really computes $\varphi(\mathbb{I})[i]$ because \mathbb{I} is available at that moment.

EXAMPLE 4.2 Copy. The aim of the query $copy$ is to compute the temporal instance \mathbb{I} concatenated with itself. For instance, if $\mathbb{I} = I_1, I_2, I_3, I_4$, then

$$copy(\mathbb{I}) = I_1, I_2, I_3, I_4, I_1, I_2, I_3, I_4$$

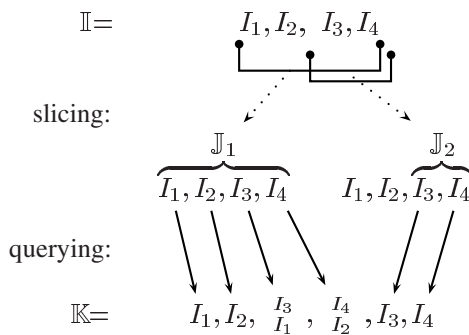
In order to express $copy$, we will slice all the states of \mathbb{I} twice, and in the querying phase we will apply the identity query id in each state of each slice. The figure below illustrates this.



EXAMPLE 4.3 Musical canon. Let v and o be strictly positive integers specifying respectively the number of voices and the offset of the canon. Considering a musical score, the aim of the query *canon* is to repeat the main melody v times, such that the second voice repeats the melody with a time offset o , and each next voice starts also after the previous one with time offset o . For instance, suppose that $\mathbb{I} = I_1, \dots, I_4$ is an encoding for a musical score. For all i, j , we denote I_{ij} the static instance $I_i \cup I_j$. Let $v=2$ and $o=2$. Then

$$\text{canon}(\mathbb{I}) = I_1, I_2, I_3, I_4, I_1, I_2, I_3, I_4$$

In order to express *canon* by a SCTL query, we will use v slices: the first one is \mathbb{I} itself and it will serve to output the first part of the melody for all voices (note that the first voice will play the entire melody thanks to this slice); the next slices are of size o and they serve to output the end of the melody played by each voice. We illustrate this idea below with $v=2$ and $o=2$ (recall that the arrow between an input state I_i and an output state is meant to capture that the output state is the answer of a temporal query evaluated at time point i over the whole temporal instance):



Slicing

A slicing process S is a function that maps temporal instances $\mathbb{I} = I_1, \dots, I_n$ to finite sequences of temporal instances. Let $S(\mathbb{I}) = \mathbb{J}_1, \dots, \mathbb{J}_p$. Each slice \mathbb{J}_j is a temporal instance I_i, \dots, I_{i+k} for some i and k . If $\mathbb{J}_j = I_f, \dots, I_\ell$ is a slice, we write $I_i \in \mathbb{J}_j$ when $i \in [f, \ell]$.

Slicing can be done in various manners. Here we focus on elementary slicing.

The slicing S is **elementary** iff $(\mathbb{I}$ and $S(\mathbb{I})$ are defined as above)

there exists $i_1, \dots, i_p \in [1, n]$ such that $i_1 < \dots < i_p$ and for all $j \in [1, p]$, $I_{i_j} \in \mathbb{J}_j$. Then the state I_{i_j} is called a *seed* for the slice \mathbb{J}_j .

For instance, let $\mathbb{I} = I_1, \dots, I_4$ and consider the slicing process S such that $S(\mathbb{I}) = \mathbb{J}_1, \mathbb{J}_2, \mathbb{J}_3$ where

$$\mathbb{J}_1 = I_1, I_2 \quad \mathbb{J}_2 = I_2 \quad \mathbb{J}_3 = I_2, I_3$$

Consider $i_1=1$, $i_2=2$ and $i_3=3$. We have $i_1 < i_2 < i_3$, and we also have $I_{i_1} \in \mathbb{J}_1$, $I_{i_2} \in \mathbb{J}_2$ and $I_{i_3} \in \mathbb{J}_3$. Thus for \mathbb{I} , S is elementary. Now consider the slicing S such that $S(\mathbb{I}) = \mathbb{J}_1, \mathbb{J}_2, \mathbb{J}_3, \mathbb{J}_4$, where $\mathbb{J}_4 = I_1, I_2, I_3$, and the other slices are defined as above. In this case, S is not elementary.

According to the definition, note that the elementary property enforces that $S(\mathbb{I})$ is a sequence of slices whose seeds are pair-wise distinct. Note also that, given a slice produced by an elementary slicing, there may be several time points as candidates to be the seed of this slice. Finally, it is important to outline that the elementary property entails that the number of slices in $S(\mathbb{I})$ is less or equal to the size of the temporal instance \mathbb{I} : if $\mathbb{I} = I_1, \dots, I_n$ and $S(\mathbb{I}) = \mathbb{J}_1, \dots, \mathbb{J}_p$, we always have $p \leq n$.

We choose to focus our study on elementary slicing processes for practical reasons. Intuitively, for a given slicing process, the property enables the computation of a slice to be done in one of its seeds. Since the number of seeds is less or equal to the size of the input temporal instance, it ensures that slicing the temporal instance is a reasonable process assuming that (1) computing the seeds of a slicing is efficient and that (2) the computation of one slice given its seed is computationally reasonable.

We now present two kinds of slicing. Both are elementary and although we do not provide the formal analysis, we claim that they have a reasonable complexity. Note that both slicings are based on a temporal language \mathcal{L} called *support*. Thus, different support languages can lead to different slicing. We use the notation $S(\mathcal{L})$ to denote the slicing process S with support \mathcal{L} .

Cut slicing

Let \mathcal{L} be a temporal language. We now define a slicing process called *cut with support* \mathcal{L} , denoted $\text{cut}(\mathcal{L})$.

Let q be a boolean query of \mathcal{L} . We denote $\text{cut}(q)$ the *cut slicing based on* q . The evaluation of $\text{cut}(q)$ over $\mathbb{I} = I_1, \dots, I_n$, denoted $\text{cut}(q)(\mathbb{I})$, is the sequence $\mathbb{J}_1, \dots, \mathbb{J}_p$ defined by:

- $I_1 \in \mathbb{J}_1$

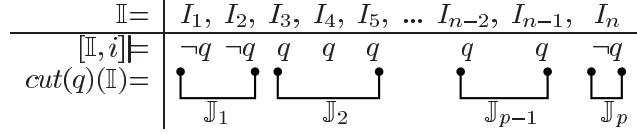


Figure 3. cut slicing

• Let $i \in [1, n-1]$ and $j \in [1, p-1]$. Suppose that $I_i \in \mathbb{J}_j$. If $[\mathbb{I}, i] \models q \iff [\mathbb{I}, i+1] \models q$ then $I_{i+1} \in \mathbb{J}_j$, otherwise $I_{i+1} \in \mathbb{J}_{j+1}$.

Intuitively, two consecutive states I_i and I_{i+1} of \mathbb{I} belong to the same slice iff I_i and I_{i+1} have the same answer for q in \mathbb{I} . For instance, consider the temporal instance \mathbb{I} of Figure 3 and let q be an arbitrary boolean query of \mathcal{L} . In the figure, we display q (resp. $\neg q$) under a state I_i iff $[\mathbb{I}, i] \models q$ (resp. $[\mathbb{I}, i] \models \neg q$). Thus we have $\text{cut}(q)(\mathbb{I}) = \mathbb{J}_1, \mathbb{J}_2, \dots, \mathbb{J}_{p-1}, \mathbb{J}_p$ where $\mathbb{J}_1 = I_1, I_2$; $\mathbb{J}_2 = I_3, I_4, I_5$; \dots ; $\mathbb{J}_{p-1} = I_{n-2}, I_{n-1}$; $\mathbb{J}_p = I_n$.

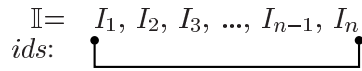
We now provide some examples of cut slicing.

EXAMPLE 4.4 Let R be a binary relation schema and let $\mathbb{I} = I_1, \dots, I_5$ be the following temporal instance over $\mathcal{R} = \{R\}$:

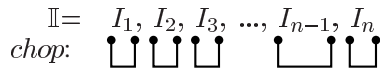
R	R	R	R	R
$\begin{array}{ c } \hline ab \\ \hline cc \\ \hline \end{array}$	$\begin{array}{ c } \hline de \\ \hline \\ \hline \end{array}$	$\begin{array}{ c } \hline ad \\ \hline \\ \hline \end{array}$	$\begin{array}{ c } \hline cd \\ \hline dd \\ \hline \end{array}$	$\begin{array}{ c } \hline bb \\ \hline \\ \hline \end{array}$
I_1	I_2	I_3	I_4	I_5

Let q be the FO query $q = \exists x R(x, x)$. Thus, we have $\text{cut}(q)(\mathbb{I}) = \mathbb{J}_1, \mathbb{J}_2, \mathbb{J}_3$ where $\mathbb{J}_1 = I_1$; $\mathbb{J}_2 = I_2, I_3$; $\mathbb{J}_3 = I_4, I_5$.

We now consider two particular slicings called identity slicing and chop. We call *identity slicing*, denoted ids , the slicing such that for all temporal instance \mathbb{I} , $\text{ids}(\mathbb{I})$ is reduced to one slice, itself: thus $\text{ids}(\mathbb{I}) = \mathbb{I}$. We illustrate the idea of ids below:



We denote *chop* the slicing S such that $S(\mathbb{I}) = \mathbb{J}_1, \dots, \mathbb{J}_n$ and for all $i \in [1, n]$, $\mathbb{J}_i = I_i$. We illustrate the idea of *chop* below:



EXAMPLE 4.5 Identity slicing and chop. The identity slicing ids can be expressed in $\text{cut}(\text{FO})$ by $\text{cut}(\text{true})$.

Intuitively, in order to express *chop* with cut, one needs a formula φ that evaluates to *true* every two states. Such a formula is expressible in μTL , because

this language can express that the size of the temporal instance is even. However, no such formula exists in TL (see Lemma 4.1).

A discussion about the cut and identity slicings is provided later on. Concerning *chop*, we have the following result.

LEMMA 4.1 *chop* is not expressible in $\text{cut}(\text{TL})$.

Sketch of proof: Suppose by absurd there exists $q \in \text{TL}$ such that $\text{cut}(q)$ is equivalent to *chop*. Then in particular, q is true in a state I_i iff q is false in I_{i+1} . Hence, q is true in I_i iff q is true in I_{i+2} , in I_{i+4} , in I_{i+6} and so on. Intuitively, we can express *even* by evaluating the following TL formula in the first state of the temporal instance:

$$q \longleftrightarrow \mathcal{F}(\text{last} \wedge \neg q)$$

This is in contradiction with the fact that *t-even* is not expressible in TL, where *t-even* is the boolean query which outputs true when the size (number of states) of the input is even. \square

Properties of cut Let $\text{cut}(q)$ be a cut slicing and let $\text{cut}(q)(\mathbb{I}) = \mathbb{J}_1, \dots, \mathbb{J}_p$.

PROPERTY 4.2 The sequence $\mathbb{J}_1, \dots, \mathbb{J}_p$ has:

- (1) *no hole*: for all $i \in [1, n]$, there exists at least one $j \in [1, p]$ such that $I_i \in \mathbb{J}_j$;
- (2) *no overlap*: for all $i \in [1, n]$, there exists at most one $j \in [1, p]$ such that $I_i \in \mathbb{J}_j$;

Note that Property 4.2 implies that any cut slicing is elementary. The computation of a cut slicing is reasonable in the sense that it simply requires to “scan” the whole temporal instance and evaluate a temporal query at each time point. The complexity of computing a cut slicing essentially depends on the complexity of the support language used.

Property 4.2 shows an important limitation of cut slicing. For instance, it is not possible to simulate the sliding windows used in streaming query languages (see [2] for a survey), because such windows are intuitively overlapping. In order to cope with the limitations of cut slicing, we introduce another slicing process.

Regular slicing

Regular slicing is based on the temporal query language ETL, introduced in the propositional case by Wolper [17] and extended to the first order case in [12]. Query evaluation in ETL tries to match words from a

regular language with the sequence of states of the temporal instance.

Let \mathcal{L} be a temporal language. We define a process called *regular slicing with support \mathcal{L}* , denoted $reg(\mathcal{L})$.

Let q_1, \dots, q_k be boolean queries of \mathcal{L} . Let $\mathcal{A} = \{a_1, \dots, a_k\}$ be an alphabet and let L be a regular language over \mathcal{A} . Intuitively, a letter a_j in the alphabet is associated with each formula q_j . We denote $reg_L(q_1, \dots, q_k)$ (or $reg(q_1, \dots, q_k)$ if there is no ambiguity) the *regular slicing based on q_1, \dots, q_k and L* . We will try to match words from L with subsequences of states in order to produce slices.

Let $\mathbb{I} = I_1, \dots, I_n$ be a temporal instance. For all $i \in [1, n]$, let $\mathcal{W}(i)$ be the following set of words:

$$\mathcal{W}(i) = \{w = v_{w_i} \dots v_{w_\ell} \mid w \in L \text{ and for all } j \in [i, \ell], [\mathbb{I}, j] \models q_{w_j}\}$$

Note that $\mathcal{W}(i)$ is a subset of L .

A *greatest element* of $\mathcal{W}(i)$ is a word of $\mathcal{W}(i)$ of maximal length.

The evaluation of $reg(q_1, \dots, q_k)$ over \mathbb{I} , denoted $reg(q_1, \dots, q_k)(\mathbb{I})$, is the sequence of slices $\mathbb{J}_1, \dots, \mathbb{J}_p$ obtained by applying the algorithm of Figure 4.

```

i:=1 ; j:=1 ;
While i≤n do
  If  $\mathcal{W}(i)$  is not empty
    Let  $w = v_{w_i} \dots v_{w_\ell}$  be a greatest element of  $\mathcal{W}(i)$  ;
     $\mathbb{J}_j := I_i, \dots, I_\ell$  ;
    j:=j+1 ;
  End if ;
  i:=i+1 ;
End while ;

```

Figure 4. Evaluation of $reg(q_1, \dots, q_k)$

Obviously, the algorithm entails that any regular slicing is elementary. Intuitively, \mathbb{I} is scanned from the left to the right. In each state I_i , we try to match words of L with the states to the right of I_i . If $\mathcal{W}(i)$ is empty then there is no match, and we proceed to the next state I_{i+1} . Otherwise, we consider the maximal length of all possible matches, i.e. the word $w = v_{w_i} \dots v_{w_\ell}$ from $\mathcal{W}(i)$, and we produce the slice I_i, \dots, I_ℓ .

We now provide some examples of regular slicing.

EXAMPLE 4.6 Let q_a, q_b and q_c be boolean queries from a temporal language \mathcal{L} . Let $\mathcal{A} = (a, b, c)$ and let L be the regular language $L = ab^* + cb$. Let $\mathbb{I} = I_1, \dots, I_8$. Below, we display q_j (resp. $\neg q_j$) under a state I_i iff $[\mathbb{I}, i] \models q_j$ (resp. $[\mathbb{I}, i] \models \neg q_j$) with $j \in \{a, b, c\}$:

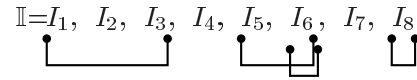
$\mathbb{I} =$	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
q_a	$\neg q_a$	q_a	$\neg q_a$	q_a	q_a	$\neg q_a$	q_a	
q_b	q_b	q_b	$\neg q_b$	q_b	q_b	$\neg q_b$	q_b	
$\neg q_c$	$\neg q_c$	q_c	$\neg q_c$	q_c	q_c	$\neg q_c$	q_c	

Consider $reg(q_a, q_b, q_c)$. We have $\mathcal{W}(1) = \{a, ab, abb\}$ because $[\mathbb{I}, 1] \models q_a$, $[\mathbb{I}, 2] \models q_b$ and $[\mathbb{I}, 3] \models q_b$. The word abb is a greatest element of $\mathcal{W}(1)$, thus the algorithm sets \mathbb{J}_1 to I_1, I_2, I_3 .

Since $\mathcal{W}(2), \mathcal{W}(3)$, and $\mathcal{W}(4)$ are empty, no slice is obtained for the states I_2, I_3 and I_4 .

We have $\mathcal{W}(5) = \{a, ab, cb\}$. The words ab and cb are greatest elements of $\mathcal{W}(5)$, but all that matters is their length: the algorithm sets \mathbb{J}_2 to I_5, I_6 in both cases.

We have $\mathcal{W}(6) = \{a\}$, thus the algorithm sets \mathbb{J}_3 to I_6 . We also have $\mathcal{W}(7) = \emptyset$ and $\mathcal{W}(8) = \{a\}$. Thus the algorithm sets \mathbb{J}_4 to I_8 . Finally, the sequence of slices obtained by $reg(q_a, q_b, q_c)(\mathbb{I})$ is:



EXAMPLE 4.7 Conditional point-wise query revisited. Let \mathcal{L} be a temporal language and $\varphi(x)$ be a query of \mathcal{L} with free variable x . In order to obtain the slicing of Example 4.1, let $\mathcal{A} = (a, b)$, $L = ab^*$ and consider the boolean formula $\psi = \exists x \varphi(x)$ and the slicing $reg_L(\neg prev(\psi) \wedge \psi, \psi)$. Intuitively, this regular slicing identifies sub-instances of maximal length such that ψ is true (the answer to φ is not empty) everywhere, and such that the first state of these sub-instances satisfies both $\neg prev(\psi)$ and ψ . In Example 4.1, the states I_2 and I_7 both satisfy $\neg prev(\psi)$.

EXAMPLE 4.8 Musical canon revisited with $reg(TL)$. In order to obtain the slicing of Example 4.3, let $\mathcal{A} = (a, b, c)$, let $L = ab^*c + bc$, and consider the boolean TL formulas $q_a = first$, $q_b = true$ and $q_c = last$. The slicing $reg_L(q_a, q_b, q_c)$ outputs two slices \mathbb{J}_1 and \mathbb{J}_2 , such that \mathbb{J}_1 is \mathbb{I} (i.e. all the states of \mathbb{I} , from the first to the last one) and \mathbb{J}_2 is the last two states of \mathbb{I} . Note that $reg_L(q_a, q_b, q_c)$ is not expressible in $reg(FO)$.

EXAMPLE 4.9 Chop revisited with $reg(FO)$. Consider the *chop* slicing from Example 4.5. This slicing is expressible in $reg(FO)$: let $\mathcal{A} = (a)$, $L = a$ and consider $reg_L(true)$.

Properties of regular slicing Let S be a regular slicing. Let $S(\mathbb{I}) = \mathbb{J}_1, \dots, \mathbb{J}_p$.

PROPERTY 4.3 The sequence $\mathbb{J}_1, \dots, \mathbb{J}_p$ is monotone, i.e. for all $j \in [1, p - 1]$, $first-ind(\mathbb{J}_j) < first-ind(\mathbb{J}_{j+1})$.

Note that Property 4.3 implies that any regular slicing is elementary. Like for cut slicing, the computation of regular slicing is reasonable in the sense that it simply requires to “scan” the whole temporal instance and

evaluate a temporal query a la ETL at each time point. Again, the complexity of computing a regular slicing essentially depends on the complexity of the support language used.

PROPERTY 4.4 If n denotes the size of \mathbb{I} , the sum of the sizes of the slices is less than n^2 : $(\sum_{j=1}^n |\mathbb{J}_j|) < n^2$.

Expressive power of *cut* versus *reg*

The notion of expressive power for standard query languages can be naturally extended to slicing languages. We have the following results.

THEOREM 4.5 If \mathcal{L} subsumes TL then $reg(\mathcal{L})$ is strictly more expressive than $cut(\mathcal{L})$.

Sketch of proof: let q be a boolean query of \mathcal{L} . We consider the slicing $cut(q)$ and we show that $cut(q)$ can be expressed by a regular slicing. Let $\mathcal{A}=(a,b,c,d)$ and let L be the regular language ab^*+cd^* . Consider the following boolean formulas: $q_a=q \wedge \neg prev(q)$, $q_b=q$, $q_c=\neg q \wedge \neg prev(\neg q)$, $q_d=\neg q$. Recall that \mathcal{L} subsumes TL, thus the formulas q_a, q_b, q_c and q_d are in \mathcal{L} . Then, the regular slicing $reg_L(q_a, q_b, q_c, q_d)$ is equivalent to $cut(q)$. Intuitively, $reg_L(q_a, q_b, q_c, q_d)$ identifies sub-instances of maximal length such that q (resp. $\neg q$) is true everywhere and such that their first state also satisfies $\neg prev(q)$ (resp. $\neg prev(\neg q)$).

We now show that the inclusion is strict. Intuitively, it suffices to show that $reg(\mathcal{L})$ may produce slice sequences with overlaps. This is impossible in $cut(\mathcal{L})$. Let $\mathcal{A}=(a)$ and consider the regular language $L=a^*$ and the boolean query $q=true$. Let us consider the regular slicing $reg_L(q)$. The evaluation of $reg_L(q)$ over the temporal instance $\mathbb{I}=I_1, \dots, I_n$ is the sequence of slices $\mathbb{J}_1, \dots, \mathbb{J}_n$ such that for all $i \in [1, n]$, $\mathbb{J}_i=I_i, \dots, I_n$. This entails that $I_n \in \mathbb{J}_i$ for all $i \in [1, n]$: thus according to Property 4.2, $reg_L(q)$ is not expressible in $cut(\mathcal{L})$. \square

As a consequence of Theorem 4.5, all the examples of slicing provided in this article for $cut(\mathcal{L})$ are also expressible in $reg(\mathcal{L})$.

Recall from Example 4.5 that the identity slicing *ids* can be expressed in $cut(\mathcal{L})$ for all \mathcal{L} by $cut(true)$. Note also that if \mathcal{L} subsumes TL, then *ids* can be expressed in $reg(\mathcal{L})$: consider the alphabet $\mathcal{A}=(a,b)$, the regular language $L=ab^*$ and the regular slicing $reg_L(first, true)$. However, if \mathcal{L} does not subsume TL, we have the following negative result.

LEMMA 4.6 The slicing language $reg(FO)$ can not express the identity slicing.

Sketch of proof: let $\mathbb{I}=I_1, \dots, I_n$ be a temporal instance. We consider a regular slicing q in $reg(FO)$. In order to show the lemma, we first need the following results.


1. For all FO formula φ , for all $i \in [1, n]$, $[\mathbb{I}, i] \models \varphi$ iff $I_i \models \varphi$. Intuitively, this means that the satisfaction of φ at time point i in \mathbb{I} only depends on the state at time point i . This result can be shown by induction on the formula φ .
2. For all $i \in [1, n]$, the set $\mathcal{W}(i)$ associated with q over \mathbb{I} is empty iff $\mathcal{W}(i)$ is empty over $\mathbb{I}_{\geq i}$, where $\mathbb{I}_{\geq i}$ denotes the temporal instance I_i, \dots, I_n . Intuitively, this means that $\mathcal{W}(i)$ does not depend on the states I_1, \dots, I_{i-1} . This is a consequence of item 1 above.
3. For all $i \in [2, n]$, the set $\mathcal{W}(i)$ is empty over \mathbb{I} iff $\mathcal{W}(i)$ is empty over $\mathbb{I}_{\geq 2}$. This is a direct consequence of item 2 above.

Suppose by absurd that q expresses the identity slicing *ids*. Consider $i \in [1, n]$. In the algorithm of figure 4, the first state of the slice obtained from $\mathcal{W}(i)$ is always I_i . Thus necessarily, the set $\mathcal{W}(1)$ associated with q over \mathbb{I} is not empty and for all $i \geq 2$, $\mathcal{W}(i)$ is empty over \mathbb{I} (otherwise, q cannot express *ids*). Then, item 3 entails that for all $i \geq 2$, $\mathcal{W}(i)$ is empty over $\mathbb{I}_{\geq 2}$. This entails that the evaluation of the regular slicing q over the temporal instance $\mathbb{I}_{\geq 2}$ is the empty sequence: a contradiction with the fact that q expresses *ids*. \square

THEOREM 4.7 The slicing languages $cut(FO)$ and $reg(FO)$ are incomparable.

Sketch of proof: on the one hand, Lemma 4.6 shows that there is a cut slicing in $cut(FO)$ which cannot be expressed in $reg(FO)$. On the other hand, the regular slicing given in the proof of Theorem 4.5 in order to show the strictness of the separation is in $reg(FO)$ but not in $cut(FO)$. \square

In order to overcome the problem raised by Lemma 4.6, one can consider the following extension: let us add the booleans *first* and *last* to FO, with the same semantics as in TL. Then *ids* is expressible in $reg(FO)$ with the regular slicing $reg_L(first, true)$ given above. In the rest of the article, we consider the above extension of FO. Although *ids* is now expressible in $reg(FO)$, we claim that Theorem 4.7 still holds, because the following slicing is not expressible in $reg(FO)$:

$$\mathbb{I} = I_1, I_2, I_3, \dots, I_{n-1}, I_n$$


Querying

Roughly, a querying process is a function that maps finite sequences of temporal instances to temporal in-

stances. We now define the querying phase more formally. Like for the slicing, we need to consider a temporal language \mathcal{L} called *support* of the querying process.

Given a query q of \mathcal{L} , two types of querying are introduced. Both take as input a temporal instance \mathbb{I} together with a slicing $\mathbb{J}_1, \dots, \mathbb{J}_p$ of \mathbb{I} , and output a temporal instance \mathbb{K} . Intuitively:

- the first kind of querying based on q , simply denoted q , consists in evaluating q over \mathbb{I} in the *first* state of each slice \mathbb{J}_j ;
- the second kind of querying based on q , denoted $pw(q)$, consists in evaluating q over \mathbb{I} in *each* state of each slice \mathbb{J}_j ;

Formally, we need to introduce two booleans which logically identify a slice \mathbb{J}_j inside the temporal instance \mathbb{I} . For all $j \in [1, p]$, we denote \mathbb{I}_j^* the temporal instance in which the boolean *first** (resp. *last**) is added in the first (resp. last) state of \mathbb{J}_j . Thus:

(1) Case of q . For each slice \mathbb{J}_j , q is evaluated over \mathbb{I}_j^* at time point $first-ind(\mathbb{J}_j)$. We denote $eval(q, \mathbb{I}_j^*)$ the static instance $q(\mathbb{I}_j^*)[first-ind(\mathbb{J}_j)]$.

(2) Case of $pw(q)$. For each slice \mathbb{J}_j , q is evaluated over \mathbb{I}_j^* at all time points between $first-ind(\mathbb{J}_j)$ and $last-ind(\mathbb{J}_j)$. We denote $eval(pw(q), \mathbb{I}_j^*)$ the temporal instance $q(\mathbb{I}_j^*)[first-ind(\mathbb{J}_j), \dots, last-ind(\mathbb{J}_j)]$.

Let δq denote either q or $pw(q)$. The *answer* of δq over \mathbb{I} and $\mathbb{J}_1, \dots, \mathbb{J}_p$, denoted $eval(\delta q, \mathbb{I}, (\mathbb{J}_1, \dots, \mathbb{J}_p))$, is the concatenation of the temporal instances $eval(\delta q, \mathbb{I}_1^*), eval(\delta q, \mathbb{I}_2^*), \dots, eval(\delta q, \mathbb{I}_p^*)$.

We now turn back to our running examples.

EXAMPLE 4.10 Conditional point-wise revisited. As mentioned in Example 4.1, we may apply $pw(\varphi)$ in the querying phase in order to output the temporal instance \mathbb{K} .

EXAMPLE 4.11 Musical canon revisited with TL. We consider once again the query *canon* described in Example 4.3. Given the slicing of Example 4.8, note that \mathbb{I}_1^* (resp. \mathbb{I}_2^*) is the temporal instance \mathbb{I} where *first** is *true* in I_1 (resp. I_3) and *last** is *true* in I_4 (resp. I_4). Intuitively, we would like to apply

$$q_1(x) = R(x) \vee prev(prev(R(x)))$$

over each state of the slice \mathbb{J}_1 and:

$$q_2(x) = R(x)$$

over each state of the slice \mathbb{J}_2 . Thus, let

$$q = [some(first \wedge first^*) \wedge q_1(x) \vee [\neg some(first \wedge first^*) \wedge q_2(x)]]$$

Then, $pw(q)$ outputs the temporal instance \mathbb{K} displayed in Example 4.3.

SQTL languages: summary

Now that the slicing and querying processes are formally defined, we combine them in order to define SQTL languages. Recall that any slicing process needs a (temporal) language as a support, as well as any querying process does. Language supports for slicing and for querying do not need to be the same. For instance, let \mathcal{L}_1 and \mathcal{L}_2 be temporal languages; $SQTL_{sl(\mathcal{L}_1)}^{\mathcal{L}_2}$ denotes the SQTL language built with a slicing sl (where sl can be either *cut* or *reg*) based on the language \mathcal{L}_1 and a querying based on the temporal language \mathcal{L}_2 .

For instance, $SQTL_{cut(TL)}^{FO}$ denotes the SQTL language such that its slicing component is *cut*(TL) and its querying component has support FO.

SQTL queries A query in $SQTL_{sl(\mathcal{L}_1)}^{\mathcal{L}_2}$ is a pair $(S, \delta q)$ where S is a slicing in $sl(\mathcal{L}_1)$ and δq is a querying in \mathcal{L}_2 i.e. δq is either q or $pw(q)$ for some $q \in \mathcal{L}_2$. Given a temporal instance \mathbb{I} , the *evaluation* of $(S, \delta q)$ over \mathbb{I} leads to the temporal instance $(S, \delta q)(\mathbb{I}) = eval(\delta q, \mathbb{I}, S(\mathbb{I}))$.

EXAMPLE 4.12 Musical canon revisited with $SQTL_{reg(TL)}^{TL}$. We consider the regular slicing $reg_L(q_a, q_b, q_c)$, from Example 4.8 (recall that q_a, q_b, q_c are TL formulas) and the TL query q , from Example 4.11. Then, $(reg_L(q_a, q_b, q_c), q)$ is a query in $SQTL_{reg(TL)}^{TL}$ which expresses the t2t query *canon* specified in Example 4.3.

The result below states that the SQTL languages defined in this article are proper extensions of both the temporal languages and the point-wise languages.

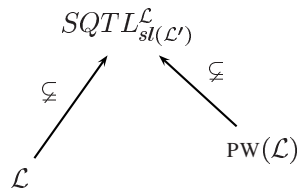
THEOREM 4.8 Let \mathcal{L} and \mathcal{L}' be two temporal languages. Let sl be either *cut* or *reg*. Over strictly temporal instances, the following holds:

1. $\mathcal{L} \subsetneq SQTL_{sl(\mathcal{L}')}^{\mathcal{L}}$, and
2. $PW(\mathcal{L}) \subsetneq SQTL_{sl(\mathcal{L}')}^{\mathcal{L}}$

Sketch of proof: The inclusions are straightforward. Let q be a query of \mathcal{L} . We show that there exists a slicing process S such that the SQTL query (S, q) (resp. $(S, pw(q))$) of $SQTL_{sl(\mathcal{L}')}^{\mathcal{L}}$ is equivalent to q (resp. $pw(q)$). Indeed, let S be a slicing process expressing the identity slicing. Recall that we made the assumption to use the extension of FO that adds the booleans *first* and *last*. Thus, the identity slicing is expressible in $sl(\mathcal{L}')$ with any support language \mathcal{L}' . Furthermore, it is easy to see that $(S, q)(\mathbb{I}) = q(\mathbb{I})$ and $(S, pw(q))(\mathbb{I}) = pw(q)(\mathbb{I})$.

The strictness of the inclusions is shown over strictly temporal instances. Indeed, on the one hand, the answer of any query of \mathcal{L} can only be a temporal instance of size 1 and there exists queries of $SQTL_{sl(\mathcal{L}')}^{\mathcal{L}}$ whose answer is a temporal instance of size greater than 1. On the other hand, the answer of any query of $PW(\mathcal{L})$ can only be a temporal instance of size greater than 1, and there exists queries of $SQTL_{sl(\mathcal{L}')}^{\mathcal{L}}$ whose answer is a temporal instance of size 1. \square

Recall that \mathcal{L} and $PW(\mathcal{L})$ are incomparable. Thus, Theorem 4.8 leads to the expressive power hierarchy displayed below:



5 Discussion

Restricted frameworks

In fact, we initially investigated a family of t2t languages already based on the two-phase paradigm of slicing-querying but with a restricted form of querying, further called local querying. Intuitively, the restriction concerns the scope of the query: for each slice, the query δq is “strictly” evaluated over the slice, disregarding the other states of the temporal instance. More formally, the restriction entails that, in the definition of the evaluation of a querying δq , given the slices $\mathbb{J}_1, \dots, \mathbb{J}_p$ of \mathbb{I} , we no more consider the temporal instances \mathbb{I}_j^* associated with \mathbb{J}_j , but \mathbb{J}_j itself, or, in other words, $\mathbb{I}_j^* = \mathbb{J}_j$.

The motivation for considering local querying is that it leads to more tractable t2t languages and that this restriction is a reasonable one for some class of applications. In the case of streaming [2, 9], where a stream is a potentially infinite temporal instance, it is of course not reasonable in practice to have a querying phase which requires the whole stream. Indeed, for streaming applications, in order to ensure that computations only need a finite amount of auxiliary memory, it turns out that the querying process is required to be local but moreover a similar restriction over the slicing process is required too: the slicing process needs to be local and bounded. For instance, the slicing process $cut(FO)$ is local in the sense that the FO query used in such a process is evaluated w.r.t. one state of the stream. However, even a slicing process as simple as $cut(FO)$ is not bounded in the sense that the size of each slice

cannot be bounded. Obviously, if the slicing process is not bounded then locality of the querying phase is not sufficient to reach the finite auxiliary memory requirement.

We are currently studying properties of t2t languages with local querying. This investigation focuses on comparative expressiveness and closure properties under composition and under nesting. Although the result obtained are not presented here, we would like to make a few comments about the impact of closure properties. The interest of studying closure under composition (resp. under nesting), for instance, is that it gives information about the impact of adding composition (resp. nesting) to a language \mathcal{L} on its expressive power. For complexity reason or for the sake of some applications, it may be desirable to choose language supports for slicing and querying having a weak expressiveness (for instance, choosing FO as the language support for *cut* may be required in order to preserve locality of slicing). This choice has an impact on the global expressive power of the t2t language and then adding composition (resp. nesting), when the language is not closed, may be a way to recover some expressiveness.

More operators

The SCTL languages presented in section 5 are of course not complete. There exists some t2t queries that cannot be expressed using these languages: the query *copy* of Example 4.2, despite the fact that it is a simple query (it can be expressed through an elementary slicing process combined with the identity querying - see Example 4.2), is not expressible in any SCTL language based on cut or regular slicing. Intuitively, this is because regular slicing is monotone (Property 4.3) and because regular slicing subsumes cut slicing in most cases (Theorem 4.5). This entails that we may need to study other slicing processes.

We believe that binary operators may be needed in order to increase the expressive power of t2t languages, depending on the applications. Indeed, concatenation is obviously one such operator. Consider two video descriptors, one containing speaker presentations about different topics and the other one containing documentaries on these topics. Both video descriptors include annotation, among which are topic identifiers (e.g. politics, economics). One may simply want to produce a single video by merging (shuffling) the presentations and the documentaries by topics. Assuming that the two video descriptors have been concatenated, we believe that it can be shown that shuffling them can be expressed if the support languages are as expressive as μTL . This observation raises questions similar to the ones concerning the closure properties and the impact of adding composition.

Finally, we would like to mention that providing a generic computational model for t2t queries through Relational temporal machines has been investigated in [4].

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] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 21st Conference on Principles of Databases Systems*, pages 1–16, 2002.
- [3] N. Bidoit, S. De Amo, and L. Segoufin. Order independent temporal properties. In *Journal of Logic and Computation*, 14(2):277–298, 2004.
- [4] N. Bidoit and F. Hantry. Relational Temporal Machines. In *Proceedings of the 14th International Symposium on Temporal Representation and Reasoning*, 2007.
- [5] N. Bidoit and M. Objois. Temporal query languages expressive power: μ TL vs. T-WHILE. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning*, pages 74–82, 2005.
- [6] A. K. Chandra and D. Harel. Structure and complexity 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] L. Golab, M. Tamer Özsu. Issues in data stream management. In *ACM SIGMOD Record*, 32(2):5–14, 2003.
- [10] 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.
- [11] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. In *Annals of Pure and Applied Logic*, 32:265–280, 1986.
- [12] L. Herr. Langages de Requête pour les Bases de Données Temporelles. *Ph.D thesis, Université Paris Sud*, 1997.
- [13] D. Leivant, Inductive definitions over finite structures. In *Information and Computation*, 89:95–108, 1990.
- [14] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*, North Holland, Amsterdam, 1974.
- [15] Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen and Andreas Steiner. Transitioning Temporal Support in TSQL2 to SQL3. In *Temporal Databases: Research and Practice*, O. Etzion, S. Jajodia, and S. Sripada (eds.), Springer, pp. 150–194, 1998
- [16] M. Y. Vardi. A temporal fixpoint calculus. In *Proceedings 5th ACM Symposium on Principles of Programming Languages*, pages 250–259, 1988.
- [17] P. Wolper. Temporal Logic Can Be More Expressive. In *Information and Control*, pages 72–99, 1983.