Formal Methods in System Design
(preprint)

Monitoring of Temporal First-order Properties with Aggregations

David Basin · Felix Klaedtke · Srdjan Marinovic · Eugen Zălinescu

Received: date / Accepted: date

Abstract In system monitoring, one is often interested in checking properties of aggregated data. Current policy monitoring approaches are limited in the kinds of aggregations they handle. To rectify this, we extend an expressive language, metric first-order temporal logic, with aggregation operators. Our extension is inspired by the aggregation operators common in database query languages like SQL. We provide a monitoring algorithm for this enriched policy specification language. We show that, in comparison to related data processing approaches, our language is better suited for expressing policies, and our monitoring algorithm has competitive performance.

Keywords Runtime Verification · Monitoring · System Compliance · Temporal Logic · Aggregation Operators

1 Introduction

Motivation. System monitoring is a wide-spread requirement for many kinds of systems, ranging from enterprise data centers to power grids. Both public and private companies are increasingly required to monitor whether their system usage complies with normative regulations. For example, US hospitals must follow the US Health Insurance Portability and Accountability Act (HIPAA) and financial services must conform to the Sarbanes-Oxley Act (SOX). First-order temporal

A preliminary version of this work has been presented at the 4th International Conference on Runtime Verification (RV 2013); see [7].

This work was partly done while the second author was at ETH Zurich.

D. Basin, S. Marinovic, and E. Zălinescu
Computer Science Department, Institute of Information Security, ETH Zurich, Universitätstr. 6, 8092 Zurich, Switzerland
E-mail: david.basin, srdan.marinovic, eugen.zalinescu @ inf.ethz.ch

F. Klaedtke
NEC Europe Ltd., Kurfürsten-Anlage 36, 69115 Heidelberg, Germany
E-mail: felix.klaedtke @ neclab.eu

logics are not only well-suited for formalizing such regulations, they also admit efficient monitoring. When used online, these monitors observe the actions of agents, such as users and their processes, and report violations. This can be in real-time, as the actions occur. Alternatively, the actions are logged and the monitor checks them later, such as during an audit. See, for example, [6, 19].

Current logic-based monitoring approaches are limited in their support for expressing and monitoring properties of aggregations. Such properties are often needed to express compliance policies, such as the following simple example from fraud prevention: A user must not withdraw more than $10,000 within a 31 day period from his credit card account. To formalize this policy, we need an operator to express the aggregation of the withdrawal amounts over the specified time window, grouped by the users. In this article, we address the problem of expressing and monitoring first-order temporal properties built from such aggregation operators.

Solution. First, we extend metric first-order temporal logic (MFOTL) with aggregation operators and with functions. This follows Hella et al.'s [20] extension of first-order logic with aggregations. We also ensure that the semantics of aggregations and grouping operations in our language mimics that of SQL. As an illustration, a formalization in our language of the above fraud-detection policy is

$$\Box \forall u.\, \forall s.\, [\mathsf{SUM}_a\, a.\, \blacklozenge_{[0,31)}\, withdraw(u, a)](s; u) \rightarrow s \preceq 10000\,. \tag{P0}$$

The SUM operator, at the current time point, groups all withdrawals for a user u over the past 31 days and sums up their amounts a. The aggregation formula defines a binary relation where the first coordinate is the SUM's result s and the second coordinate is the user u for whom the result is calculated. If the user's sum is greater than 10,000, then the policy is violated at the current time point. The formula (P0) therefore states that the aggregation condition must hold for each user and every time point.

For comparison, an SQL query for determining the violations with respect to the above policy at a specific time is

$$\mathsf{SELECT\ SUM}(a)\ \mathsf{AS}\ s, u\ \mathsf{FROM}\ W\ \mathsf{GROUP\ BY}\ u\ \mathsf{HAVING\ SUM}(a) > 10000\,.$$

Here W is the dynamically created view consisting of the withdrawals of all users within the 31 day time window relative to the given time. Note that the subscript a of the formula's aggregation operator in (P0) corresponds to a in the SQL query and the third appearance of a in (P0) is implicit in the query, as it is fixed by the view's definition. The second a in (P0) is redundant; its inclusion emphasizes that the variable a is bound, i.e., it does not correspond to a coordinate in the resulting relation.

Not all formulas in our language are monitorable. Unrestricted use of logic operators may require infinite relations to be built and manipulated. The second part of our solution, therefore, is a monitorable fragment of our language. It can express all our examples, which represent typical policy patterns, and it allows the liberal use of aggregations and functions. We extend our monitoring algorithm for MFOTL [8] to this fragment. In more detail, the algorithm processes log files sequentially and evaluates formulas in a bottom-up manner, using extended relational algebra operators to compute the evaluation of a formula from the evaluation of its direct subformulas. In particular, aggregation formulas are handled

as the homonymous relational algebra operators. Functions are handled similarly to Prolog, where variables are instantiated before functions are evaluated.

We have implemented our monitoring solution and we evaluate it, comparing it with the relational database management system PostgreSQL [23] and the stream-processing tool STREAM [2]. Our evaluation focuses on two aspects: the suitability of our proposed language for formalizing complex policies with aggregations (our examples are from the domain of fraud detection) and the performance of our prototype implementation. The results show that our language is better suited for specifying policies than SQL and our prototype's performance is superior to PostgreSQL's performance. This is because temporal reasoning must be explicitly encoded in SQL queries and PostgreSQL does not process logged data sequentially in time. STREAM's query language CQL [3] has limited support for temporal reasoning and several temporal constructs must be explicitly encoded, as is the case with SQL. It is thus less suited than our language for specifying the example policies. However, STREAM's performance is better than our tool's. Nevertheless, the performance of our prototype tool is still within the same order of magnitude as STREAM's performance and it is efficient enough for practical use.

Contributions. Although aggregations have appeared previously in monitoring, our language is the first to add expressive SQL-like aggregation operators to a first-order temporal language. This enables us to express complex compliance policies with aggregations. Our prototype implementation is therefore the first tool to handle such policies, and it does so with acceptable performance.

Related Work. Our MFOTL extension is inspired by the aggregation operators in database query languages like SQL and by Hella et al.'s extension of first-order logic with aggregation operators [20]. Hella et al.'s work is theoretically motivated: they investigate the expressiveness of such an extension in a non-temporal setting. A minor difference between their aggregation operators and ours is that their operators yield terms rather than formulas, as in our extension.

Monitoring algorithms for different variants of first-order temporal logics have been proposed by Hallé and Villemaire [19], Bauer at al. [10], and Basin et al. [8]. Except for the counting quantifier [10], none of them support aggregations. Bianculli et al. [11] present a policy language based on a first-order temporal logic with a restricted set of aggregation operators that can only be applied to atomic formulas. For monitoring, they require a fixed finite domain and provide a translation to a propositional temporal logic. Such a translation is not possible in our setting since variables range over an infinite domain. In the context of database triggers and integrity constraints, Sistla and Wolfson [24] describe an integration of aggregation operators into their monitoring algorithm for a first-order temporal logic. Their aggregation operators are different from those presented here in that they involve two formulas that select the time points to be considered for aggregation and they use a database query to select the values to be aggregated from the selected time points.

Other monitoring approaches that support aggregations are LarvaStat [14], LOLA [16], EAGLE [4], and an approach based on algebraic alternating automata [17]. These approaches allow one to aggregate over the events in system traces, where events are either propositions or parametrized propositions. They do not support grouping, which is needed to obtain statistics per group of events,

e.g., the events generated by the same agent. Moreover, quantification over data elements and correlating data elements is more restrictive in these approaches than in a first-order setting.

Most data stream management systems like STREAM [2] and Gigascope [15] handle SQL-like aggregation operators. For example, in STREAM's query language CQL [3] one selects events in a specified time range, relative to the current position in the stream, into a table that one aggregates. The temporal expressiveness of such languages is weaker than our language; in particular, linear-time temporal operators are not supported.

Organization. The remainder of the article is structured as follows. In Section 2, we extend MFOTL with aggregation operators. In Section 3, we present our monitoring algorithm, which we evaluate in Section 4. In Section 5, we draw conclusions.

2 MFOTL with Aggregation Operators

2.1 Preliminaries

We use standard notation for sets and set operations. We also use set notation with sequences. For instance, for a set A and a sequence $\bar{s} = (s_1, \ldots, s_n)$, we write $A \cup \bar{s}$ for the union $A \cup \{s_i \mid 1 \leq i \leq n\}$ and we denote the length of $\bar{s}$ by $|\bar{s}|$. Let $\mathbb{I}$ be the set of nonempty intervals over $\mathbb{N}$. We often write an interval in $\mathbb{I}$ as $[b, b') := \{a \in \mathbb{N} \mid b \leq a < b'\}$, where $b \in \mathbb{N}$, $b' \in \mathbb{N} \cup \{\infty\}$, and $b < b'$.

A *multi-set* M with domain D is a function $M : D \to \mathbb{N} \cup \{\infty\}$. This definition extends the standard one to multi-sets where elements can have an infinite multiplicity. A multi-set M is *finite* if $M(a) \in \mathbb{N}$ for every $a \in D$ and the set $\{a \in D \mid M(a) > 0\}$ is finite. We use the brackets $\{\!|$ and $|\!\}$ to specify multi-sets. For instance, $\{\!|2 \cdot \lfloor n/2 \rfloor \mid n \in \mathbb{N}|\!\}$ denotes the multi-set $M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ with $M(n) = 2$ if n is even and $M(n) = 0$ otherwise. A multi-set M is *empty* if $M(a) = 0$ for any $a \in D$. We denote the empty multi-set by $\emptyset$.

Given a domain D, an *aggregation operator* is a function from multi-sets with domain D to $D \cup \{\bot_\infty\}$ such that finite multi-sets are mapped to elements of $D \setminus \{\bot_\infty\}$ and infinite multi-sets are mapped to $\bot_\infty$. Common aggregations operators on finite non-empty multi-sets M that only contain rational numbers are:

$$\begin{aligned}
\mathsf{CNT}(M) &:= \sum_{a \in D} M(a)\,, \\
\mathsf{SUM}(M) &:= \sum_{a \in D} M(a) \cdot a\,, \\
\mathsf{MIN}(M) &:= \min\{a \in D \mid M(a) > 0\}\,, \\
\mathsf{MAX}(M) &:= \max\{a \in D \mid M(a) > 0\}\,,
\end{aligned}$$

and

$$\mathsf{AVG}(M) := \mathsf{SUM}(M)/\mathsf{CNT}(M)\,.$$

On the empty multi-set, the definition of the aggregations operators CNT and SUM is straightforward, namely, $\mathsf{CNT}(\emptyset) := \mathsf{SUM}(\emptyset) := 0$. However, the definition

of the other aggregations operators MIN, MAX, and AVG on the empty multi-set is less standard. For example, the average over an empty multi-set is undefined. We can define $\mathsf{AVG}(\emptyset) := \bot$, where $\bot$ is a special domain element representing undefinedness. Analogously, we can define $\mathsf{MIN}(\emptyset)$ and $\mathsf{MAX}(\emptyset)$ as $\bot$, or we can assume special domain elements ∞ and $-\infty$ and define $\mathsf{MIN}(\emptyset)$ and $\mathsf{MAX}(\emptyset)$ as ∞ and $-\infty$, respectively. Note that when $\bot$, ∞, and $-\infty$ are domain elements, one must extend the above definitions to finite, non-empty multi-sets that contain such elements. This can, for example, be done by ignoring such elements and their multiplicity. For readability, we omit a definition of these aggregation operators on such "ill-formed" multi-sets. These definitions are not relevant for the results of this article.

2.2 Syntax

A *signature* $\mathcal{S}$ is a tuple $(\mathsf{F}, \mathsf{R}, \iota)$, where F is a finite set of function symbols, R is a finite set of predicate symbols disjoint from F, and the function $\iota : \mathsf{F} \cup \mathsf{R} \to \mathbb{N}$ assigns to each symbol $s \in \mathsf{F} \cup \mathsf{R}$ an arity $\iota(s)$. In the following, let $\mathcal{S} = (\mathsf{F}, \mathsf{R}, \iota)$ be a signature and V a countably infinite set of variables, where $\mathsf{V} \cap (\mathsf{F} \cup \mathsf{R}) = \emptyset$.

Function symbols of arity 0 are called *constants*. Let $\mathsf{C} \subseteq \mathsf{F}$ be the set of constants of $\mathcal{S}$. *Terms* over $\mathcal{S}$ are defined inductively: Constants and variables are terms, and $f(t_1, \ldots, t_n)$ is a term if $t_1, \ldots, t_n$ are terms and f is a function symbol of arity $n > 0$. We denote by $\mathit{fv}(t)$ the set of the variables that occur in the term t. We denote by T the set of all terms over $\mathcal{S}$, and by $\mathsf{T}_\emptyset$ the set of ground terms, that is, terms without variables. A *substitution* θ is a function from variables to terms. We use the same symbol θ to denote its homomorphic extension to terms.

Given a finite set Ω of aggregation operators, the MFOTL$_\Omega$ *formulas* over the signature $\mathcal{S}$ are given by the grammar

$$\varphi ::= r(t_1, \ldots, t_{\iota(r)}) \mid (\neg\varphi) \mid (\varphi \vee \varphi) \mid (\exists x.\, \varphi) \mid (\bullet_I\, \varphi) \mid (\varphi\ \mathsf{S}_I\ \psi) \mid [\omega_t\, \bar{z}.\, \varphi](y; \bar{g})\,,$$

where r, t and the t_is, I, and ω range over the elements in R, T, $\mathbb{I}$, and Ω, respectively, x and y range over elements in V, and $\bar{z}$ and $\bar{g}$ range over sequences of elements in V. Note that we overload notation: ω denotes both an aggregation operator and its corresponding symbol. This grammar extends MFOTL's grammar [21] in two ways. First, it introduces aggregation operators. Second, terms may also be built from function symbols and not just from variables and constants. For ease of exposition, we do not consider future-time temporal operators.

We call $[\omega_t\, \bar{z}.\, \psi](y; \bar{g})$ an *aggregation formula*. It is inspired by the homonymous relational algebra operator. Intuitively, by viewing variables as (relation) attributes, $\bar{g}$ are the attributes on which grouping is performed, t is the term on which the aggregation operator ω is applied, and y is the attribute that stores the result. The variables in $\bar{z}$ are ψ's attributes that do not appear in the described relation. We define the semantics in Section 2.3, where we also provide examples.

The set of *free variables* of a formula φ, denoted $\mathit{fv}(\varphi)$, is defined as expected for the standard logic connectives. For an aggregation formula, it is defined as $\mathit{fv}\big([\omega_t\, \bar{z}.\, \varphi](y; \bar{g})\big) := \{y\} \cup \bar{g}$. A variable is *bound* if it is not free. We denote by $\bar{\mathit{fv}}(\varphi)$ the sequence of free variables of a formula φ that is obtained by ordering the free variables of φ by their occurrence when reading the formula from left to right. A formula is *well-formed* if, for each of its subformulas $[\omega_t\, \bar{z}.\, \psi](y; \bar{g})$, it holds that

(a) $y \notin \bar{g}$, (b) $fv(t) \subseteq fv(\psi)$, (c) the elements of $\bar{z}$ and $\bar{g}$ are pairwise distinct, and (d) $\bar{z} = fv(\psi) \setminus \bar{g}$. Note that, given condition (d), the use of one of the sequences $\bar{z}$ and $\bar{g}$ is redundant. However, we use this syntax to make the free and bound variables explicit in aggregation formulas. Throughout this article, we consider only well-formed formulas.

To omit parentheses, we assume that Boolean connectives bind stronger than temporal connectives, and unary connectives bind stronger than binary ones, except for the quantifiers, which bind weaker than Boolean ones. As syntactic sugar, we use standard Boolean connectives such as $\varphi \wedge \psi := \neg(\neg\varphi \vee \neg\psi)$, the universal quantifier $\forall x.\, \varphi := \neg\exists x.\, \neg\varphi$, and the temporal operators $\varphi\, \mathsf{T}_I\, \psi := \neg(\neg\varphi\, \mathsf{S}_I\, \neg\psi)$, $\blacklozenge_I\, \varphi := \mathsf{t}\, \mathsf{S}_I\, \varphi$, and $\blacksquare_I\, \varphi := \mathsf{f}\, \mathsf{T}_I\, \varphi$, where $I \in \mathbb{I}$, $\mathsf{t} := p \vee \neg p$, and $\mathsf{f} := \neg\mathsf{t}$, for some predicate symbol p of arity 0, assuming without loss of generality that R contains such a symbol. Non-metric variants of the temporal operators are easily defined, for example, $\blacklozenge\, \varphi := \blacklozenge_{[0,\infty)}\, \varphi$.

2.3 Semantics

We distinguish between predicate symbols whose corresponding relations are *rigid* over time and those that are *flexible*, i.e., their interpretations can change over time. We denote by R_r and R_f the sets of rigid and flexible predicate symbols, where $\mathsf{R} = \mathsf{R}_r \cup \mathsf{R}_f$ with $\mathsf{R}_r \cap \mathsf{R}_f = \emptyset$. We assume that R_r contains the binary predicate symbols $\approx$ and $\prec$, which have their expected interpretation, namely, equality and ordering.

A *structure* $\mathcal{D}$ over the signature $\mathcal{S}$ consists of a domain $\mathbb{D} \neq \emptyset$ and interpretations $f^{\mathcal{D}} \in \mathbb{D}^{\iota(f)} \to \mathbb{D}$ and $r^{\mathcal{D}} \subseteq \mathbb{D}^{\iota(r)}$, for each $f \in \mathsf{F}$ and $r \in \mathsf{R}$. A *temporal structure* over the signature $\mathcal{S}$ is a pair $(\bar{\mathcal{D}}, \bar{\tau})$, where $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \dots)$ is a sequence of structures over $\mathcal{S}$ and $\bar{\tau} = (\tau_0, \tau_1, \dots)$ is a sequence of non-negative integers with the following properties.

1. The sequence $\bar{\tau}$ is monotonically increasing, that is, $\tau_i \leq \tau_{i+1}$, for all $i \geq 0$. Moreover, $\bar{\tau}$ makes progress, that is, for every $\tau \in \mathbb{N}$, there is some index $i \geq 0$ such that $\tau_i > \tau$.
2. All structures $\mathcal{D}_i$, with $i \geq 0$, have the same domain, denoted $\mathbb{D}$.
3. Function symbols and rigid predicate symbols have rigid interpretations, that is, $f^{\mathcal{D}_i} = f^{\mathcal{D}_{i+1}}$ and $p^{\mathcal{D}_i} = p^{\mathcal{D}_{i+1}}$, for all $f \in \mathsf{F}$, $p \in \mathsf{R}_r$, and $i \geq 0$. We also write $f^{\bar{\mathcal{D}}}$ and $p^{\bar{\mathcal{D}}}$ for $f^{\mathcal{D}_i}$ and $p^{\mathcal{D}_i}$, respectively.

We call the elements in the sequence $\bar{\tau}$ *timestamps* and the indices of the elements in the sequences $\bar{\mathcal{D}}$ and $\bar{\tau}$ *time points*.

A *valuation* is a mapping $v : \mathsf{V} \to \mathbb{D}$. For a valuation v, a variable sequence $\bar{x} = (x_1, \dots, x_n) \in \mathsf{V}^n$, and $\bar{d} = (d_1, \dots, d_n) \in \mathbb{D}^n$, we write $v[\bar{x} \mapsto \bar{d}]$ for the valuation that maps x_i to d_i, for $1 \leq i \leq n$, and the other variables' valuation is unaltered. We abuse notation by also applying a valuation v to terms. That is, given a structure $\mathcal{D}$, we extend v homomorphically to terms.

For the remainder of the article, we fix a countable domain $\mathbb{D}$ that contains the rational numbers $\mathbb{Q}$ and elements like $\bot_\infty$ and $\bot$. We only consider a single-sorted logic. One could alternatively have sorts for the different types of elements like data elements and the aggregations. Furthermore, we assume that function symbols are always interpreted by total functions. Partial functions like division

over scalar domains can be extended to total functions, e.g., by mapping elements outside the function's domain to $\bot$. Since the treatment of partial functions is not essential to our work, we treat $\bot$ as any other element of $\mathbb{D}$. Alternative treatments are possible, for example based on multi-valued logics [22].

Definition 1 Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal structure over the signature $\mathcal{S}$, with $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \dots)$ and $\bar{\tau} = (\tau_0, \tau_1, \dots)$, φ a formula over $\mathcal{S}$, v a valuation, and $i \in \mathbb{N}$. We define the relation $(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \varphi$ inductively as follows:

$$
\begin{array}{lll}
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models p(t_1, \dots, t_{\iota(r)}) & \text{iff} & \big(v(t_1), \dots, v(t_{\iota(r)})\big) \in p^{\mathcal{D}_i} \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \neg\psi & \text{iff} & (\bar{\mathcal{D}}, \bar{\tau}, v, i) \not\models \psi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \psi \vee \psi' & \text{iff} & (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \psi \text{ or } (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \psi' \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \exists x.\, \psi & \text{iff} & (\bar{\mathcal{D}}, \bar{\tau}, v[x \mapsto d], i) \models \psi, \text{ for some } d \in \mathbb{D} \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \bullet_I \psi & \text{iff} & i > 0,\ \tau_i - \tau_{i-1} \in I, \text{ and } (\bar{\mathcal{D}}, \bar{\tau}, v, i-1) \models \psi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \psi\ \mathsf{S}_I\ \psi' & \text{iff} & \text{for some } j \le i,\ \tau_i - \tau_j \in I,\ (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi', \\
 & & \text{and } (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \psi, \text{ for all } k \text{ with } j < k \le i \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models [\omega_t\, \bar{z}.\, \psi](y; \bar{g}) & \text{iff} & v(y) = \omega(M) \text{ and if } \bar{g} \neq \emptyset \text{ then } M \text{ is non-empty,}
\end{array}
$$

where $M : \mathbb{D} \to \mathbb{N} \cup \{\infty\}$ is the multi-set

$$\{\!\{ v[\bar{z} \mapsto \bar{d}](t) \mid (\bar{\mathcal{D}}, \bar{\tau}, v[\bar{z} \mapsto \bar{d}], i) \models \psi, \text{ for some } \bar{d} \in \mathbb{D}^{|\bar{z}|} \}\!\}.$$

Note that the semantics for the aggregation formula is independent of the order of the variables in the sequence $\bar{z}$.

For a temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$, a time point $i \in \mathbb{N}$, a formula φ, a valuation v, and a sequence $\bar{z}$ of variables with $\bar{z} \subseteq fv(\varphi)$, we define the set

$$[\![\varphi]\!]^{(\bar{\mathcal{D}}, \bar{\tau}, i)}_{\bar{z}, v} := \left\{ \bar{d} \in \mathbb{D}^{|\bar{z}|} \mid (\bar{\mathcal{D}}, \bar{\tau}, v[\bar{z} \mapsto \bar{d}], i) \models \varphi \right\}.$$

We drop the superscript when it is clear from the context. We drop the subscript when $\bar{z} = \bar{fv}(\varphi)$. In this case the valuation v is irrelevant and $[\![\varphi]\!]^{(\bar{\mathcal{D}}, \bar{\tau}, i)}$ denotes the set of satisfying elements of φ at time point i in $(\bar{\mathcal{D}}, \bar{\tau})$.

With this notation, we illustrate the semantics for aggregation formulas in the case where we aggregate over a variable. We use the same notation as in Definition 1. In particular, consider a formula $\varphi = [\omega_x\, \bar{z}.\, \psi](y; \bar{g})$, with $x \in \mathsf{V}$, and a valuation v. Note that v (and thus also $v[\bar{z} \mapsto \bar{d}]$) fixes the values of the variables in $\bar{g}$ because these are free in φ. The multi-set M is as follows. If $x \notin \bar{g}$, then $M(a) = |\{\bar{d} \in [\![\varphi]\!]_{\bar{z}, v} \mid d_j = a\}|$, for any $a \in \mathbb{D}$, where j is the index of x in $\bar{z}$. If $x \in \bar{g}$, then $M(v(x)) = |[\![\varphi]\!]_{\bar{z}, v}|$ and $M(a) = 0$, for any $a \in \mathbb{D} \setminus \{v(x)\}$.

Example 2 Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal structure over a signature with a ternary predicate symbol p, with $p^{\mathcal{D}_0} = \{(1, b, a), (2, b, a), (1, c, a), (4, c, b)\}$. Moreover, let φ be the formula $[\mathsf{SUM}_x\, x, y.\, p(x, y, g)](s; g)$ and $\bar{z} = (x, y)$. At time point 0, for a valuation v_1 with $v_1(g) = a$, we have $[\![p(x, y, g)]\!]_{\bar{z}, v_1} = \{(1, b), (2, b), (1, c)\}$ and $M = \{\!\{1, 2, 1\}\!\}$. For a valuation v_2 with $v_2(g) = b$, we have $[\![p(x, y, g)]\!]_{\bar{z}, v_2} = \{(4, c)\}$ and $M = \{\!\{4\}\!\}$. Finally, for a valuation v_3 with $v_3(g) \notin \{a, b\}$, we have that both $[\![p(x, y, g)]\!]_{\bar{z}, v_3}$ and M are empty. So the formula φ is only satisfied under a valuation v with $v(s) = 4$ and either $v(g) = a$ or $v(g) = b$. Indeed, we have $[\![\varphi]\!] = \{(4, a), (4, b)\}$. The tables in Figure 1 illustrate this example.

If we group on the variable x instead of g, we get $[\![[\mathsf{SUM}_x\, y, g.\, p(x, y, g)](s; x)]\!] = \{(2, 1), (2, 2), (4, 4)\}$, and $[\![[\mathsf{SUM}_x\, x, y, g.\, p(x, y, g)](s)]\!] = \{(8)\}$, if we do not group on

x	y	g
1	b	a
2	b	a
1	c	a
4	c	b

x	y	g
1	b	
2	b	a
1	c	
4	c	b

Fig. 1 Relation $p^{\mathcal{D}_0}$ from Example 2. The two boxes represent the multi-set M for the two valuations v_1 and v_2, respectively.

any variable. Finally, note that if the multi-set over which we aggregate is infinite, the aggregated value is $\perp_\infty$. For example, we have $[\![[\mathsf{SUM}_x\, x, y.\, \neg p(x,y,g)](s;g)]\!] = \mathbb{D} \times \{\perp_\infty\}$ and $[\![[\mathsf{SUM}_x\, x, y, g.\, \neg p(x,y,g)](s)]\!] = \{(\perp_\infty)\}$.

Example 3 This example illustrates the special case where aggregation operators are applied on formulas that have no satisfying elements. Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal structure over a signature with a binary predicate symbol q, with $q^{\mathcal{D}_0} = \emptyset$. We have $[\![[\omega_x\, x.\, q(x,y)](s;y)]\!] = \emptyset$, for any aggregation operator ω, while $[\![[\mathsf{SUM}_x\, x.\, q(x,y)](s)]\!] = \{(0)\}$ and $[\![[\mathsf{AVG}_x\, x.\, q(x,y)](a)]\!] = \{(\perp)\}$. Furthermore, if $\mathsf{MIN}(\emptyset)$ is defined as $\perp$ then $[\![[\mathsf{MIN}_x\, x.\, q(x,y)](m)]\!] = \{(\perp)\}$, while if it is defined as ∞, we obtain $[\![[\mathsf{MIN}_x\, x.\, q(x,y)](m)]\!] = \{(\infty)\}$ if $\infty \in \mathbb{D}$, and $[\![[\mathsf{MIN}_x\, x.\, q(x,y)](m)]\!] = \emptyset$, if $\infty \notin \mathbb{D}$.

The issue with the definition of aggregation operators on empty multi-sets, illustrated by Example 3, also appears in SQL. There, aggregation operators return the special domain element NULL on empty multi-sets.

Example 4 Consider the formula $\varphi = [\mathsf{SUM}_a\, a.\, \psi](s;u)$, where ψ is the formula $\blacklozenge_{[0,31)}\, \mathit{withdraw}(u,a)$. Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal structure with the relations $\mathit{withdraw}^{\mathcal{D}_0} = \{(\mathrm{Bob}, 9), (\mathrm{Bob}, 3)\}$ and $\mathit{withdraw}^{\mathcal{D}_1} = \{(\mathrm{Bob}, 3)\}$, and the timestamps $\tau_0 = 5$ and $\tau_1 = 8$. We have that $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},0)} = [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},1)} = \{(\mathrm{Bob}, 9), (\mathrm{Bob}, 3)\}$ and therefore $[\![\varphi]\!]^{(\bar{\mathcal{D}},\bar{\tau},0)} = [\![\varphi]\!]^{(\bar{\mathcal{D}},\bar{\tau},1)} = \{(12, \mathrm{Bob})\}$. Our semantics ignores the fact that the tuple $(\mathrm{Bob}, 3)$ occurs at both time points 0 and 1. Note that the withdraw events do not have unique identifiers in this example.

To account for multiple occurrences of an event, we can attach to each event additional information to make it unique. For example, assume we have a predicate symbol *ts* at hand that records the timestamp at each time point, i.e., $\mathit{ts}^{\mathcal{D}_i} = \{\tau_i\}$, for $i \in \mathbb{N}$. For the formula $\varphi' = [\mathsf{SUM}_a\, a.\, \psi'](s;u)$ with $\psi' = \blacklozenge_{[0,31)}\, \mathit{withdraw}(u,a) \wedge \mathit{ts}(\tau)$, we have that $[\![\varphi']\!]^{(\bar{\mathcal{D}},\bar{\tau},0)} = \{(12, \mathrm{Bob})\}$ and $[\![\varphi']\!]^{(\bar{\mathcal{D}},\bar{\tau},1)} = \{(15, \mathrm{Bob})\}$ because $[\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},0)} = \{(\mathrm{Bob}, 9, 5), (\mathrm{Bob}, 3, 5)\}$ while $[\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},1)} = \{(\mathrm{Bob}, 9, 5), (\mathrm{Bob}, 3, 5), (\mathrm{Bob}, 3, 8)\}$. To further distinguish between withdraw events at time points with equal timestamps, we would need additional information about the occurrence of an event, for example information obtained from a predicate symbol *tpts* that is interpreted as $\mathit{tpts}^{\mathcal{D}_i} = \{(i, \tau_i)\}$, for $i \in \mathbb{N}$.

The multiplicity issue illustrated by Example 4 also appears in databases. SQL is based on a multi-set semantics and one uses the DISTINCT keyword to switch to a set-based semantics. However, it is problematic to define a multi-set semantics for first-order logic that associates a tuple $\bar{d} \in \mathbb{D}^{|f_v(\varphi)|}$ with a multiplicity denoting how

often $\bar{d}$ satisfies the formula φ rather than a Boolean value. For instance, there are several ways to define a multi-set semantics for disjunction: the multiplicity of $\bar{d}$ for $\psi \vee \psi'$ can be either the maximum or the sum of the multiplicities of $\bar{d}$ for ψ and ψ'. Depending on the choice, standard logical laws become invalid, for example the distributivity of existential quantification or conjunction over disjunction. Defining a multi-set semantics for negation is even more problematic.

3 Monitoring Algorithm

In this section, we present our monitoring algorithm for MFOTL$_\Omega$. The algorithm is inspired by those in [8, 9, 12] and it is based on formulating the evaluation of formulas φ in a fragment of MFOTL$_\Omega$ in terms of extended relational algebra operators applied to the evaluation of the direct subformulas of φ. We start with an overview of our monitoring approach.

We assume that policies are of the form $\Box \forall \bar{x}.\, \varphi$, where φ is an MFOTL$_\Omega$ formula and $\bar{x}$ is the sequence of φ's free variables. The policy requires that $\forall \bar{x}.\, \varphi$ holds at every time point in the temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$. In the following, we assume that $(\bar{\mathcal{D}}, \bar{\tau})$ is a *temporal database*, i.e., (1) the domain $\mathbb{D}$ is countably infinite, (2) the relation $p^{\mathcal{D}_i}$ is finite, for each $p \in \mathsf{R}_f$ and $i \in \mathbb{N}$, (3) $p^{\bar{\mathcal{D}}}$ is a recursive relation, for each $p \in \mathsf{R}_r$, and (4) $f^{\bar{\mathcal{D}}}$ is computable, for each $f \in \mathsf{F}$. We also assume that the aggregation operators in Ω are computable functions on finite multi-sets.

The inputs of our monitoring algorithm are a formula ψ, which is logically equivalent to $\neg\varphi$, and a temporal database $(\bar{\mathcal{D}}, \bar{\tau})$, which is processed iteratively. The algorithm outputs, again iteratively, the relation $[\![\psi]\!]^{(\bar{\mathcal{D}}, \bar{\tau}, i)}$, for each $i \geq 0$. As ψ and $\neg\varphi$ are equivalent, the tuples in $[\![\psi]\!]^{(\bar{\mathcal{D}}, \bar{\tau}, i)}$ are the policy violations at time point i. Note that we drop the outermost quantifier as we are interested not only in whether the policy is violated. An instantiation of the free variables $\bar{x}$ that satisfies ψ provides additional information about the violations.

3.1 Monitorable Fragment

Not all formulas are effectively monitorable. Consider, for example, the policy formalization $\Box \forall x.\, \forall y.\, p(x) \rightarrow q(x, y)$, with the formula $\psi = p(x) \wedge \neg q(x, y)$ that we use for monitoring. There are infinitely many violations for time points i with $p^{\mathcal{D}_i} \neq \emptyset$, namely, any tuple $(a, b) \in \mathbb{D}^2 \backslash q^{\mathcal{D}_i}$ with $a \in p^{\mathcal{D}_i}$. In such a case, $[\![\psi]\!]^{(\bar{\mathcal{D}}, \bar{\tau}, i)}$ is infinite and its elements cannot be enumerated in finite time. We define a fragment of MFOTL$_\Omega$ that guarantees finiteness. Furthermore, the set of violations at each time point can be effectively computed bottom-up over the formula structure. In the following, we treat the Boolean connective $\wedge$ and the temporal operator T_I as primitives.

Definition 5 The set $\mathcal{F}$ of *monitorable formulas* with respect to $(H_p)_{p \in \mathsf{R}_r}$ is defined by the rules given in Figure 2, where $H_p \subseteq \{1, \ldots, \iota(p)\}$, for each $p \in \mathsf{R}_r$.

Let ℓ be a label of a rule from Figure 2. We say that a formula $\varphi \in \mathcal{F}$ is of *kind* ℓ if there is a derivation tree for φ having as its root a rule labeled by ℓ.

$$\frac{p \in \mathsf{R}_f \quad x_1, \ldots, x_{\iota(p)} \in \mathsf{V} \text{ are pairwise distinct}}{p(x_1, \ldots, x_{\iota(p)}) \in \mathcal{F}}\ \mathsf{FLX}$$

$$\frac{\varphi \in \mathcal{F} \quad p \in \mathsf{R}_r \quad \bigcup_{i=1}^{\iota(p)} fv(t_i) \subseteq fv(\varphi)}{\varphi \wedge p(t_1, \ldots, t_{\iota(p)}) \in \mathcal{F}}\ \mathsf{RIG}_{\wedge} \qquad \frac{\varphi \in \mathcal{F} \quad p \in \mathsf{R}_r \quad \bigcup_{i=1}^{\iota(p)} fv(t_i) \subseteq fv(\varphi)}{\varphi \wedge \neg p(t_1, \ldots, t_{\iota(p)}) \in \mathcal{F}}\ \mathsf{RIG}_{\wedge\neg}$$

$$\frac{\varphi \in \mathcal{F} \quad p \in \mathsf{R}_r \quad \bigcup_{i=1, i\neq j}^{\iota(p)} fv(t_i) \subseteq fv(\varphi) \quad t_j \in \mathsf{V} \quad j \in H_p}{\varphi \wedge p(t_1, \ldots, t_{\iota(p)}) \in \mathcal{F}}\ \mathsf{RIG}'_{\wedge}$$

$$\frac{\varphi, \psi \in \mathcal{F}}{\varphi \wedge \psi \in \mathcal{F}}\ \mathsf{GEN}_{\wedge} \qquad \frac{\varphi, \psi \in \mathcal{F} \quad fv(\psi) \subseteq fv(\varphi)}{\varphi \wedge \neg\psi \in \mathcal{F}}\ \mathsf{GEN}_{\wedge\neg} \qquad \frac{\varphi, \psi \in \mathcal{F} \quad fv(\psi) = fv(\varphi)}{\varphi \vee \psi \in \mathcal{F}}\ \mathsf{GEN}_{\vee}$$

$$\frac{\varphi \in \mathcal{F}}{\exists x.\, \varphi \in \mathcal{F}}\ \mathsf{GEN}_{\exists} \qquad \frac{\varphi \in \mathcal{F}}{\bullet_I \varphi \in \mathcal{F}}\ \mathsf{GEN}_{\bullet} \qquad \frac{\varphi \in \mathcal{F}}{[\omega_t\, \bar{z}.\, \varphi](y; \bar{g}) \in \mathcal{F}}\ \mathsf{GEN}_{\omega}$$

$$\frac{\varphi, \psi \in \mathcal{F} \quad fv(\varphi) \subseteq fv(\psi)}{\varphi\, \mathsf{S}_I\, \psi \in \mathcal{F}}\ \mathsf{GEN}_{\mathsf{S}} \qquad \frac{\varphi, \psi \in \mathcal{F} \quad fv(\varphi) \subseteq fv(\psi)}{\neg\varphi\, \mathsf{S}_I\, \psi \in \mathcal{F}}\ \mathsf{GEN}_{\neg\mathsf{S}}$$

$$\frac{\varphi, \psi \in \mathcal{F} \quad fv(\psi) \subseteq fv(\varphi)}{\varphi\, \mathsf{T}_I\, \psi \in \mathcal{F}}\ \mathsf{GEN}_{\mathsf{T}}$$

Fig. 2 The derivation rules defining the fragment $\mathcal{F}$ of monitorable formulas.

Before describing some of the rules, we first explain the meaning of the set H_p, for $p \in \mathsf{R}_r$ with arity k. The set H_p contains the indexes j for which we can determine the values of the variable x_j that satisfy $p(x_1, \ldots, x_k)$, given that the values of the variables x_i with $i \neq j$ are fixed. Formally, given a temporal database $(\bar{\mathcal{D}}, \bar{\tau})$ and a rigid predicate symbol p of arity $k > 0$, we say that an index j, with $1 \leq j \leq k$, is *effective* for p if for any $\bar{a} \in \mathbb{D}^{k-1}$, the set $\{d \in \mathbb{D} \mid (a_1, \ldots, a_{j-1}, d, a_j, \ldots, a_{k-1}) \in p^{\bar{\mathcal{D}}}\}$ is finite. For instance, for the rigid predicate $\approx$, the set of effective indexes is $H_{\approx} = \{1, 2\}$. Similarly, for the rigid predicate $\prec_{\mathbb{N}}$, defined as $a \prec_{\mathbb{N}} b$ iff $a, b \in \mathbb{N}$ and $a < b$, we have $H_{\prec_{\mathbb{N}}} := \{1\}$.

We describe the intuition behind the first four rules in Figure 2. The meaning of the other rules should then be obvious. The first rule (FLX) requires that in an atomic formula $p(t_1, \ldots, t_{\iota(p)})$ with $p \in \mathsf{R}_f$, the terms t_i are pairwise distinct variables. This formula is monitorable since we assume that p's interpretation is always a finite relation. For the rules ($\mathsf{RIG}_{\wedge}$) and ($\mathsf{RIG}_{\wedge\neg}$), consider formulas of the form $\varphi \wedge p(t_1, \ldots, t_{\iota(p)})$ and $\varphi \wedge \neg p(t_1, \ldots, t_{\iota(p)})$ with $p \in \mathsf{R}_r$ and $\bigcup_{i=1}^{\iota(p)} fv(t_i) \subseteq fv(\varphi)$. In both cases, the second conjunct further restricts the satisfying tuples of φ. An example is the formula $\varphi(x, y) \wedge x + 1 \approx y$. If φ is monitorable, the conjunction is also monitorable as it can be evaluated by removing from $[\![\varphi]\!]$ the tuples that do not satisfy the second conjunct $x + 1 \approx y$. The rule ($\mathsf{RIG}'_{\wedge}$) treats the case where one of the terms t_i is a variable that does not appear in φ. We require here that the index j is effective, so that the values of this variable are determined by the values of the other variables, which themselves are given by the tuples in $[\![\varphi]\!]$. An example is the formula $p(x, y) \wedge z \approx x + y$. The required conditions on t_j are necessary. If j is not effective, then we cannot guarantee finiteness. Consider, for example, the formula $q(x) \wedge x \prec y$. If we do not require that t_j is a variable, then we would have to solve equations to determine the value of the variable that does not occur in φ. Consider, for example, the formula $q(x) \wedge x \approx y \cdot y$.

The rule (FLX) may seem quite restrictive. However, one can often rewrite a formula of the form $p(t_1, \ldots, t_n)$ with $p \in \mathsf{R}_f$ into an equivalent formula in $\mathcal{F}$. For instance, $p(x+1, x)$ can be rewritten to $\exists y.\, p(y, x) \wedge x + 1 \approx y$. Alternatively, one can add additional rules that handle such cases directly.

The following lemma shows that φ's membership in $\mathcal{F}$ guarantees the finiteness of $[\![\varphi]\!]$. The proof consists of a straightforward induction on the formula structure.

Lemma 6 *Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal database, $i \in \mathbb{N}$ a time point, φ a formula, and H_p the set of effective indexes for p, for each $p \in \mathsf{R}_r$. If φ is a monitorable formula with respect to $(H_p)_{p \in \mathsf{R}_r}$, then $[\![\varphi]\!]^{(\bar{\mathcal{D}}, \bar{\tau}, i)}$ is finite.*

There are formulas like $(x \approx y)\, \mathsf{S}\, p(x, y)$ that describe finite relations but are not in $\mathcal{F}$. Finiteness can also be guaranteed by semantic notions like domain independence or syntactic notions like range restriction, see, for example, [1] and also [8,13] for a generalization of these notions to a temporal setting. If we restrict ourselves to MFOTL without future operators, the range restricted fragment in [8] is more general than the fragment $\mathcal{F}$. This is because, in contrast to the rules in Figure 2, range restrictions are not local conditions, that is, conditions that only relate formulas with their direct subformulas. However, the evaluation procedures in [1,8,13] also work in a bottom-up recursive manner. So one still must rewrite the formulas to evaluate them bottom-up. No rewriting is needed for formulas in $\mathcal{F}$. Furthermore, the fragment ensures that aggregation operators are always applied to *finite* multi-sets. Thus, for any $\varphi \in \mathcal{F}$, the element $\bot_\infty \in \mathbb{D}$ never appears in a tuple of $[\![\varphi]\!]$, provided that $p^{\mathcal{D}_i} \subseteq D^{\iota(p)}$ and $f^{\bar{\mathcal{D}}}(\bar{a}) \in D$, for every $p \in \mathsf{R}$, $f \in \mathsf{F}$, $i \in \mathbb{N}$, and $\bar{a} \in D^{\iota(f)}$, where $D = \mathbb{D} \setminus \{\bot_\infty\}$.

3.2 MFOTL$_\Omega$ and Extended Relational Algebra Operators

Our monitoring algorithm is based on interpreting MFOTL$_\Omega$ connectives in terms of extended relational algebra operators. This interpretation is represented by equalities between the evaluation of a formula and the evaluation of its direct subformulas, for each kind of formula defined in Section 3.1. Such equalities extend the standard ones [1] that express the relationship between first-order logic (without function symbols) and relational algebra, to function symbols, temporal operators, and group-by operators. Before presenting the equalities, we introduce the extended relational algebra operators.

3.2.1 Extended Relational Algebra Operators

We start by defining constraints. We assume a given infinite set of variables $Z = \{z_1, z_2, \ldots\} \subseteq \mathsf{V}$, ordered by their indices. A *constraint* is a formula $r(t_1, \ldots, t_n)$ or its negation, where r is a rigid predicate symbol of arity n and the t_is are constraint terms, i.e., terms with variables in Z. We assume that for each domain element $d \in \mathbb{D}$, there is a corresponding constant, also denoted by d. A tuple $(a_1, \ldots, a_k)$ satisfies the constraint $r(t_1, \ldots, t_n)$ iff $\bigcup_{i=1}^{n} fv(t_i) \subseteq \{z_1, \ldots, z_k\}$ and $(v(t_1), \ldots, v(t_n)) \in r^{\mathcal{D}}$, where v is a valuation with $v(z_i) = a_i$, for all $i \in \{1, \ldots, k\}$. Satisfaction of a constraint $\neg r(t_1, \ldots, t_n)$ is defined similarly.

In the following, let C be a set of constraints, $A \subseteq \mathbb{D}^m$, and $B \subseteq \mathbb{D}^n$. The *selection* of A with respect to C is the m-ary relation

$$\sigma_C(A) := \{\bar{a} \in A \mid \bar{a} \text{ satisfies all constraints in } C\}\,.$$

The integer i is a *column* in A if $1 \leq i \leq m$. Let $\bar{s} = (s_1, s_2, \ldots, s_k)$ be a sequence of $k \geq 0$ columns in A. The *projection* of A on $\bar{s}$ is the k-ary relation

$$\pi_{\bar{s}}(A) := \left\{(a_{s_1}, a_{s_2}, \ldots, a_{s_k}) \in \mathbb{D}^k \,\middle|\, (a_1, a_2, \ldots, a_m) \in A\right\}.$$

Let $\bar{s}$ be a sequence of columns in $A \times B$. The *join* and the *antijoin* of A and B with respect to $\bar{s}$ and C are defined as

$$A \bowtie_{\bar{s},C} B := (\pi_{\bar{s}} \circ \sigma_C)(A \times B)$$

and

$$A \rhd_{\bar{s},C} B := A \setminus (A \bowtie_{\bar{s},C} B)\,.$$

Let ω be an operator in Ω, G a set of $k \geq 0$ columns in A, and t a constraint term. The *ω-aggregate* of A on t with grouping by G is the $(k+1)$-ary relation

$$\omega_t^G(A) := \left\{(b, \bar{a}) \,\middle|\, \bar{a} = (a_{g_1}, a_{g_2}, \ldots, a_{g_k}) \in \pi_{\bar{g}}(A) \text{ and } b = \omega(M_{\bar{a}})\right\}.$$

Here $\bar{g} = (g_1, g_2, \ldots, g_k)$ is the maximal subsequence of $(1, 2, \ldots, m)$ such that $g_i \in G$, for $1 \leq i \leq k$, and $M_{\bar{a}} : \mathbb{D}^{m-k} \to \mathbb{N}$ is the finite multi-set

$$M_{\bar{a}} := \left\{\!\!\left\{(\pi_{\bar{h}} \circ \sigma_{\{d \approx t\} \cup D})(A) \,\middle|\, d \in \mathbb{D}\right\}\!\!\right\},$$

where $\bar{h}$ is the maximal subsequence of $(1, 2, \ldots, m)$ with no element in G and $D := \{a_i \approx z_{g_i} \mid 1 \leq i \leq k\}$.

3.2.2 Interpreting MFOTL$_\Omega$ Connectives as Extended Regular Algebra Operators

Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal database, $i \in \mathbb{N}$, and $\varphi \in \mathcal{F}$. We express $[\![\varphi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$ in terms of the generalized relational algebra operators. The following equalities follow directly from the semantics of MFOTL$_\Omega$ formulas and the definition of the extended relational algebra operators.

Kind (FLX). This case is straightforward. For a predicate symbol $p \in \mathsf{R}_f$ of arity n and pairwise distinct variables $x_1, \ldots, x_n \in \mathsf{V}$,

$$[\![p(x_1, \ldots, x_n)]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = p^{\mathcal{D}_i}\,.$$

Kinds ($\mathsf{RIG}_\wedge$) *and* ($\mathsf{RIG}_{\wedge\neg}$). Let ψ and $p(t_1, \ldots, t_n)$ be two formulas such that $\psi \wedge p(t_1, \ldots, t_n)$ is a formula of kind ($\mathsf{RIG}_\wedge$). Note that $\psi \wedge \neg p(t_1, \ldots, t_n)$ is a formula of kind ($\mathsf{RIG}_{\wedge\neg}$). Then

$$[\![\psi \wedge p(t_1, \ldots, t_n)]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \sigma_{\{p(\theta(t_1),\ldots,\theta(t_n))\}}\left([\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}\right)$$

and

$$[\![\psi \wedge \neg p(t_1, \ldots, t_n)]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \sigma_{\{\neg p(\theta(t_1),\ldots,\theta(t_n))\}}\left([\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}\right),$$

where the substitution $\theta : fv(\psi) \to \{z_1, \ldots, z_{|fv(\psi)|}\}$ is given by $\theta(x) = z_j$, with j the index of x in $\bar{fv}(\psi)$. For instance, if $\varphi \in \mathcal{F}$ is the formula $\psi(x, y) \wedge (x - y) \ mod\ 2 \approx 0$ then $[\![\varphi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \sigma_{\{(z_1 - z_2) \ mod\ 2 \approx 0\}}[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$.

Kind ($\mathsf{RIG}'_\wedge$). Let $\psi \wedge p(t_1,\ldots,t_n)$ be a formula of kind ($\mathsf{RIG}'_\wedge$), with $\bar{fv}(\psi) = (y_1,\ldots,y_\ell)$. Then

$$[\![\psi \wedge p(t_1,\ldots,t_n)]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \bigcup_{\bar{d}\in[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}} [\![p(t_1,\ldots,t_n) \wedge \bigwedge_{j\in\{1,\ldots,\ell\}} y_j \approx d_j]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}.$$

For instance, let $\varphi(x,y,z) = \psi(y,z) \wedge x \prec y + z$. Assume that $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \{(2,0),(1,2)\}$. Then $[\![\varphi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = [\![x \prec y+z \wedge y \approx 2 \wedge z \approx 0]\!] \cup [\![x \prec y+z \wedge y \approx 1 \wedge z \approx 2]\!] = \{(0,2,0),(1,2,0)\} \cup \{(0,1,2),(1,1,2),(2,1,2)\}$.

Kinds ($\mathsf{GEN}_\wedge$) *and* ($\mathsf{GEN}_{\wedge\neg}$). Let $\psi \wedge \psi'$ and $\psi \wedge \neg\psi'$ be formulas of kind ($\mathsf{GEN}_\wedge$) and respectively ($\mathsf{GEN}_{\wedge\neg}$), with $\bar{fv}(\psi) = (y_1,\ldots,y_n)$ and $\bar{fv}(\psi') = (y'_1,\ldots,y'_\ell)$. Then

$$[\![\psi \wedge \psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} \bowtie_{\bar{s},C} [\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$$

and

$$[\![\psi \wedge \neg\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} \rhd_{\bar{s},C} [\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)},$$

where (a) $\bar{s} = (1,\ldots,n,n+i_1,\ldots,n+i_\ell)$ with i_j such that $(i_1,\ldots,i_\ell)$ is the maximal subsequence of $(1,\ldots,\ell)$ with $y'_{i_j} \notin fv(\psi)$, and (b) $C = \{z_j \approx z_{n+h} \mid y_j = y'_h,\ 1 \le j \le n,\ \text{and } 1 \le h \le \ell\}$. For instance, if $\varphi = p(x,y) \wedge q(y,z)$ then $\bar{s} = (1,2,4)$ and $C = \{z_2 \approx z_3\}$.

Kind ($\mathsf{GEN}_\vee$). Let $\psi \vee \psi'$ be a formula of kind ($\mathsf{GEN}_\vee$). Then

$$[\![\psi \vee \psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} \cup [\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}.$$

Kind ($\mathsf{GEN}_\exists$). Let $\exists x.\,\psi$ be a formula of kind ($\mathsf{GEN}_\exists$) with $\bar{fv}(\psi) = (y_1,\ldots,y_k)$. Then

$$[\![\exists x.\,\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \pi_{\bar{\jmath}}\left([\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}\right),$$

where $\bar{\jmath} = (1,\ldots,k)$ if $x \notin fv(\psi)$ and otherwise $\bar{\jmath} = (1,\ldots,j-1,j+1,\ldots,k)$ with j such that $x = y_j$.

Kind ($\mathsf{GEN}_\bullet$). Let $\bullet_I\,\psi$ be a formula of kind ($\mathsf{GEN}_\bullet$). Then

$$[\![\bullet_I\,\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \begin{cases} [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i-1)} & \text{if } i > 0 \text{ and } \tau_i - \tau_{i-1} \in I, \\ \emptyset & \text{otherwise.} \end{cases}$$

Kinds (GEN_S) *and* ($\mathsf{GEN}_{\neg\mathsf{S}}$). Let $\psi\,\mathsf{S}_I\,\psi'$ and $\neg\psi\,\mathsf{S}_I\,\psi'$ be two formulas of kind (GEN_S) and respectively ($\mathsf{GEN}_{\neg\mathsf{S}}$), with $\bar{fv}(\psi) = (y_1,\ldots,y_n)$ and $\bar{fv}(\psi') = (y'_1,\ldots,y'_\ell)$. Then

$$[\![\psi\ \mathsf{S}_I\ \psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \bigcup_{j\in\{i'\mid i'\le i,\ \tau_i-\tau_{i'}\in I\}} \left([\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},j)} \bowtie_{\bar{s},C} \left(\bigcap_{k\in\{j+1,\ldots,i\}} [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},k)}\right)\right),$$

and

$$[\![\neg\psi\ \mathsf{S}_I\ \psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \bigcup_{j\in\{i'\mid i'\le i,\ \tau_i-\tau_{i'}\in I\}} \left([\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},j)} \rhd_{\bar{s},C} \left(\bigcap_{k\in\{j+1,\ldots,i\}} [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},k)}\right)\right),$$

where $\bar{s}$ and C are as for the case of kinds ($\mathsf{GEN}_\wedge$) and ($\mathsf{GEN}_{\wedge\neg}$). For instance, for $\bar{fv}(\psi) = (x,y,z)$ and $\bar{fv}(\psi') = (z,z',x)$, we have $\bar{s} = (1,2,3,5)$ and $C = \{z_1 \approx z_6, z_3 \approx z_4\}$.

Kinds ($\mathsf{GEN}_{\mathsf{T}}$). Let $\psi\ \mathsf{T}_I\ \psi'$ be a formula of kind ($\mathsf{GEN}_{\mathsf{T}}$). Then

$$[\![\psi\ \mathsf{T}_I\ \psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \Big(\bigcap_{j\in\{i'\mid i'\le i,\ \tau_i-\tau_{i'}\in I\}} [\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},j)} \Big) \cup \bigcup_{j\in\{i'\mid i'\le i,\ \tau_i-\tau_{i'}\in I\}} \Big([\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},j)} \bowtie_{\bar{s},C} \big(\bigcap_{k\in\{j,\ldots,i\}} [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},k)} \big) \Big),$$

where $\bar{s}$ and C are as for the case of kinds ($\mathsf{GEN}_{\wedge}$) and ($\mathsf{GEN}_{\wedge\neg}$). This equality follows the semantics of the T operator, that is, $(\bar{\mathcal{D}},\bar{\tau},v,i) \models \psi \mathsf{T}_I \psi'$ iff $(\bar{\mathcal{D}},\bar{\tau},v,j) \models \psi'$ for all j with $j \le i$ and $\tau_i - \tau_j \in I$, or there is a j with $j \le i$ and $\tau_i - \tau_j \in I$ such that $(\bar{\mathcal{D}},\bar{\tau},v,k) \models \psi$, for all k with $j \le k \le i$.

Kind (GEN_{ω}). Let $[\omega_t\, \bar{z}'.\, \psi](y;\bar{g})$ be a formula of kind (GEN_{ω}). It holds that

$$[\![[\omega_t\, \bar{z}'.\, \psi](y;\bar{g})]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \omega^{G}_{\theta(t)}\big([\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}\big),$$

where $\bar{fv}(\psi) = (y_1,\ldots,y_n)$, for some $n \ge 0$, $G = \{i \mid y_i \in \bar{g}\}$, and the substitution $\theta : fv(\psi) \to \{z_1,\ldots,z_n\}$ is given by $\theta(x) = z_j$, where j is the index of x in $\bar{fv}(\psi)$. For instance, for $[\mathsf{SUM}_{x+y}\, x, y.\, p(x,y,z)](s;z)$, we have $G = \{3\}$ and $\theta(t) = z_1 + z_2$.

Remark. We do not have a translation from formulas in $\mathcal{F}$ into extended relational algebra expressions because one cannot fix in advance the relational symbols used by such expressions. Indeed, the right-hand side of the equalities for the kind ($\mathsf{RIG}'_{\wedge}$) and the kinds corresponding to temporal operators depend not only on the left-hand side formula, but also on the temporal database.

3.3 Algorithmic Realization

For a given formula $\psi \in \mathcal{F}$, the algorithm iteratively processes the given temporal database $(\bar{\mathcal{D}},\bar{\tau})$. At each time point i, it calls the procedure eval to compute $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$. The input of eval at time point i is the formula ψ, the time point i with its timestamp τ_i, and the interpretations of the flexible predicate symbols, i.e., $r^{\mathcal{D}_i}$, for each $r \in \mathsf{R}_f$. Note that $\bar{\mathcal{D}}$'s domain and the interpretations of the rigid predicate symbols and the function symbols, including the constants, do not change over time. We assume that they are fixed in advance.

The computation of $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$ is by recursion over ψ's formula structure. To accelerate the computation of $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$, the monitoring algorithm maintains state for each temporal subformula, storing previously computed intermediate results. The monitor's state is initialized by the procedure init and updated in each iteration by the procedure eval. We describe the algorithm's state for each temporal operator when we present the pseudo-code that handles the operator.

The pseudo-code of the procedures init and eval is given in Figure 3. Our pseudo-code (also used in Figures 4 and 5) is written in a functional-programming style with pattern matching. The symbol $\langle\rangle$ denotes the empty sequence, $+\!\!+$ sequence concatenation, $h :: L$ the sequence with head h and tail L, and $\lambda x.f(x)$ denotes a function f. The functions $\mathsf{hd}(L)$ and $\mathsf{tl}(L)$ return the head and respectively the tail of the non-empty list L.

```
proc init(φ)
  for each ψ ∈ sf(φ) with ψ = ●_I ψ' do (A_ψ, τ_ψ) ← (∅, 0)
  for each ψ ∈ sf(φ) with ψ = ψ1 S_I ψ2 do L_ψ ← ⟨⟩
  for each ψ ∈ sf(φ) with ψ = ψ1 T_I ψ2 do (H_ψ, L_ψ) ← (⟨⟩, ⟨⟩)

proc eval(φ, i, τ, Γ)
  case φ = p(x1, ..., xn)
    return Γ_p

  case φ = ψ ∧ p(t1, ..., tn)
         & kind_rig(φ)
  case φ = ψ ∧ ¬p(t1, ..., tn)
         & kind_rig(φ)
    A ← eval(ψ, i, τ, Γ)
    C ← get_info_rig(φ)
    return σ_C(A)

  case φ = ψ ∧ p(t1, ..., tn)
         & kind_rig'(φ)
    A ← eval(ψ, i, τ, Γ)
    k ← get_info_rig'(φ)
    R ← ∅
    for each ā ∈ A
      R ← R ∪ reval(p, k, ā)
    return R

  case φ = ψ ∧ ¬ψ'
  case φ = ψ ∧ ψ'
    A ← eval(ψ, i, τ, Γ)
    A' ← eval(ψ', i, τ, Γ)
    C, s̄ ← get_info_and(φ)
    if φ = ψ ∧ ψ' then
      return A ⋈_{C,s̄} B
    else
      return A ▷_{C,s̄} B

  case φ = ψ ∨ ψ'
    A ← eval(ψ, i, τ, Γ)
    A' ← eval(ψ', i, τ, Γ)
    return A ∪ A'

  case φ = ∃x̄. ψ
    A ← eval(ψ, i, τ, Γ)
    s̄ ← get_info_exists(φ)
    return π_s̄(A)

  case φ = [ω_t z̄. ψ](y; ḡ)
    A ← eval(ψ, i, τ, Γ)
    H, t' ← get_info_agg(φ)
    return ω^H_{t'}(A)

  case φ = ●_I ψ
    A' ← A_φ
    A_φ ← eval(ψ, i, τ, Γ)
    τ' ← τ_φ
    τ_φ ← τ
    if i > 0 and (τ − τ') ∈ I then
      return A'
    else
      return ∅

  case φ = ¬ψ S_I ψ'
  case φ = ψ S_I ψ'
    A ← eval(ψ, i, τ, Γ)
    A' ← eval(ψ', i, τ, Γ)
    return eval_since(φ, τ, A, A')

  case φ = ψ T_I ψ'
    A ← eval(ψ, i, τ, Γ)
    A' ← eval(ψ', i, τ, Γ)
    R ← eval_palways(φ, τ, A')
    S ← eval_since'(φ, τ, A', A)
    return R ∪ S
```

Fig. 3 The init and eval procedures.

First-order Connectives. We now describe the eval procedure in more detail. The cases correspond to the rules defining the set of monitorable formulas. The pseudocode for the cases corresponding to non-temporal connectives follows closely the equalities given in Section 3.2.2. Note that extended relational algebra operators have standard, efficient implementations [18], which can be used to evaluate the expressions on the right-hand side of these equalities.

The predicates kind_rig and kind_rig' check whether the input formula φ is indeed of the intended kind. The get_info_$*$ procedures return the parameters used by the corresponding relational algebra operators. For instance, get_info_rig returns the singleton set consisting of the constraint corresponding to the restrictions $p(t_1, \ldots, t_{\iota(p)})$ or $\neg p(t_1, \ldots, t_{\iota(p)})$. Similarly, get_info_rig' returns the effective index corresponding to the unique variable that appears only in the right conjunct of φ. The procedure $\mathsf{reval}(p, k, \bar{a})$ returns the set $\{d \in \mathbb{D} \mid (a_1, \ldots, a_{k-1}, d, a_k, \ldots, a_{n-1}) \in p^{\bar{\mathcal{D}}}\}$, for any $\bar{a} \in \mathbb{D}^{n-1}$, where n is the arity of the rigid predicate symbol p.

Aggregation Operators. Computing the aggregation $\omega^H_{t'}(A)$ is standard [18]. Namely, one iterates through the tuples in the relation A and maintains a data structure that associates an accumulated value for the aggregation term t' to each group of A, that is, to each tuple of values for the aggregation attributes in H. The accumulation depends on the aggregation operator. For instance, for CNT, it is the number of tuples of A seen so far that belong to the group, for SUM, it is the

```
proc eval_since(φ, τ, A, A′)
  b ← interval_right_margin(φ)
  drop_old(L_φ, b, τ)
  C, s̄ ← get_info_and(φ)
  case φ = ¬ψ S_I ψ′ then
    f ← λB.B ▷_{s̄,C} A
  case φ = ψ S_I ψ′ then
    f ← λB.B ⋈_{s̄,C} A
  g ← λ(κ, B).(κ, f(B))
  L_φ ← map(g, L_φ)
  L_φ ← L_φ ++ ⟨(τ, A′)⟩
  return fold_left(aux_since, ∅, L_φ)

proc drop_old(L, b, τ)
  case L = ⟨⟩
    return ⟨⟩
  case L = (κ, B) :: L′
    if τ − κ ≥ b then
      return drop_old(L′, b, τ)
    else return L

proc aux_since(R, (κ, B))
  if (τ − κ) ∈ I then return R ∪ B
  else return R
```

Fig. 4 The eval_since procedure.

sum of values for t' corresponding to such tuples, and for AVG it is the pair of the values used for CNT and SUM. The accumulated values are updated at each iteration. For instance, for CNT, the accumulated value is increased by one. When A's scan is finished, the aggregated value for each group is obtained from the accumulated value. Suitable data structures are hash tables and balanced search trees as they allow for fast lookups and updates.

Finally, note that when handling an aggregation operator, one only needs a suitable accumulation, functions for initializing and updating this accumulation, and a function f for obtaining the aggregated value from the accumulated value. In the general case, the accumulation consists of all the values for t' seen so far and the function f is the aggregation operator itself. For many aggregation operators, for instance for the ones in considered in this article, the computation of the accumulated value can be carried out more efficiently.

Temporal Operators. Consider first the case where the formula φ is of the form $\bullet_I \psi$. In this case, the state stores between the iterations $i-1$ and i, when $i > 0$, the timestamp of the last time point, namely $\tau_\varphi := \tau_{i-1}$, and the tuples that satisfy ψ at last time point $i-1$, i.e., the relation $A_\varphi := [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i-1)}$. To evaluate φ at the current time point i, we recursively evaluate the subformula ψ at i, we update the state, and we return the relation resulting from the evaluation of ψ at the previous time point, provided that the temporal constraint is satisfied. Otherwise we return the empty relation. Note that by storing the relation $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$ at time point i, the subformula ψ need not be evaluated again at time point i during the evaluation of ψ at time point $i+1$.

Consider now the case where the formula φ is of the form $\psi \,\mathsf{S}_I\, \psi'$ or $\neg\psi \,\mathsf{S}_I\, \psi'$, where $I = [a, b)$, for some $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$. This case is mainly handled by the sub-procedure eval_since, given in Figure 4. For clarity of presentation, we assume that $\varphi = \psi \mathsf{S}_I \psi'$, the other case being similar. The evaluation of φ reflects the logical equivalence $\psi \,\mathsf{S}_I\, \psi' \equiv \bigvee_{d \in I} \psi \,\mathsf{S}_{[d,d]}\, \psi'$. Note that we abuse notation here, as the right-hand side is not a formula when $b = \infty$. The function interval_right_margin(φ) returns b.

The state at time point i, that is, after the procedure eval$(\varphi, i, \tau_i, \Gamma_i)$ has been executed, consists of the list L_φ of tuples (τ_j, R_j^i) ordered with j ascending, where

```
proc eval_palways(φ, τ, A′)
  b ← interval_right_margin(φ)
  drop_old(H_φ, b, τ)
  H_φ ← H_φ ++ ⟨(τ, A′)⟩
  (R, κ) ← hd(H_φ)
  if (τ − κ) ∈ I then
    return fold_left(aux_palways, R, tl(H_φ))
  else return ∅

proc aux_palways(R, (κ, B))
  if (τ − κ) ∈ I then return R ∩ B
  else return R
```

Fig. 5 The eval_palways procedure.

j is such that $j \leq i$ and $\tau_i - \tau_j < b$ and with

$$R_j^i := [\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},j)} \bowtie_{\bar{s},C} \Big(\bigcap_{k\in\{j+1,\ldots,i\}} [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},k)} \Big) ,$$

with $\bar{s}$ and C defined as in Section 3.2.2. We have

$$[\![\varphi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = \bigcup_{j\in\{i'\mid i'\leq i,\tau_i-\tau_{i'}\in I\}} R_j^i \,.$$

The computation of this union is performed in the last line of the eval_since procedure. Note that, in general, not all the relations R_j^i in the list L_φ are needed for the evaluation of φ at time point i. However, the relations R_j^i with j such that $\tau_i - \tau_j \notin I$, that is $\tau_i - \tau_j < a$, are stored for the evaluation of φ at future time points $i' > i$. By storing these relations, the subformulas ψ_1 and ψ_2 need not be evaluated again at time points $j < i$ during the evaluation of ψ at time point i.

We now explain how the state is updated at time point i from the state at time point $i-1$. We first drop from the list L_φ the tuples that are no longer relevant. More precisely, we drop the tuples that have as their first component a timestamp τ_j for which the distance to the current timestamp τ_i is too large with respect to the right margin of I. This is done by the procedure drop_old. Next, the state is updated using the logical equivalence $\alpha \,\mathsf{S}\, \beta \equiv (\alpha \wedge \bullet(\alpha \,\mathsf{S}\, \beta)) \vee \beta$. This is accomplished in two steps. First, we update each element of L_φ so that the tuples in the stored relations also satisfy ψ at the current time point i. This step corresponds to the conjunction in the above equivalence and it is performed by the map function. The update is based on the equality $R_j^i = R_j^{i-1} \bowtie_{\bar{s},C} [\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$. Note that the join distributes over the intersection. The second step, which corresponds to the disjunction in the above equivalence, consists of appending the tuple (τ_i, R_i^i) to L_φ. Note that $R_i^i = [\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$.

Finally, we consider the case where the formula φ is of the form $\psi \,\mathsf{T}_I\, \psi'$. The pseudo-code for this case in the eval procedure reflects the logical equivalence $\psi \,\mathsf{T}_I\, \psi' \equiv (\blacksquare_I \,\psi') \vee (\psi' \,\mathsf{S}_I\, \psi \wedge \psi')$. This case is mainly handled by the procedures eval_palways and eval_since', which correspond to the left-hand and respectively the right-hand side of the union operator of the right-hand side of the equality given in Section 3.2.2. The pseudo-code of the eval_since' procedure is similar to that of the eval_since procedure, and thus omitted. The only difference consists in replacing the assignment $L_\varphi \leftarrow L_\varphi +\!\!+ \langle(\tau, A')\rangle$ by $L_\varphi \leftarrow L_\varphi +\!\!+ \langle(\tau, A \cap A')\rangle$. The list L_φ, which is part of the state maintained for φ, has the same meaning as for the case of the S_I operator. Note also that the order of the parameters in the call to eval_since' is reversed in comparison to eval_since; this matches the previously

given equivalence. The pseudo-code of the eval_palways is given in Figure 5 and it represents the evaluation of formulas of the form $\blacksquare_I \psi'$ ("always in the past ψ'"). This procedure uses and maintains the other part of the state for φ, namely the list H_φ. At time point i, after the procedure $\mathsf{eval}(\varphi, i, \tau_i, \Gamma_i)$ has been executed, the list H_φ consists of the tuples $(\tau_j, [\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},j)})$ with $j \leq i$ and $\tau_i - \tau_j \in I$, ordered with j ascending. The list H_φ is updated at each iteration by eliminating old tuples using the same procedure drop_old as in the S_I case, and by appending the tuple $(\tau_i, [\![\psi']\!]^{(\bar{\mathcal{D}},\bar{\tau},i)})$. The procedure eval_palways returns the intersection on the left-hand side of the union operator of the equality for the T_I operator. As in the S_I case, this intersection is computed by calling the standard fold_left function on the list H_φ, this time using the auxiliary procedure aux_palways.

The following theorem states the correctness of our algorithm. Its proof follows the algorithm's presentation, and it proceeds by induction using the lexicographic ordering on tuples $(i, |\varphi|)$, where $i \in \mathbb{N}$ and $|\varphi|$ denotes φ's size, defined as expected.

Theorem 7 *Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal database, $i \in \mathbb{N}$, and $\psi \in \mathcal{F}$. The procedure $\mathsf{eval}(\psi,\ i,\ \tau_i,\ \Gamma_i)$ returns the relation $[\![\psi]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}$, whenever $\mathsf{init}(\psi)$, $\mathsf{eval}(\psi,\ 0,\ \tau_0,\ \Gamma_0)$, $\ldots$, $\mathsf{eval}(\psi, i{-}1, \tau_{i-1}, \Gamma_{i-1})$ were called previously in this order, where $\Gamma_j = (p^{\mathcal{D}_j})_{p \in \mathsf{R}_f}$ is the family of interpretations of flexible predicates at j, for every time point $j \in \mathbb{N}$.*

Optimizations. Several optimizations are possible when evaluating formulas, in particular those formulas of the form $\psi_1\ \mathsf{S}_I\ \psi_2$ and $\psi_1\ \mathsf{T}_I\ \psi_2$. For instance, when $I = [0, \infty)$, for the S_I operator, it is sufficient to store the resulting relation from the previous time point as we have $[\![\psi_1\, \mathsf{S}\, \psi_2]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} = [\![\psi_2]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)} \cup \big([\![\psi_1\, \mathsf{S}\, \psi_2]\!]^{(\bar{\mathcal{D}},\bar{\tau},i-1)} \bowtie [\![\psi_1]\!]^{(\bar{\mathcal{D}},\bar{\tau},i)}\big)$. Further optimizations for incrementally updating the relations of the temporal formulas are described in [8].

We also present an optimization for the frequently occurring pattern $[\omega_x\, \bar{z}.\, \blacklozenge_I\, \psi](y; \bar{g})$. Instead of applying the aggregation operator to the relation for the formula $\blacklozenge_I\, \psi$, we directly compute the aggregation from the relations for the formula ψ at the time points in the specified time window. The approach is an adaptation of the one for handling stand-alone aggregation operators, where one maintains a map between groups and accumulated values. For $[\omega_x\, \bar{z}.\, \blacklozenge_I\, \psi](y; \bar{g})$, this map is not re-built from scratch at each time point; instead, it is stored and updated at each time point. In addition to a function for updating accumulations when tuples "enter" the relation for $\blacklozenge_I\, \psi$, we also use a function to update accumulations when tuples "leave" this relation. For instance, for CNT, one decreases the accumulated value by 1. By using a multi-set for storing the relation for $\blacklozenge_I\, \psi$, we can efficiently determine when a tuple enters and when it leaves this relation.

4 Evaluation

In this section, we evaluate our extension of metric first-order temporal logic with aggregation operators. First, we evaluate whether MFOTL$_\Omega$ is a suitable language for expressing complex policies with aggregations. Second, we evaluate the performance of our prototype monitor, comparing it with the stream-processing tool STREAM [2] and the relational database PostgreSQL [23]. In contrast to existing logic-based monitoring solutions, both of these tools support the aggregation of

$$\Box \forall u.\, \forall s.\, [\mathsf{SUM}_a\ a, \tau.\, \blacklozenge_{[0,31)}\ \mathit{withdraw}(u, a) \wedge \mathit{ts}(\tau)](s; u) \rightarrow s \preceq 10000 \tag{P1}$$

$$\Box \forall u.\, \forall s.\, [\mathsf{SUM}_a\ a, \tau.\, \blacklozenge_{[0,31)}\ \mathit{withdraw}(u, a) \wedge \mathit{ts}(\tau)](s; u) \wedge \big(\neg \mathit{limit\_off}(u)\ \mathsf{S}\ \mathit{limit\_on}(u)\big) \rightarrow s \preceq 10000 \tag{P2}$$

$$\Box \forall u.\, \forall s.\, \forall \ell.\, [\mathsf{SUM}_a\ a, \tau.\, \blacklozenge_{[0,31)}\ \mathit{withdraw}(u, a) \wedge \mathit{ts}(\tau)](s; u) \wedge \big(\neg \exists \ell'.\, \mathit{limit}(u, \ell')\ \mathsf{S}\ \mathit{limit}(u, \ell)\big) \rightarrow s \preceq \ell \tag{P3}$$

$$\Box \forall u.\, \forall s.\, \forall m.\, [\mathsf{AVG}_a\ a, \tau.\, \blacklozenge_{[0,91)}\ \mathit{withdraw}(u, a) \wedge \mathit{ts}(\tau)](s; u) \wedge [\mathsf{MAX}_a\ a.\, \blacklozenge_{[0,8)}\ \mathit{withdraw}(u, a)](m; u) \rightarrow m \preceq 2 \cdot s \tag{P4}$$

$$\Box \forall s.\, [\mathsf{AVG}_c\ c, u.\, [\mathsf{CNT}_a\ a, \tau.\, \blacklozenge_{[0,31)}\ \mathit{withdraw}(u, a) \wedge \mathit{ts}(\tau)](c; u)](s) \rightarrow s \preceq 150 \tag{P5}$$

$$\Box \forall u.\, \forall c.\, [\mathsf{CNT}_j\ v, p, \kappa.\, [\mathsf{AVG}_a\ a, \tau.\, \blacklozenge_{[0,31)}\ \mathit{withdraw}(u, a) \wedge \mathit{ts}(\tau)](v; u) \wedge \blacklozenge_{[0,31)}\ \mathit{withdraw}(u, p) \wedge \mathit{ts}(\kappa) \wedge 2 \cdot v \prec p](c; u) \rightarrow c \preceq 5 \tag{P6}$$

Fig. 6 Policy formalizations.

data values and their performance is comparable to other state-of-the-art tools in their respective domains.

4.1 Specification Language

To evaluate MFOTL$_\Omega$'s suitability for specifying policies with aggregations, we compare specifications in MFOTL$_\Omega$ with those in the prominent query languages CQL [3] (STREAM's query language) and SQL. For the comparison, we use the following six policies rooted in the domain of fraud detection.

1. The sum of withdrawals of each user in the last 31 days does not exceed the limit of $10,000.
2. Similar to the first policy, except that the withdrawals must not exceed $10,000 only when the flag for checking the limit is set.
3. Similar to the second policy, except that the withdrawal limit is set by the user.
4. The maximal withdrawal of each user in the last week must be at most twice the average of the user's withdrawals over the last 91 days.
5. The average number of withdrawals per user in the last 31 days must not exceed a given threshold of 150, where the average is taken over all users.
6. For each user, the number of withdrawal peaks in the last 31 days does not exceed a threshold of 5, where a withdrawal peak is a value at least twice the average over the last 31 days.

The MFOTL$_\Omega$ formulas that formalize the given policies are presented in Figure 6. Note that since we restrict ourselves in this article to the past-only fragment of MFOTL$_\Omega$, the outermost temporal operator $\Box$ ("always") is not part of our definition of the logic given in Section 2. However, we include it in our formalizations to emphasize that policies must be fulfilled at all time points. We use $\mathit{withdraw}(u, a)$ to denote that the user u has withdrawn the amount a and $\mathit{ts}(\tau)$ to denote the timestamp τ of a time point.

In the MFOTL$_\Omega$ formalization of the first policy, the SUM operator adds all the withdrawal amounts in the past 31 days, and we require that the result is less than 10,000. We use $\mathit{ts}(\tau)$ to differentiate the different withdrawals of the

same amount made by the user within a given time window (see Example 4). The formalization of the second policy is a simple extension of the first, where we just add the condition that a user's limit is set. We use $\mathit{limit\_on}(u)$ to denote that a user u sets the limit flag and $\mathit{limit\_off}(u)$ to denote that u unsets it. Using the temporal operator S, we express the existence of a time point in the past where the user has set the limit flag and has not unset it since then. To formalize the third policy, we use $\mathit{limit}(u, \ell)$ to represent that u sets his limit to ℓ. To simulate setting no limit, a user can set an arbitrarily high limit, and we assume that when a new account is opened, the limit is set to some default value. The S operator is now used to find the latest limit that has been set by the user. This limit is then used to constrain the sum of all withdrawals.

For the fourth policy, AVG computes the average withdrawal amount over the last 91 days and MAX finds the maximum withdrawal amount for the last week. We require that the maximum is at most double the average. For the fifth policy, we first use CNT to count the number of withdrawals made over last 31 days by each user. We then use AVG to compute the average number of withdrawals per user during this time period, which we require not to exceed 150. Finally for the sixth policy, we use AVG to compute the average withdrawal over the last 31 days for each user. The CNT operator then counts all withdrawals with amounts greater than twice the calculated average. We require that the count is not greater than 5.

Before we compare MFOTL_Ω with SQL and CQL, we remark that the given MFOTL_Ω formalizations follow the common pattern $\Box \forall \bar{x}. \forall \bar{y}. \varphi(\bar{x}, \bar{y}) \wedge c(\bar{x}, \bar{y}) \rightarrow \psi(\bar{y}) \wedge c'(\bar{y})$, where c and c' represent restrictions, i.e., formulas of the form $r(\bar{t})$ and $\neg r(\bar{t})$ with $r \in \mathsf{R}_r$. The formula to be monitored, i.e., $\varphi(\bar{x}, \bar{y}) \wedge c(\bar{x}, \bar{y}) \wedge \neg(\psi(\bar{y}) \wedge c'(\bar{y}))$ is in the fragment $\mathcal{F}$ if φ and ψ are in $\mathcal{F}$, and both c and c' satisfy the conditions of the (RIG) rules. See Figure 2 in Section 3.1. It can be easily checked that this is indeed the case for the given formulas (P1) to (P6) and we can thus use our monitoring solution for them.

Comparison with SQL. SQL does not have temporal operators, and thus all temporal reasoning must be explicitly specified. This can be done by adapting the standard embedding of temporal logic into first-order logic to represent MFOTL_Ω formulas as SQL queries. The key ideas underlying the embedding are the following. First, we add to each predicate two additional attributes, tp and ts, which represent the time point and the timestamp of an event's occurrence. Second, we use the *tpts* predicate from Example 4, with two attributes, tp and ts, whose interpretation consists of all pairs of time points and associated timestamps. Finally, we express temporal constraints by arithmetic expressions over the newly introduced temporal data, that is, the data values for the tp and ts attributes. The *tpts* predicate is needed to preserve the semantic equivalence between MFOTL_Ω and its embedding in first-order logic, as there can be time points at which no event occurs. Expressing first-order formulas with aggregations as extended relational algebra expressions is done in a standard way [1].

To illustrate this approach, consider the following SQL query for reporting violations with respect to the first policy (P1).

```
SELECT T1.ts, SUM(T2.a) AS s, T2.u
FROM (SELECT * FROM tpts) AS T1,
     (SELECT tp AS tp', ts AS ts', u, a FROM withdraw) AS T2
WHERE T2.tp' ≤ T1.tp AND 0 ≤ T1.ts − T2.ts' AND T1.ts − T2.ts' ≤ 30
GROUP BY T1.tp, T1.ts, T2.u
```

```
HAVING SUM(T2.a) > 10000
ORDER BY T1.ts
```

A drawback of using SQL is that the queries are less succinct because they must explicitly account for temporal constraints within policies. Therefore, without an automated translation from MFOTL$_\Omega$ to SQL, queries for complex policies are difficult to specify and maintain. Moreover, and regardless of whether an automated translation is used, queries are hard to simplify and optimize. This is not just due to the query's complexity, the structure is also lost: since there is no distinction between temporal data and other data, an SQL engine cannot exploit the policy's temporal dimension to optimize the query's execution. Our performance evaluation in Section 4.2 illustrates this point.

Comparison with CQL. STREAM's query language CQL for data streams extends SQL with the *sliding window* construct. This construct takes as input a stream of timestamped events and a range. For each event in the stream, it outputs a relation that contains the current event and all the preceding events that fall within the given range. CQL's time model differs from MFOTL$_\Omega$'s and thus the meaning of range in CQL and MFOTL$_\Omega$ do not match. In CQL, there is no notion of time points and the sliding window evaluation is applied after each received event.

To illustrate the sliding window construct, consider the following CQL query, which returns all the violations of the first policy (P1).

```
sum_rel := SELECT SUM(a) AS s, u FROM withdraw [RANGE 31] GROUP BY u
SELECT * FROM sum_rel WHERE s > 10000
```

Here, the sliding window construct, syntactically denoted with the [...] expression, is applied over the withdraw stream. That is, for each event e with timestamp τ, a relation is created that contains e and all the events that happened between τ and $\tau - 31$ days. Finally, both SELECT queries are evaluated using the standard SQL semantics.

The CQL's sliding windows construct roughly corresponds to the $\blacklozenge_I$ operator in MFOTL$_\Omega$, where I is of the form $[0, t)$ with $t \in \mathbb{N} \cup \{\infty\}$. All other MFOTL$_\Omega$ operators must be implicitly encoded. To illustrate, consider the following CQL query to find violations of the second policy (P2).

```
cnt_on := SELECT COUNT(*) AS c_on, u FROM limit_on [RANGE Unbounded] GROUP BY u
cnt_off := SELECT COUNT(*) AS c_off, u FROM limit_off [RANGE Unbounded] GROUP BY u
limit_is_on := SELECT cnt_on.u FROM cnt_on, cnt_off
                              WHERE cnt_on.u = cnt_off.u AND c_off = c_on
SELECT sum_rel.s, sum_rel.u
FROM sum_rel, limit_is_on
WHERE sum_rel.u = limit_is_on.u AND s > 10000
```

To mimic the semantics of the S operator, we first count the number of limit_on and limit_off events for each user and produce the corresponding cnt_on and cnt_off relations. The [RANGE Unbounded] sliding window ranges over the entire stream up to the currently processed position. Second, we create the limit_is_on stream, which contains the users u that have the limit turned on at the current timestamp. The limit is turned on for user u if there are as many limit_on events as limit_off events. We assume here that, for each user, the limit is initially turned off and that the limit_on and limit_off events alternate. Finally, for each (s, u) tuple in sum_rel, we check whether u has turned his limit on at the current timestamp and, if so, whether s is greater than 10,000.

The previous workaround for policy (P2) does not apply to policy (P3). Here we must find the latest limit set for each user, and this is not possible without directly accessing the timestamps of events. Thus, to express the policy (P3) in CQL, we assume that events in the limit stream are tuples of the form (τ, u, ℓ) timestamped by τ, in contrast to withdraw events which are tuples of the form (u, a). We encode the MFOTL$_\Omega$ subformula $\neg\exists \ell'.\, \mathit{limit}(u, \ell')\, \mathsf{S}\, \mathit{limit}(u, \ell)$ with an SQL query that uses the timestamp field explicitly, in a manner similar to the approach used to express temporal operators in SQL. This encoding can be generalized and used for any MFOTL$_\Omega$ temporal operator. However, it has similar drawbacks to using SQL, as seen by STREAM's performance on (P3).

Temporal reasoning using only the sliding window in STREAM is limited in general. For example, we cannot check that certain event patterns happen at every time point in a given time window, whereas in MFOTL$_\Omega$ we can simply use the $\blacksquare_I$ operator. Moreover, we cannot select tuples from a time window that is strictly in the past. It is therefore in general not clear how to specify in CQL temporal constraints of the form $\varphi\, \mathsf{S}_I\, \psi$, with $0 \notin I$.

To illustrate the first limitation, consider the following policy. If during the past week a user's acount balance is continually negative, that is, the amount withdrawn exceeds the amount deposited at each time point during the week, then the user must not withdraw more money from his account. In MFOTL$_\Omega$, this policy is formalized by the following formula, where $\mathit{deposit}(u, a)$ has the expected meaning.

$$\begin{aligned}\square \forall u.\, \big(\blacksquare_{[0,8)}\, \exists w.\, \exists d.\, &[\mathsf{SUM}_a\, a, \tau.\, \blacklozenge\, \mathit{withdraw}(u, a) \wedge \mathit{ts}(\tau)](w; u) \wedge \\ &[\mathsf{SUM}_a\, a, \tau.\, \blacklozenge\, \mathit{deposit}(u, a) \wedge \mathit{ts}(\tau)](d; u) \wedge w \succ d \big) \rightarrow \\ \neg\exists a.\, &\mathit{withdraw}(u, a)\end{aligned}$$

The following policy illustrates the second limitation. If a user makes a withdrawal larger than \$1,000, then he must not have been in-debt during the last seven days. In MFOTL$_\Omega$, this policy is formalized by the formula

$$\square \forall u.\, \big(\exists a.\, \mathit{withdraw}(u, a) \wedge a \succ 1000\big) \rightarrow \big(\neg \mathit{indebt}(u)\, \mathsf{S}_{[8,\infty)}\, \mathit{outdebt}(u)\big)\,,$$

where we assume that the time points when the user u goes into debt and out of debt are marked by $\mathit{indebt}(u)$ and $\mathit{outdebt}(u)$, respectively. The subformula $\neg \mathit{indebt}(u)\, \mathsf{S}_{[8,\infty)}\, \mathit{outdebt}(u)$ holds when the last *outdebt* event for the user u happened more than 7 days ago and no *indebt* event for u has happened since then. Here we assume that each user is initially not in debt, and this is marked with a corresponding *outdebt* event.

In summary, the sliding window operator is restrictive, even in CQL's simple underlying time model, namely, a stream of timestamped events. Since the sliding window operator is CQL's only construct for performing temporal reasoning directly, one must often combine it in ad-hoc ways with other language constructs to express temporal constraints. In contrast, MFOTL$_\Omega$ has richer support for expressing temporal constraints over a more sophisticated time model (e.g., time points are timestamped and multiple events can happen at the same time point). In particular, the temporal operator S_I in combination with the other Boolean connectives often allows one to express temporal properties naturally.

Table 1 Running times (STREAM / MonPoly extension / PostgreSQL) in seconds. Timeouts after 3,600 seconds are marked with the symbol † and out of memory or runtime errors with ‡.

policy \ time span	400	800	1200	1600	2000
(P1)	8 / 9 / 76	9 / 19 / 279	11 / 29 / 610	12 / 39 / 1065	14 / 48 / 1650
(P2)	21 / 10 / 247	23 / 20 / 1646	24 / 30 / †	26 / 40 / †	28 / 50 / †
(P3)	‡ / 21 / 193	‡ / 40 / 1125	‡ / 61 / †	‡ / 81 / †	‡ / 101 / †
(P4)	‡ / 22 / 168	‡ / 44 / 604	‡ / 66 / 1230	‡ / 88 / 2251	‡ / 110 / 3458
(P5)	12 / 9 / 75	15 / 19 / 280	15 / 29 / 612	17 / 38 / 1068	19 / 48 / 1650
(P6)	24 / 76 / 83	33 / 157 / 337	41 / 234 / 745	49 / 313 / 1351	59 / 395 / 2099

4.2 Tool Performance

For our performance evaluation, we use the policies from Section 4.1 and synthetically generated logs with different time spans (in days).[1] The logs contain withdraw events from 500 users, except for (P6), for which we consider only 100 users. Each user makes on average five withdrawals per day. The SQL queries for PostgreSQL and the CQL queries for STREAM are manually obtained from the corresponding MFOTL$_\Omega$ formulas (P1) to (P6). The MFOTL$_\Omega$ formulas and SQL queries have the same semantics, while the semantic differences between MFOTL$_\Omega$ and CQL are not substantial for the policies and logs considered. In particular, the tools (PostgreSQL version 9.1.4, STREAM version 0.6.0, and our prototype, which extends our monitoring tool MonPoly [5]) output the same violations. Finally, note that the formulas differ in the number of temporal and aggregation operators, as well as their respective nesting.

Table 1 shows the running times of the three tools on a standard desktop computer with 8 GB of RAM and an Intel Core i5 CPU with 2.67 GHz. PostgreSQL's running times only account for the query evaluation, performed once per log file, and not for populating the database. For MAX aggregations, STREAM aborts with a runtime error. We mark this in the table with the symbol ‡. Overall, our tool's performance is between STREAM's and PostgreSQL's for our examples. We also note that STREAM and our tool scale linearly in our experiments with respect to the logs' time spans. This is not the case for PostgreSQL.

Regarding memory usage, our tool uses less than 50 MB for each policy, and memory consumption does not depend on the logs' time span. STREAM's memory usage is set in advance as a configuration parameter. In these experiments, we set this parameter to 1.5 GB for policy (P3) and to 64 MB for the other policies. STREAM runs out of memory for (P3). Setting the parameter higher, e.g. to 2 GB, leads to a memory-related runtime error for (P3), which is also marked with the symbol ‡ in the table. PostgreSQL's memory consumption increases with the time span. It varies from around 400 MB for (P1) and (P5), to 2.5 GB for (P2) and (P6), and to around 4 GB for (P3), for the last value of the time span for which a timeout does not occur.

In the following, we comment on the running times. We first focus on our tool. We observe that the formulas (P1), (P2), and (P5) are roughly equally hard to monitor. This is because their running times are dominated by the evaluation of the subformula of the form $[\omega_a\, a, \tau.\, \blacklozenge_{[0,31)}\, \mathit{withdraw}(u, a) \wedge \mathit{ts}(\tau)](v; u)$, which is

[1] Our prototype, the formulas, and the input data are available as an archive at `http://sourceforge.net/projects/monpoly/files/fmsd-experiments.tgz`.

common to all three formulas. In more detail, the number of tuples satisfying the temporal subformula at a time point is on average $31mn$, where m is the average number of withdrawals per day of a user and n is the number of users. This size is significantly larger than the size of the relations corresponding to the additional subformulas in (P2) and (P5). For (P2), at each time point, on average the relations for $\mathit{limit\_on}(u)$ and $\mathit{limit\_off}(u)$ contain $(n/10)/2$ tuples each and the relation for $\neg \mathit{limit\_off}(u)\ \mathsf{S}\ \mathit{limit\_on}(u)$ contains $n/2$ tuples, because the limit flag is toggled for each user on average every 10 days. For (P5), the outer aggregation operator AVG is applied to a relation of average size n. Note that in general the nesting of aggregation operators does not have a substantial impact on the running times, since aggregating over a relation does not increase its size.

For formulas (P3), (P4), and (P6), the main impact on the running times is due to the computation of the natural join $[\![\varphi]\!] \bowtie [\![\psi]\!]$, where φ and ψ denote the two main conjuncts in the formalization of the formulas (P2), (P3), (P4), and (P6). The formula (P3) is slower to monitor than (P2) because the natural join can be optimized when $\mathit{fv}(\psi) \subseteq \mathit{fv}(\varphi)$, which is the case for (P2) but not for (P3). This remark also applies to (P4) and (P6). The relations for the additional subformulas in (P3) are also larger than in (P2): on average, the relation for $\mathit{limit}(u, \ell)$ contains $n/10$ tuples and the relation for $\neg\exists \ell'.\, \mathit{limit}(u, \ell')\ \mathsf{S}\ \mathit{limit}(u, \ell)$ contains n tuples because the limits are changed on average every 10 days for each user. The formula (P4) takes longer to monitor than (P3) because it uses a significantly larger time window. Finally, (P6) takes significantly longer to monitor than (P4) because the input and output relations of the main join operator are also larger. For (P3) and (P4), the two input relations and the output relation each have size n. For (P6), the sizes of the input relations are on average n and $31mn$ while the output relation is on average of size $31mn$.

PostgreSQL performs worst in these experiments. This is not surprising as PostgreSQL was not designed for this application domain. In particular, PostgreSQL has no support for temporal reasoning and we treat time as just another data value, as explained in Section 4.1. Treating time as data has the following disadvantages. First, it is not suited for the online event processing: query evaluation does not scale because the database grows over time and the query must be reevaluated on the entire database each time new events are added. Second, even for offline processing (as done in our experiments), the query evaluation procedure does not take advantage of the temporal ordering of events. This deficiency is most evident when evaluating the SQL queries for the formulas (P2) and (P3). We note that while PostgreSQL is faster on (P3) than on (P2), it consumes significantly more memory for (P3) than for (P2).

In contrast to PostgreSQL, STREAM is designed for online event processing and its running times, except for policy (P3), are consistently better than those of our tool. For the policy (P3), we have bypassed STREAM's default temporal reasoning by treating time as data, and we observe a very high memory consumption, as is the case with PostgreSQL. We also remark that the extension needed to go from formalizing (P1) to formalizing (P2) has a larger impact on STREAM's performance than on our tool. This is because extending the CQL query for (P1) requires a workaround, which does not use the sliding window construct.

Even though STREAM generally outperforms our tool, the performance differences are not as significant as one might expect. One reason why our tool is slower is because it must account for MFOTL$_\Omega$'s underlying time model, which

is more complex than CQL's. MFOTL$_\Omega$ has also a richer tool set than CQL to express temporal patterns.

5 Conclusion

Existing logic-based policy monitoring approaches offer little support for aggregations. To rectify this shortcoming, we extended metric first-order temporal logic with expressive SQL-like aggregation operators and presented a monitoring algorithm for this language. Our experimental results for a prototype implementation of the algorithm are promising. The prototype's performance is in the reach of optimized stream-processing tools, despite its richer input language and its lack of systematic optimization. As future work, we will investigate performance optimizations for our monitor. In general, it remains to be seen how logic-based monitoring approaches can benefit from the techniques used in stream processing.

Acknowledgements This work was partially supported by the Zurich Information Security and Privacy Center (ZISC).

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
2. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.
3. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–144, 2006.
4. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lect. Notes Comput. Sci.*, pages 44–57, 2004.
5. D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. MONPOLY: Monitoring usage-control policies. In *Proceedings of the 2nd International Conference on Runtime Verification (RV'11)*, volume 7186 of *Lect. Notes Comput. Sci.*, pages 360–364, 2012.
6. D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. Monitoring data usage in distributed systems. *IEEE Trans. Software Eng.*, 39(10):1403–1426, 2013.
7. D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring of temporal first-order properties with aggregations. In *Proceedings of the 4th International Conference on Runtime Verification (RV'13)*, volume 8174 of *Lect. Notes Comput. Sci.*, pages 40–58, 2013.
8. D. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *Proceedings of the 28th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'08)*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–60, 2008.
9. D. Basin, F. Klaedtke, and E. Zălinescu. Algorithms for monitoring real-time properties. In *Proceedings of the 2nd International Conference on Runtime Verification (RV'11)*, volume 7186 of *Lect. Notes Comput. Sci.*, pages 260–275, 2012.
10. A. Bauer, R. Goré, and A. Tiu. A first-order policy language for history-based transaction monitoring. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC'09)*, volume 5684 of *Lect. Notes Comput. Sci.*, pages 96–111, 2009.
11. D. Bianculli, C. Ghezzi, and P. S. Pietro. The tale of SOLOIST: A specification language for service compositions interactions. In *Proceedings of the 9th International Symposium on Formal Aspects of Component Software (FACS'12)*, volume 7684 of *Lect. Notes Comput. Sci.*, pages 55–72, 2013.

12. J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.
13. J. Chomicki, D. Toman, and M. H. Böhlen. Querying ATSQL databases with temporal logic. *ACM Trans. Database Syst.*, 26(2):145–178, 2001.
14. C. Colombo, A. Gauci, and G. J. Pace. LarvaStat: Monitoring of statistical properties. In *Proceedings of the 1st International Conference on Runtime Verification (RV'10)*, volume 6418 of *Lect. Notes Comput. Sci.*, pages 480–484, 2010.
15. C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 647–651, 2003.
16. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime monitoring of synchronous systems. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174, 2005.
17. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting statistics over runtime executions. *Form. Method. Syst. Des.*, 27(3):253–274, 2005.
18. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems: The complete book*. Pearson Education, 2009.
19. S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Trans. Serv. Comput.*, 5(2):192–206, 2012.
20. L. Hella, L. Libkin, J. Nurmonen, and L. Wong. Logics with aggregate operators. *J. ACM*, 48(4):880–907, 2001.
21. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, 1990.
22. O. Owe. Partial logics reconsidered: A conservative approach. *Form. Asp. Comput.*, 5(3):208–223, 1993.
23. PostgreSQL Global Development Group. PostgreSQL, Version 9.1.4, 2012. `http://www.postgresql.org/`.
24. A. P. Sistla and O. Wolfson. Temporal conditions and integrity constraints in active database systems. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 269–280, 1995.