A Formally Verified, Optimized Monitor for Metric First-Order Dynamic Logic

David Basin, Thibault Dardinier, Lukas Heimes, Srđan Krstić(✉), Martin Raszyk(✉), Joshua Schneider(✉), and Dmitriy Traytel(✉)

Institute of Information Security, Department of Computer Science, ETH Zürich, Zurich, Switzerland
{srdan.krstic,martin.raszyk,joshua.schneider,traytel}@inf.ethz.ch

Abstract. Runtime monitors for rich specification languages are sophisticated algorithms, especially when they are heavily optimized. To gain trust in them and safely explore the space of possible optimizations, it is important to verify the monitors themselves. We describe the development and correctness proof in Isabelle/HOL of a monitor for metric first-order dynamic logic. This monitor significantly extends previous work on formally verified monitors by supporting aggregations, regular expressions (the dynamic part), and optimizations including multi-way joins adopted from databases and a new sliding window algorithm.

1 Introduction

As the complexity of IT systems increases, so does the complexity and importance of their verification. Research in runtime verification (RV) has developed well-established formal techniques that can often be applied more easily than traditional formal methods such as model checking. RV is based on dynamic analysis, trading off completeness for efficiency. It is mechanized using *monitors*, which are algorithms that search sequences of events, either offline from log files or online, for patterns indicating faults.

Monitors must be trusted when they are used as verifiers. This trust can be justified by checking the monitors themselves for correctness [16,17,31,36,41,42,44,45,49]. Recently, a simplified version of the algorithm used in the MonPoly tool [8,9] has been formalized and proved correct in Isabelle/HOL [45] (Sect. 2). MonPoly and its formal counterpart, called VeriMon, are both monitors for metric first-order temporal logic (MFOTL). However, VeriMon only supports a restricted fragment of this logic and lacks many optimizations that are necessary for an acceptable and competitive performance.

We present a formally verified monitor, VeriMon+, that substantially extends and improves VeriMon. VeriMon+ closes all expressiveness gaps between MonPoly and VeriMon. It supports aggregation operators like sum and average [7] similar to those found in database query languages, arbitrary negations of closed formulas, the unbounded $\bigcirc$ (Next) operator, and constraints involving terms (e.g., $P(x) \wedge y = x + 2$). Due to space limitations, our focus (Sect. 3) will be

N. Peltier and V. Sofronie-Stokkermans (Eds.): IJCAR 2020, LNAI 12166, pp. 432–453, 2020.
https://doi.org/10.1007/978-3-030-51074-9_25

primarily on aggregations, our largest addition. Moreover, VeriMon+ exceeds MFOTL in expressiveness by featuring a significantly richer specification language, metric first-order *dynamic* logic (MFODL). To our knowledge, it is the first monitor for MFODL with past and bounded future operators (Sect. 4). This logic combines MFOTL with regular expressions, similar to linear dynamic logic [22] but enriched with metric constraints, aggregations, and first-order quantification.

We have also implemented and proved correct several new optimizations. First, to speed up the evaluation of conjunctions, we integrated an efficient algorithm for multi-way joins [38,39], which we generalized to include anti-joins (Sect. 5). Second, we developed a specialized sliding window algorithm to evaluate the Since and Until operators more efficiently (Sect. 6). VeriMon+ is executable via the generation of OCaml code from Isabelle. To this end, we augmented the code generation setup for IEEE floating point numbers in OCaml [50] with a linear ordering, which is needed for efficient set and mapping data structures.

The result of our efforts is both a verified monitor and a tool for evaluating unverified monitors. Since MFODL is extremely expressive, this gives us very wide scope. For example, we discovered previously unknown bugs in MonPoly via differential testing (Sect. 7), extending a previous case study [45]. As this experience suggests, and we firmly believe, formal verification is the most reliable way to obtain correct, optimized monitors.

In sum, our main contribution is a verified monitor for MFODL with aggregations, a highly expressive specification language that combines regular expressions and first-order temporal logic. Our monitor includes optimizations that are novel in the context of first-order monitoring. Our formalization is publicly available [20,21].

Related Work. We refer to a recent book [4] for an introduction to runtime verification. The main families of specification languages in this domain are extensions of LTL [13,26,46], automata [3], stream expressions [19], and rule systems [23]. We combine two expressive temporal logics and their corresponding monitoring algorithms. MFOTL, implemented in MonPoly [7–9], supports first-order quantification over parametrized events, but it cannot express all regular patterns. Metric dynamic logic (MDL), implemented in Aerial [12], supports regular expressions, but it is not first-order. VeriMon+ is based on VeriMon [45], which only supports a fragment of MFOTL and is inefficient (Sect. 7). We refer to [45, Section 1] for an overview of related monitor formalizations. Relational database systems have been formalized by Malecha et al. [33] and by Benzaken et al. [15]. These works use binary joins only, which are not worst-case optimal.

Another efficient first-order monitor, DejaVu [25], supports past-only first-order temporal logic. It uses binary decision diagrams (BDDs) and does not restrict the use of negation, unlike MonPoly, which uses finite tables. DejaVu's performance is incomparable to MonPoly's and it is unclear whether multi-way joins can improve conjunctions of BDDs. Aerial and VeriMon+ evaluate regular expressions using derivatives [2,18], which also have been used for timed regular

datatype *data* = Int *int* | Flt *double* | Str *string* **type_synonym** *ts* = *nat*
type_synonym *db* = (*string* × *data list*) *set* **typedef** *trace* = {*s* :: (*db* × *ts*) *stream*. trace *s*}
datatype *trm* = V *nat* | C *data* | *trm* + *trm* | … **typedef** $\mathcal{I}$ = {(*a* :: *nat*, *b* :: *enat*). *a* ≤ *b*}

datatype *frm* = *string*(*trm list*) | *trm* ≈ *trm* | *trm* ≺ *trm* | *trm* ⪯ *trm*
| ¬ *frm* | ∃ *frm* | *frm* ∨ *frm* | *frm* ∧ *frm* | ●$_\mathcal{I}$ *frm* | ○$_\mathcal{I}$ *frm* | *frm* S$_\mathcal{I}$ *frm* | *frm* U$_\mathcal{I}$ *frm*

fun etrm :: *data list* ⇒ *trm* ⇒ *data* **where**
etrm *v* (V *x*) = *v* ! *x* | etrm *v* (C *x*) = *x* | etrm *v* (*t*$_1$ + *t*$_2$) = etrm *v* *t*$_1$ + etrm *v* *t*$_2$ | …

fun sat :: *trace* ⇒ *data list* ⇒ *nat* ⇒ *frm* ⇒ *bool* **where**
sat σ *v* *i* (*r*(*ts*)) = ((*r*, map (etrm *v*) *ts*) ∈ Γ σ *i*) | sat σ *v* *i* (*t*$_1$ ≈ *t*$_2$) = (etrm *v* *t*$_1$ = etrm *v* *t*$_2$)
| sat σ *v* *i* (*t*$_1$ ≺ *t*$_2$) = (etrm *v* *t*$_1$ < etrm *v* *t*$_2$) | sat σ *v* *i* (*t*$_1$ ⪯ *t*$_2$) = (etrm *v* *t*$_1$ ≤ etrm *v* *t*$_2$)
| sat σ *v* *i* (¬φ) = (¬sat σ *v* *i* φ) | sat σ *v* *i* (∃φ) = (∃*z*. sat σ (*z*#*v*) *i* φ)
| sat σ *v* *i* (α ∨ β) = (sat σ *v* *i* α ∨ sat σ *v* *i* β) | sat σ *v* *i* (α ∧ β) = (sat σ *v* *i* α ∧ sat σ *v* *i* β)
| sat σ *v* *i* (●$_I$ φ) = (case *i* of 0 ⇒ False | *j* + 1 ⇒ T σ *i* − T σ *j* ∈$_\mathcal{I}$ *I* ∧ sat σ *v* *j* φ)
| sat σ *v* *i* (○$_I$ φ) = (T σ (*i* + 1) − T σ *i* ∈$_\mathcal{I}$ *I* ∧ sat σ *v* (*i* + 1) φ)
| sat σ *v* *i* (α S$_I$ β) = (∃*j* ≤ *i*. T σ *i* − T σ *j* ∈$_\mathcal{I}$ *I* ∧ sat σ *v* *j* β ∧ (∀*k* ∈ {*j* <.. *i*}. sat σ *v* *k* α))
| sat σ *v* *i* (α U$_I$ β) = (∃*j* ≥ *i*. T σ *j* − T σ *i* ∈$_\mathcal{I}$ *I* ∧ sat σ *v* *j* β ∧ (∀*k* ∈ {*i* ..< *j*}. sat σ *v* *k* α))

Fig. 1. Syntax and semantics of MFOTL as presented in [45], with additions in gray

expressions [47]. Quantified regular expressions [1,34] extend regular expressions with data and aggregations. They can be evaluated efficiently, but can neither express metric constraints nor future modalities directly.

2 A Verified Monitor for Metric First-Order Temporal Logic

VeriMon [45] is a formally verified monitor for a large fragment of MFOTL [8]. The monitor takes an MFOTL formula, which may be open, and incrementally processes an infinite stream of time-stamped events. It outputs for every stream position the set of variable assignments that satisfy the formula. Thus, the monitor can be used to extract data from the stream. Typically, one is interested in the violations of a property specified as an MFOTL formula, which can be obtained by monitoring the negated formula.

We give an overview of MFOTL and VeriMon. We also cover some of the smaller additions in our new monitor, VeriMon+, highlighted in gray. For readability, we liberally use abbreviations and symbolic notation, departing mildly from Isabelle's syntax.

Figure 1 shows MFOTL's syntax and semantics. Events have a name (*string*) and a list of parameters of type *data*. In VeriMon+, *data* is a disjoint union of integers, double-precision floats, and strings. Multiple events are grouped together into a database (*db*) if they are considered to occur simultaneously. We call an infinite stream of databases, augmented with their corresponding time-stamps, an event stream or *trace*. Time-stamps (*ts*) are modeled as natural numbers (*nat*). We write T σ *i* to denote the time-stamp of the *i*th database Γ σ *i* of the event stream σ. The predicate trace expresses that the time-stamps

are monotone, i.e., $\mathsf{T}\ \sigma\ i \leq \mathsf{T}\ \sigma\ (i{+}1)$ for all $i \geq 0$, and always eventually strictly increasing, i.e., $\forall t.\ \exists i.\ t < \mathsf{T}\ \sigma\ i$. Consecutive time-points i can have the same time-stamp.

Terms and formulas are represented by the datatypes *trm* and *frm*, respectively. Our formalization uses de Bruijn indices for free and bound variables (constructor V). In examples, we prefer the standard named syntax (and omit V). The type $\mathcal{I}$ models nonempty, possibly unbounded intervals over *nat*. We write $n \in_{\mathcal{I}} I$ for n's membership in I, and $[a, b]$ for the unique interval satisfying $n \in_{\mathcal{I}} [a, b]$ iff $a \leq n \leq b$. The right bound b is of type *enat*, i.e., either a natural number or infinity ∞ for an unbounded interval.

The functions etrm and sat (Fig. 1) define MFOTL's semantics. Both take a variable assignment v, which is a list of type *data list* whose ith element $v\ !\ i$ is the value assigned to the variable with index i. The function etrm evaluates terms under a given assignment. The expression $\mathsf{sat}\ \sigma\ v\ i\ \varphi$ is true iff the formula φ is satisfied by v at time-point i in the trace σ. VeriMon+ adds arithmetic operators and type conversions to terms, as well as the predicates $\prec$ and $\preceq$. Their semantics on *data* is lifted from the corresponding operations on integers, floats, and strings, whenever they are meaningful. The ordering $\leq$ on *data* is total: strings are compared lexicographically and $\mathsf{Int}\ i < \mathsf{Flt}\ f < \mathsf{Str}\ s$.

VeriMon computes sets of satisfactions (i.e., satisfying assignments) by recursion over the formula's structure. It represents these sets as finite tables, to which it applies standard relational operations such as the natural join ($\bowtie$) and union. Tables are sets of tuples, which are lists of optional *data* values; missing values are denoted by $\bot$. This representation allows us to use tuples with the same length across subformulas with different free variables. The predicate $\mathsf{wf\_tuple}$ defines the well-formed tuples for a given length n and a set of variables V. We also refer to V as the columns of a tuple (or table).

definition $\mathsf{wf\_tuple} :: \mathit{nat} \Rightarrow \mathit{nat\ set} \Rightarrow \mathit{tuple} \Rightarrow \mathit{bool}$ **where**
$\mathsf{wf\_tuple}\ n\ V\ v = (\mathsf{length}\ v = n \wedge (\forall x < n.\ v\ !\ x = \bot \longleftrightarrow x \notin V))$

The set of satisfactions may be infinite. VeriMon supports only a fragment of MFOTL for which all computed tables are finite. The predicate safe (omitted) defines the monitorable fragment [45]. It accepts only certain combinations of operators and constrains the free variables of subformulas. Also, the intervals of all U operators must be bounded.

VeriMon's interface consists of two functions $\mathsf{init} :: \mathit{frm} \Rightarrow \mathit{mstate}$ and $\mathsf{step} :: \mathit{db} \times \mathit{ts} \Rightarrow \mathit{mstate} \Rightarrow (\mathit{nat} \times \mathit{table})\ \mathit{list} \times \mathit{mstate}$. The former initializes the monitor's state, and the latter updates it with a new time-stamped database to report any new satisfactions. We require that satisfactions are reported for every time-point and in order. Note that a formula containing a future operator such as U cannot necessarily be evaluated at time-point i after observing the ith database. Therefore, the output for several time-points may become available at once, so step returns a list of pairs of time-points and tables.

We describe the evaluation of $\alpha\, \mathsf{S}_{[a,\,b]}\, \beta$ in more detail. This formula is equivalent to the disjunction of $\alpha\, \mathsf{S}_{[c,\,c]}\, \beta$ for all c such that $a \leq c \leq b$. Suppose that the most recent time-point is i with time-stamp τ. The monitor's state for $\alpha\, \mathsf{S}_{[a,\,b]}\, \beta$

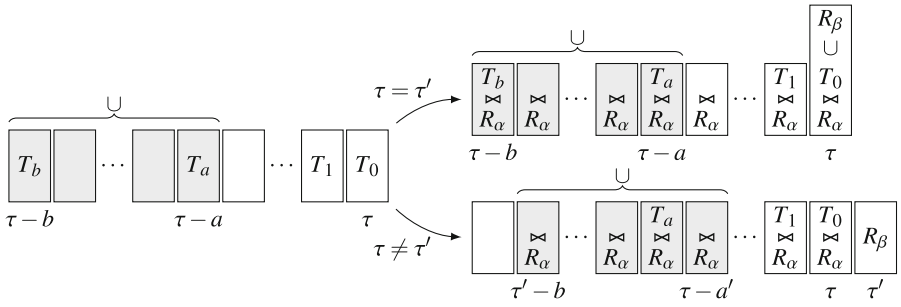

Fig. 2. Simplified state of a Since operator and its update

consists of a list of tables T_c with the satisfactions of $\alpha \,\mathsf{S}_{[c,\,c]}\, \beta$ at time-point i, along with the corresponding time-stamps $\tau - c$. VeriMon also stores the satisfactions T_c (and time-stamps) for $0 \leq c < a$, which are not yet in the interval. Figure 2 (left) depicts a state, where we assume for simplicity that we store a table for every time-stamp between $\tau - b$ and τ. (In reality, time-stamps not in the trace do not have a corresponding entry in this list.) The state is updated for every new time-point with time-stamp τ', for which we already know the satisfactions R_α and R_β of the subformulas α and β. In Fig. 2, we distinguish whether τ' equals τ (otherwise $\tau' > \tau$ by monotonicity). The update consists of three steps: (1) remove tables that fall out of the interval; (2) evaluate the conjunction of each remaining table with R_α using a relational join; and (3) add the new tuples from R_β, either by inserting them into the most recent table T_0 or by adding a new table, depending on whether τ' equals τ. Finally, we take the union of all tables within the interval to obtain the satisfactions of $\alpha \,\mathsf{S}_{[a,b]}\, \beta$.

We summarize VeriMon's correctness, which we also prove for VeriMon+. It relates the monitor's implementation to its specification $\mathsf{verdicts} :: \mathit{frm} \Rightarrow (\mathit{db} \times \mathit{ts})\ \mathit{list} \Rightarrow (\mathit{nat} \times \mathit{tuple})\ \mathit{set}$, which defines the expected output on a stream prefix. The first result shows that verdicts characterizes an MFOTL monitor, where $\mathsf{prefix}\ \pi\ \sigma$ means that π is a prefix of σ, and $\mathsf{map\ the}\ v$ converts v to an assignment by mapping $\bot$ to an unspecified value.

Lemma 1 ([45], Lemma 2). *Suppose that* safe φ *is true. Then,* verdicts φ *is sound and eventually complete, i.e., for all prefixes* π *of trace* σ*, time-points* i*, and tuples* v*,*

(a) $(i, v) \in \mathsf{verdicts}\ \varphi\ \pi \longrightarrow \mathsf{sat}\ \sigma\ (\mathsf{map\ the}\ v)\ i\ \varphi$*, and*
(b) $i < \mathsf{length}\ \pi \wedge \mathsf{wf\_tuple}\ (\mathsf{nfv}\ \varphi)\ (\mathsf{fv}\ \varphi)\ v \wedge (\forall \sigma'.\ \mathsf{prefix}\ \pi\ \sigma' \longrightarrow \mathsf{sat}\ \sigma'\ (\mathsf{map\ the}\ v)\ i\ \varphi) \longrightarrow (\exists \pi'.\ \mathsf{prefix}\ \pi'\ \sigma \wedge (i, v) \in \mathsf{verdicts}\ \varphi\ \pi')$.

Above, nfv φ is the smallest number larger than all free variables of φ, written fv φ. The next result establishes the implementation's correctness using the state invariant $\mathsf{wf\_mstate} :: \mathit{frm} \Rightarrow (\mathit{db} \times \mathit{ts})\ \mathit{list} \Rightarrow \mathit{mstate} \Rightarrow \mathit{bool}$ (omitted). Let set convert lists into sets, $\mathsf{last\_ts}\ \pi$ be the last time-stamp in π, and $\pi_1 @ \pi_2$ be the concatenation of π_1 and π_2.

Theorem 1 ([45], Theorem 1). *The initialization* init *establishes the invariant and the update* step *preserves the invariant and its output can be described in terms of* verdicts*:*

(a) If safe φ*, then* wf_mstate φ [] (init φ)*.*
(b) Let step (db, τ) $mst = (A, mst')$*. If* wf_mstate φ π mst *and* last_ts $\pi \leq \tau$*, then* $(\bigcup(i, V) \in$ set $A.$ $\{(i, v) \mid v \in V\}) =$ verdicts φ $(\pi$ @ $[(db, \tau)]) -$ verdicts φ π *and* wf_mstate φ $(\pi$ @ $[(db, \tau)])$ mst'*.*

3 Aggregations

Basin et al. [7] extended MFOTL with a generic aggregation operator. This operator was inspired by the group-by clause and aggregation functions of SQL. It first partitions the satisfying assignments of its subformula into groups, and then computes a summary value, such as count, sum, or average, for each group. We formalized the aggregation operator's semantics, added an evaluation algorithm to VeriMon+, and proved its correctness.

Consider the formula $s \leftarrow \mathsf{Sum}\ x; x.\ P(g, x)$. The aggregation operator $s \leftarrow \mathsf{Sum}\ x; x$ has four parameters: a result variable (s), the aggregation type (Sum), an aggregation term (first x), and a list of variables that are bound by the operator and thus excluded from grouping (second x). When evaluated, the above formula yields a set of tuples (s, g). There is one such tuple for every value of g with at least one P event that has g's value as its first parameter. The values of g partition the satisfactions of $P(g, x)$ into groups. For every group, the sum over the values of x in that group is assigned to the variable s.

We added the constructor $nat \leftarrow agg\_op\ trm; nat.\ frm$ to frm. Consider the instance $y \leftarrow \Omega\ t; b.\ \varphi$. The operator binds b variables simultaneously in the formula φ and in the term t, over which we aggregate. In examples, we list the bound variables explicitly instead of writing the number b. The remaining free variables (possibly none) of φ are used for grouping. The variable y receives the result of the aggregation operation $\Omega = (\omega, d)$, where ω is one of Cnt (count), Min, Max, Sum, Avg (average), or Med (median). The default value d, which we usually omit, determines the result for empty groups (e.g., 0 for Cnt). The formula's free variables are those of φ excluding the b bound variables, plus y.

Figure 3 shows the semantics of the aggregation operator $y \leftarrow \Omega\ t; b.\ \varphi$. The assignment v determines both a group and a candidate value $v\,!\,y$ for the aggregation's result on that group. The sat function checks whether the value is correct. First, it computes the set M, which encodes a multiset in the form of pairs (x, c), where c is x's multiplicity. This multiset contains the values of the term t under all assignments z @ v that satisfy φ, where z is an assignment to the bound variables. The expression $\mathsf{card}^{\infty}\ Z$ stands for the cardinality of Z when it is finite, and ∞ otherwise. Then, sat compares $v\,!\,y$ to the result of the aggregation operation Ω on M, which is given by agg_op Ω M (omitted).

We extended the safe predicate with sufficient conditions that describe when the aggregation formula $y \leftarrow \Omega\ t; b.\ \varphi$ has finitely many satisfactions. We require that φ satisfies safe, that the variable y is not free in φ excluding the b bound variables, and that all bound variables and the variables in t occur free in φ. We

$$\begin{aligned}
&\textbf{fun}\ \mathsf{sat} :: \mathit{trace} \Rightarrow \mathit{data\ list} \Rightarrow \mathit{nat} \Rightarrow \mathit{frm} \Rightarrow \mathit{bool}\ \textbf{where}\ \ldots\\
&\mid \mathsf{sat}\ \sigma\ v\ i\ (y \leftarrow \Omega\ t; b.\ \varphi) = \big(\underline{\mathsf{let}}\ M = \{(x, \mathsf{card}^{\infty}\ Z) \mid x\ Z.\\
&\qquad Z = \{z.\ \mathsf{length}\ z = b \wedge \mathsf{sat}\ \sigma\ (z\ @\ v)\ i\ \varphi \wedge \mathsf{etrm}\ (z\ @\ v)\ t = x\} \wedge Z \neq \{\}\}\\
&\quad \underline{\mathsf{in}}\ (M = \{\} \longrightarrow \mathsf{fv}\ \varphi \subseteq \{0\ ..{<}\ b\}) \wedge v\,!\,y = \mathsf{agg_op}\ \Omega\ M)\\
&\textbf{fun}\ \mathsf{eval_agg} :: \mathit{nat} \Rightarrow \mathit{bool} \Rightarrow \mathit{nat} \Rightarrow \mathit{agg_op} \Rightarrow \mathit{nat} \Rightarrow \mathit{trm} \Rightarrow \mathit{table} \Rightarrow \mathit{table}\ \textbf{where}\\
&\quad \mathsf{eval_agg}\ n\ g_0\ y\ \Omega\ b\ t\ R = \big(\underline{\mathsf{if}}\ g_0 \wedge R = \{\}\ \underline{\mathsf{then}}\ \mathsf{singleton_table}\ n\ y\ (\mathsf{agg_op}\ \Omega\ \{\})\ \underline{\mathsf{else}}\\
&\qquad (\lambda k.\ \underline{\mathsf{let}}\ G = \{v \in R \mid \mathsf{drop}\ b\ v = k\};\ M = (\lambda x.\ (x, \mathsf{card}^{\infty}\ \{v \in G \mid \mathsf{meval_trm}\ t\ v = x\}))\ `\\
&\qquad\quad (\mathsf{meval_trm}\ t)\ `\ G\ \underline{\mathsf{in}}\ k[y := \mathsf{Some}\ (\mathsf{agg\_op}\ \Omega\ M)])\ `\ (\mathsf{drop}\ b)\ `\ R\big)
\end{aligned}$$

Fig. 3. Semantics and evaluation of the aggregation operator

adopted the convention [7] that an aggregation formula is not satisfied when M is empty, unless all free variables of φ are bound by the operator. Otherwise, there would be infinitely many groups (and hence, satisfactions) with the aggregate value $\mathsf{agg\_op}\ \Omega\ \{\}$, assuming that φ is safe.

Figure 3 also defines $\mathsf{eval\_agg}$, which evaluates the aggregation operator. It takes a table R with φ's satisfactions, and returns a table with the aggregation operator's satisfactions. The first argument n controls the length of the tuples in the tables (Sect. 2). The argument g_0 specifies whether all free variables of φ are bound by the operator. The remaining arguments y, Ω, b, and t are those of the operator. We write $f\ `\ X$ for the image of X under f.

In $\mathsf{eval\_agg}$, we first check whether $g_0 \wedge R = \{\}$ is true to handle the special case mentioned above. (The expression $\mathsf{singleton\_table}\ n\ y\ a$ is a table with a single tuple of length n that assigns a to variable y.) Otherwise, we compute the aggregate value separately for each group k. The set of groups is obtained by discarding the first b values of each tuple in R. To every group k, we apply the lambda-term to augment the tuple with the aggregate value. The set G contains all tuples in the group. Note that these tuples extend k with assignments to the b bound variables. Then, we compute the image of G under the term t, which is evaluated by $\mathsf{meval\_trm} :: \mathit{trm} \Rightarrow \mathit{tuple} \Rightarrow \mathit{data}$ (omitted). Finally, we obtain the multiset M by counting how many tuples in G map to each value in the image.

4 Regular Expressions

VeriMon+ extends VeriMon's language by generalizing MFOTL's temporal operators to regular expressions. The resulting metric first-order dynamic logic (MFODL) can be seen [24, §3.16] as the "supremum" (in the sense of combining features) of metric dynamic logic (MDL) [12] and MFOTL [8]. Peycheva's master's thesis [40] develops a monitor for past-only MFODL. We give the first formal definition of MFODL with past and future operators. We also define a fragment whose formulas can be evaluated using finite relations (Sect. 4.1). This fragment guides our evaluation algorithm's design (Sect. 4.2).

Figure 4 (left) defines the syntax and semantics of our variant of regular expressions. The type re is parametrized by a type variable $'a$, which is used

datatype $'a\ re = \star^{nat} \mid {'a}? \mid 're + re \mid re \cdot re \mid (re)^*$

fun match :: $(nat \Rightarrow 'a \Rightarrow bool) \Rightarrow 're \Rightarrow nat \otimes nat$ **where**
  match *test* $(\star^k) = \{(i,j) \mid j = i+k\}$
| match *test* $(x?) = \{(i,i) \mid test\ i\ x\}$
| match *test* $(r+s) =$ match *test* $r \cup$ match *test* s
| match *test* $(r \cdot s) =$ match *test* $r \bullet$ match *test* s
| match *test* $(r^*) = ($match *test* $r)^*$

datatype $frm = \ldots \mid \blacktriangleleft_{\mathcal{I}}\ (frm\ re) \mid \triangleright_{\mathcal{I}}\ (frm\ re)$

fun sat :: $trace \Rightarrow data\ list \Rightarrow nat \Rightarrow frm \Rightarrow bool$ **where** ...
| sat $\sigma\ v\ i\ (\blacktriangleleft_I\ r) = (\exists j \le i.\ \mathsf{T}\ \sigma\ i - \mathsf{T}\ \sigma\ j \in_{\mathcal{I}} I \wedge (j,i) \in$ match (sat $\sigma\ v$) $r)$
| sat $\sigma\ v\ i\ (\triangleright_I\ r) = (\exists j \ge i.\ \mathsf{T}\ \sigma\ j - \mathsf{T}\ \sigma\ i \in_{\mathcal{I}} I \wedge (i,j) \in$ match (sat $\sigma\ v$) $r)$

$\bullet_I\ \alpha \rightsquigarrow \blacktriangleleft_I\ (\alpha? \cdot \star)$
$\alpha\ \mathsf{S}_I\ \beta \rightsquigarrow \blacktriangleleft_I\ (\beta? \cdot (\star \cdot \alpha?)^*)$
$\bigcirc_I\ \alpha \rightsquigarrow \triangleright_I\ (\star \cdot \alpha?)$
$\alpha\ \mathsf{U}_I\ \beta \rightsquigarrow \triangleright_I\ ((\alpha? \cdot \star)^* \cdot \beta?)$

Fig. 4. Syntax and semantics of MFODL (left) and conversion of MFOTL into MFODL (right)

in the _? constructor. The semantics is given by match and assigns to each expression a binary relation ($\otimes$) on natural numbers. Intuitively, a pair (i, j) is in the relation assigned to r when r matches the portion of a trace from i to j. The trace notion is abstracted away in match via the argument *test*, which indicates whether a parameter of type $'a$ may advance past a given point.

In more detail, the wildcard operator $\star^k$ matches all pairs (i, j), where $j = i + k$; we write $\star$ for the useful special case $\star^1$. The test $x?$ only matches pairs of the form (i, i) that pass *test* i x. The semantics of alternation ($+$) as union ($\cup$), concatenation ($\cdot$) as relation composition ($\bullet$), and Kleene star (_*) as reflexive-transitive closure (_*) is standard.

Figure 4 (left) also shows *frm*'s extension with two constructors that use regular expressions. The regular expression's parameter nests a recursive occurrence of *frm*, i.e., our regular expressions' leaves are formulas, which in turn may further nest regular expressions, and so on. MDL's syntax is often presented as a mutually recursive datatype [12]. Our nested formulation is beneficial because it lets us formalize regular expressions independently, for use in different applications (e.g., monitors for MDL and MFODL).

In terms of their semantics, the two new operators naturally generalize the S_I and U_I operators. The past match operator $\blacktriangleleft_I\ r$ is satisfied at i if there is an earlier time-point j subject to the same temporal constraint I as in the satisfaction of S_I and moreover the regular expression r matches from j to i. For the future match operator $\triangleright_I\ r$, the situation is symmetric with the existentially quantified j being a future time-point. In both cases, the *test* parameter of match is recursively instantiated with the satisfaction predicate sat.

We can embed MFOTL into MFODL by expressing the temporal operators using semantically equivalent formulas built from regular expressions (Fig. 4, right). Thus, we could in principle remove the operators $\bullet$, S, $\bigcirc$, and U from *frm* and use regular expressions instead. We prefer to keep these operators in *frm* as this allows us to optimize their evaluation in a way that is not available for the more general match operators (Sect. 6).

datatype *context* = past | futu **datatype** *mode* = strict | lax

fun safe :: *context* ⇒ *mode* ⇒ *frm re* ⇒ *bool* **where**
safe _ _ ($\star^k$) = True
| safe _ *m* (($\neg\varphi$)?) = (*m* = lax ∧ safe φ)
| safe _ _ (φ?) = safe φ
| safe *c m* (*r* + *s*) = ((*m* = lax ∨ fv *r* = fv *s*) ∧ safe *c m r* ∧ safe *c m s*)
| safe futu *m* (*r* · *s*) = ((*m* = lax ∨ fv *r* ⊆ fv *s*) ∧ safe futu lax *r* ∧ safe futu *m s*)
| safe past *m* (*r* · *s*) = ((*m* = lax ∨ fv *s* ⊆ fv *r*) ∧ safe past *m r* ∧ safe past lax *s*)
| safe *c m* (r^*) = (*m* = lax ∧ safe *c m r*)

fun safe :: *frm* ⇒ *bool* **where** …
| safe ($\blacktriangleleft_I$ *r*) = safe past strict *r* | safe ($\triangleright_{[a,b]}$ *r*) = safe futu strict *r* ∧ *b* < ∞

Fig. 5. Safety conditions for MFODL

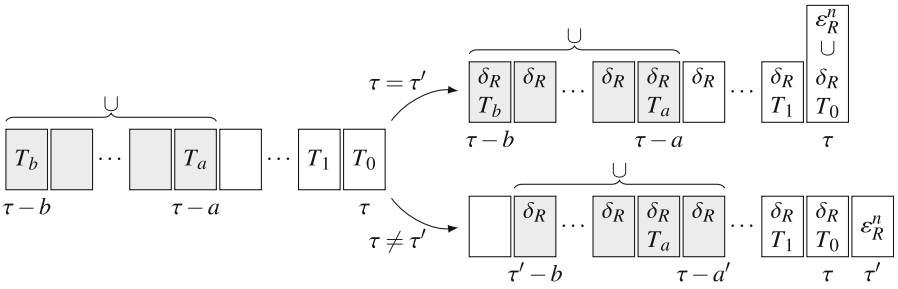

Fig. 6. Simplified state of a past match operator and its update

We conclude MFODL's introduction with an example. Many systems for user authentication follow a policy like: "A user should not be able to authenticate after entering the wrong password three times in a row within the last 10 minutes." We write ✗(u) for the event "User u entered the wrong password" and ✓(u) for "User u has successfully authenticated." Additionally, we abbreviate $\varphi? \cdot \star$ by φ. (This abbreviation is only used when φ appears in a regular expression position, e.g., as an argument of $\cdot$). Then the formula

$$✓(u) \wedge \blacktriangleleft_{[0,600]} \left(✗(u) \cdot (\neg✓(u))^* \cdot ✗(u) \cdot (\neg✓(u))^* \cdot ✗(u) \cdot (\neg✓(u))^*\right)$$

expresses this policy's violations: its satisfying assignments are precisely the users that successfully authenticate after entering wrong credentials for three times in the last 600 seconds, without intermediate successful authentications. We can express this property in MFOTL using three nested S operators, one for each of the ✗(u) subformulas. Yet, it is unclear which intervals to put as arguments to S beyond the fact that they should sum up to 600. The rather impractical solution exploits that there are only finitely many ways to split the intervals due to their bounds being natural numbers and constructs the disjunction of all possible splits (180 901 in this case). MFODL remediates this infeasible construction.

4.1 Finitely Evaluable Regular Expressions

Following MonPoly's design [8], VeriMon+ represents all sets of satisfying assignments with finite tables. The databases occurring in the trace are all finite, yet their combination may not be. Therefore, MonPoly and VeriMon+ work with syntactic restrictions that ensure that all sets that arise are finite. For example, negation must occur under a conjunction $\alpha \wedge \neg\beta$, where the free variables of β, written $\mathsf{fv}\ \beta$, are contained in those of α. We say that $\neg\beta$ is guarded by α and compute $\alpha \wedge \neg\beta$ as the anti-join ($\triangleright$) of the corresponding tables. For disjunctions $\alpha \vee \beta$, we must have $\mathsf{fv}\ \alpha = \mathsf{fv}\ \beta$. Similar restrictions also apply to temporal operators: to evaluate $\alpha\ \mathsf{S}_I\ \beta$ and $\alpha\ \mathsf{U}_I\ \beta$ we require $\mathsf{fv}\ \alpha \subseteq \mathsf{fv}\ \beta$.

We derive a new sufficient criterion for match operators to have finitely many satisfying assignments. To develop some intuition, we first consider several examples that result in infinite tables. The first example is any expression with a Kleene star as the topmost operator. The formula $\varphi = \blacktriangleleft_{[0,b]}\ (r^*)$ is satisfied at all points i for all assignments v (regardless of r's free variables) since $0 \in_{\mathcal{I}} I$ and any (i, i) matches r^*. Thus, when we evaluate φ at i, we can choose i as the witness for the existential quantifier in the definition of sat. It follows that Kleene stars must be guarded by a finite table.

The union of two finite tables is finite only if the tables have the same columns (assuming an infinite domain *data*). This explains the requirement for the subformulas of $\vee$ to have the same variables, but a similar requirement is needed for the $+$ of regular expressions. Perhaps more surprisingly, concatenation can also hide a union: consider $\varphi = \blacktriangleleft_{[0,b]}\ (r \cdot s^*)$ and assume that s matches (j, i) for some $j < i$. By the semantics of concatenation, we can split the satisfactions of φ into those that use s^*'s matching pair (i, i) (i.e., the satisfying assignments of $\blacktriangleleft_I\ r$ at i) and those that do not. To combine these assignments it seems necessary to take the union of the satisfaction of φ and $\blacktriangleleft_I\ r$, which in turn requires these formulas to have the same free variables, or equivalently $\mathsf{fv}\ s \subseteq \mathsf{fv}\ r$ (overloading notation to apply fv to regular expression). The future match operator behaves symmetrically, requiring the side condition $\mathsf{fv}\ r \subseteq \mathsf{fv}\ s$ for $\triangleright_{[0,b]}\ (r^* \cdot s)$.

MonPoly also allows the left subformula of S and U to be negated: $(\neg\alpha)\ \mathsf{S}_I\ \beta$ and $(\neg\alpha)\ \mathsf{U}_I\ \beta$. Hence, we should support the MFODL variants $\blacktriangleleft_I\ (\beta? \cdot (\star \cdot (\neg\alpha)?)^*)$ and $\triangleright_I\ (((\neg\alpha)? \cdot \star)^* \cdot \beta?)$, but also generalize these patterns to flexibly support negated tests.

Our solution to these issues comprises the predicates shown in Fig. 5. The safe predicate on regular expressions is parametrized by two flags: *context* distinguishing whether the expression occurs under a past or a future match operator and *mode* determining whether the tests may be negated and other safety conditions relaxed. The most interesting cases are those for concatenation. There, in addition to the fv side conditions, only one argument is checked recursively in the same mode as the overall expression. The other argument is checked using the lax mode, in which side conditions are skipped, except for the requirement that (possibly negated) formulas under the test operators are safe. The context parameter dictates which argument keeps, and which changes, the mode.

$$
\begin{array}{l}
\textbf{fun } \varepsilon_{\mathsf{lax}} :: \mathit{table} \Rightarrow (\mathit{frm} \Rightarrow \mathit{table}) \\
\qquad \Rightarrow \mathit{frm}\ \mathit{re} \Rightarrow \mathit{table}\ \textbf{where} \\
\quad \varepsilon_{\mathsf{lax}}\ X\ R\ (\star^k) = (\underline{\mathsf{if}}\ k = 0\ \underline{\mathsf{then}}\ X\ \underline{\mathsf{else}}\ \{\}) \\
\mid \varepsilon_{\mathsf{lax}}\ X\ R\ ((\neg\varphi)?) = X \triangleright R\ \varphi \\
\mid \varepsilon_{\mathsf{lax}}\ X\ R\ (\varphi?) = X \bowtie R\ \varphi \\
\mid \varepsilon_{\mathsf{lax}}\ X\ R\ (r + s) = \varepsilon_{\mathsf{lax}}\ X\ R\ r \cup \varepsilon_{\mathsf{lax}}\ X\ R\ s \\
\mid \varepsilon_{\mathsf{lax}}\ X\ R\ (r \cdot s) = \varepsilon_{\mathsf{lax}}\ X\ R\ r \bowtie \varepsilon_{\mathsf{lax}}\ X\ R\ s \\
\mid \varepsilon_{\mathsf{lax}}\ X\ R\ (r^*) = X
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun } \varepsilon_{\mathsf{strict}} :: \mathit{nat} \Rightarrow (\mathit{frm} \Rightarrow \mathit{table}) \\
\qquad \Rightarrow \mathit{frm}\ \mathit{re} \Rightarrow \mathit{table}\ \textbf{where} \\
\quad \varepsilon_{\mathsf{strict}}\ n\ R\ (\star^k) = \\
\qquad (\underline{\mathsf{if}}\ k = 0\ \underline{\mathsf{then}}\ \{\overbrace{(\bot, \ldots, \bot)}^{n}\}\ \underline{\mathsf{else}}\ \{\}) \\
\mid \varepsilon_{\mathsf{strict}}\ n\ R\ (\varphi?) = R\ \varphi \\
\mid \varepsilon_{\mathsf{strict}}\ n\ R\ (r + s) = \varepsilon_{\mathsf{strict}}\ n\ R\ r \cup \varepsilon_{\mathsf{strict}}\ n\ R\ s \\
\mid \varepsilon_{\mathsf{strict}}\ n\ R\ (r \cdot s) = \varepsilon_{\mathsf{lax}}\ (\varepsilon_{\mathsf{strict}}\ n\ R\ r)\ R\ s
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun } \delta :: (\mathit{frm}\ \mathit{re} \Rightarrow \mathit{frm}\ \mathit{re}) \Rightarrow (\mathit{frm} \Rightarrow \mathit{table}) \Rightarrow (\mathit{frm}\ \mathit{re} \Rightarrow \mathit{table}) \Rightarrow \mathit{frm}\ \mathit{re} \Rightarrow \mathit{table}\ \textbf{where} \\
\quad \delta\ \kappa\ R\ T\ (\star^k) = (\underline{\mathsf{if}}\ k = 0\ \underline{\mathsf{then}}\ \{\}\ \underline{\mathsf{else}}\ T\ (\kappa\ (\star^{k-1}))) \\
\mid \delta\ \kappa\ R\ T\ (\varphi?) = \{\} \\
\mid \delta\ \kappa\ R\ T\ (r + s) = \delta\ \kappa\ R\ T\ r \cup \delta\ \kappa\ R\ T\ s \\
\mid \delta\ \kappa\ R\ T\ (r \cdot s) = \delta\ (\lambda t.\ \kappa\ (r \cdot t))\ R\ T\ s \cup \varepsilon_{\mathsf{lax}}\ (\delta\ \kappa\ R\ T\ r)\ R\ s \\
\mid \delta\ \kappa\ R\ T\ (r^*) = \delta\ (\lambda t.\ \kappa\ (r^* \cdot\ t))\ R\ T\ r
\end{array}
$$

Fig. 7. The core evaluation functions for MFODL

4.2 Evaluation Algorithm

The evaluation algorithm's structure for the past match operator $\blacktriangleleft_I\ r$ (Fig. 6) closely resembles the evaluation of $\alpha\ \mathsf{S}_I\ \beta$ (Fig. 2). What is different is the data that is stored for each time-stamp and the way we update it. For S, each stored table T_c corresponds to the satisfactions of $\alpha\ \mathsf{S}_{[c,\,c]}\ \beta$. For $\blacktriangleleft_I\ r$, each T_c is a mapping from a regular expression s to the table denoting the satisfactions of $\blacktriangleleft_{[c,\,c]}\ s$. (We represent mappings here by plain functions for readability.) Clearly, this mapping's domain must be finite. We restrict it to the finite set $\Delta(r)$ of right partial derivatives [2,12] of the overall regular expression r, which correspond to the states of a non-deterministic automaton that matches r from right to left.

Partial derivatives allow us to extend satisfactions of $\blacktriangleleft_{[c,\,c]}\ s$ for $s \in \Delta(r)$ at time-point i to satisfactions of $\blacktriangleleft_{[c+(\tau_{i+1}-\tau_i),\,c+(\tau_{i+1}-\tau_i)]}\ s$ for $s \in \Delta(r)$ at time-point $i+1$. The Since operator's counterpart of this extension is the join with R_α, the new satisfactions of α, which is performed for all T_cs for every update. Here, the extension function δ_R inputs a function R assigning the new satisfactions for all tests occurring in r (possibly with a negation stripped) and updates the mapping T_c. It is defined as $\delta_R\ T = (\lambda s.\ \delta\ \mathsf{id}\ R\ T\ s)$ where δ is defined recursively on the structure of regular expressions as shown in Fig. 7. The first parameter of δ uses continuation passing style. It builds up a regular expression context that we use when evaluating the leaves. It is thus guaranteed that if we apply δ to any regular expression $s \in \Delta(r)$, all calls to T will apply T to some $s' \in \Delta(r)$.

The function δ uses the recursive function $\varepsilon_{\mathsf{lax}}$ in its definition. This function computes the assignments that give rise to matches of the form (i, i) under the assumption that a guard (in form of the table X) is given. For δ, the recursive call acts as $\varepsilon_{\mathsf{lax}}$'s guard.

The function $\varepsilon_{\mathsf{strict}}$ is used to update the state with satisfying assignments at the newly added time-point (Fig. 6). It is only specified for expressions satisfying

type_synonym *atable* = *nat set* × *table* **type_synonym** *query* = *atable set*

fun ↓ :: *atable* ⇒ *nat set* ⇒ *atable* **where** $(U, A) \downarrow V = (U \cap V, \{v \downarrow V \mid v \in A\})$
fun ↓ :: *query* ⇒ *nat set* ⇒ *query* **where** $Q \downarrow V = \{(U, A) \downarrow V \mid (U, A) \in Q\}$
fun extend :: *nat set* ⇒ *query* ⇒ *nat set* × *tuple* ⇒ *query* **where**
  extend $V\ Q\ (T, t) = \{(U, \{v \in A \mid \forall i \in T \cap U.\ t\,!\,i = v\,!\,i\}) \downarrow V \mid (U, A) \in Q\}$

fun generic_join :: *nat set* ⇒ *query* ⇒ *query* ⇒ *table* **where**
  generic_join $V\ Q_{pos}\ Q_{neg} =$ if $|V| \leq 1$ then $(\bigcap(\_, X) \in Q_{pos}.\ X) - (\bigcup(\_, Y) \in Q_{neg}.\ Y)$ else
    let $(I, J) =$ getIJ $V\ Q_{pos}\ Q_{neg}$;
      $Q^I_{pos} = \{(V, X) \in Q_{pos} \mid V \cap I \neq \{\}\} \downarrow I$; $Q^I_{neg} = \{(U, X) \in Q_{neg} \mid U \subseteq I\}$;
      $A^I =$ generic_join $I\ Q^I_{pos}\ Q^I_{neg}$;
      $Q^J_{pos} = \{(V, X) \in Q_{pos} \mid V \cap J \neq \{\}\}$; $Q^J_{neg} = Q_{neg} - Q^I_{neg}$;
      $R = \{(t,$ generic_join J (extend $J\ Q^J_{pos}\ (I, t))$ (extend $J\ Q^J_{neg}\ (I, t))) \mid t \in A^I\}$
    in $(\bigcup(t, A) \in R.\ \{$the (join1 $(v, t)) \mid v \in A\})$

Fig. 8. Multi-way join algorithm

safe past strict and uses $\varepsilon_{\mathsf{lax}}$ for subexpressions that only satisfy safe past lax. The recursive structure of $\varepsilon_{\mathsf{strict}}$ and $\varepsilon_{\mathsf{lax}}$ follows the one of safe past. We write $\varepsilon^n_R = (\lambda r.\ \varepsilon_{\mathsf{strict}}\ n\ R\ r)$ and use $\cup$ to denote the pointwise union of mappings in Fig. 6. The Since operator's counterpart of this update is the addition of the satisfactions for the subformula β (Fig. 2).

The above description just sketches our evaluation algorithm and our formalization provides full details. Our proofs establish the monitor's overall correctness, which amounts to the same statement as Theorem 1 but now covers the syntax and semantics extended with the match operators (and aggregations). In particular, the formalization also includes the future match operators for which the evaluation uses similar ideas (partial derivatives), but in a symmetric fashion following the definition of safe future.

5 Multi-way Join

The natural join $\bowtie$ is a central operation in first-order monitors. Not only is it used to evaluate conjunctions; temporal operators also crucially rely on it. Despite this operation's importance, both MonPoly and VeriMon naively compute $A \bowtie B$ as nested unions: $\bigcup v \in A.\ \bigcup w \in B.\ \lceil \mathsf{join1}\,(v, w) \rceil$, where join1 joins two tuples v and w if possible, and $\lceil \_ \rceil$ converts the optional result into a set. In this section, we describe a recent development from database theory that we formalize and extend to optimize the computation of joins.

Ngo et al. [37] and Veldhuizen [48] have developed worst-case optimal multi-way join algorithms that compute the natural join of multiple tables. Here, optimality means that the algorithm never constructs an intermediate result that is larger than the maximum size of all input tables and the overall output. This strictly improves over any evaluation plan using binary joins: There are tables A, B, and C such that the size of $A \bowtie B \bowtie C$ is linear in $|A| = |B| = |C|$, but any plan constructs a quadratic intermediate result from the binary join it

evaluates first [39, Fig. 2]. The key idea of the multi-way join is to build the result table column-wise, adding one or more columns at a time, while taking all tables that refer to the currently added columns into account. All intermediate results are restrictions of the overall result to the processed columns, and thus not larger than the overall result.

Figure 8 shows our formalization of the multi-way join algorithm following Ngo et al.'s unified presentation [39] but generalizing it to support anti-joins $\triangleright$; these additions are highlighted in gray. A *query* is a set of *atable*s, i.e., tables annotated with the columns (represented by *nat*) they have. The main function, generic_join, takes as input a set of columns V and two queries Q_{pos} and Q_{neg}. It computes the multi-way join of Q_{pos} while subtracting the tuples of tables in Q_{neg}. For example, generic_join $\{a, b, c, d\}$ $\{(\{a, b\}, A), (\{b, c\}, B), (\{c, d\}, C)\}$ $\{(\{d\}, D), (\{a, c\}, E)\}$ computes $A \bowtie B \bowtie C \triangleright D \triangleright E$.

The algorithm proceeds by recursion on V. The base case in which V is empty or a singleton set is evaluated directly using intersections and unions. We first describe the recursive structure of the original algorithm [39], obtained by ignoring the highlighted anti-join additions in Fig. 8. The algorithm is parametrized by the getIJ function, which partitions V into two nonempty sets I and J that each determine the number of columns and the order in which they are added. Ngo et al. [39] show how different multi-way join algorithms [37,48] can be obtained by using specific instances of getIJ. We use a heuristic to pick first the column i that maximizes the number of tuples in Q_{pos} it affects (by setting $I = \{i\}$). The partitioning only affects performance, not correctness.

Once I and J are fixed, the algorithm constructs a reduced query Q^I_{pos} by focusing on tables that have a column in I. Furthermore, it restricts their columns to I via the overloaded notation $\_ \downarrow I$, which denotes the restriction of tuples (by setting the optional *data* values for columns outside I to $\perp$ [45]), annotated tables, and queries (Fig. 8).

Next, Q^I_{pos} is evaluated recursively, yielding table A^I with columns I. We now consider tables that have a column in J. This yields a second reduced query Q^J_{pos}, which is, however, not restricted to J. Keeping the columns I in Q^J_{pos} allows us to focus on tuples in Q^J_{pos} that match some $t \in A^I$, i.e., coincide with t for all values in columns I. The function extend performs this matching. For each tuple $t \in A^I$, it creates the query extend J Q^J_{pos} (I, t) consisting of tables from Q^J_{pos} restricted to t-matching tuples (in database terminology this is a semi-join) further restricted to columns J. These queries are again solved recursively, each resulting in a table A_t with columns J. The final step consists of merging the tuples t with A_t. Since t and A have disjoint columns I and J, the function call join1 (v, t) will return some result (which we extract via the) for all $v \in A$.

We extend the algorithm to support anti-joins by introducing a second query Q_{neg}, which we think of as being negated. It is not possible to split Q_{neg}'s tables column-wise. Instead, our generalization processes tables with columns U from Q_{neg} once the positive query has accumulated a superset of U as its columns. This is an improvement over the naive strategy of computing Q_{pos} first and only then removing tuples from it.

The correctness of generic_join relies on several side conditions, e.g., no input table may have zero columns and V must be the union of the columns in the positive query. A wrapper function mwjoin takes care of these corner cases, e.g., by computing V from Q_{pos} and Q_{neg}. We omit mwjoin's straightforward definition, but show its correctness property (which only differs from generic_join's correctness by having fewer assumptions):

$$Q_{pos} \neq \{\} \wedge (\forall (V, A) \in Q_{pos} \cup Q_{neg}.\ (\forall v \in A.\ \mathsf{wf\_tuple}\ n\ V\ v) \wedge (\forall x \in V.\ x < n)) \longrightarrow$$
$$z \in \mathsf{mwjoin}\ Q_{pos}\ Q_{neg} \longleftrightarrow \mathsf{wf\_tuple}\ n\ (\bigcup (V, \_) \in Q_{pos}.\ V)\ z \wedge$$
$$(\forall (V, A) \in Q_{pos}.\ z \downarrow V \in A) \wedge (\forall (U, B) \in Q_{neg}.\ z \downarrow U \notin B)$$

In words: whenever Q_{pos} is nonempty and all tables in Q_{pos} and Q_{neg} fit their declared columns, a tuple z belongs to the output of mwjoin iff it has the correct columns and matches all positive tables from Q_{pos} and does not match any negative ones from Q_{neg}.

The multi-way join algorithm is integrated in VeriMon+ by adding a new constructor Ands :: *frm list* ⇒ *frm* to the formula datatype. At least one of the subformulas of Ands must be non-negated, and the columns of the negative subformulas must be a subset of the positive ones. Since MonPoly's parser, which we reuse in VeriMon+, generates formulas with binary conjunctions, we have defined a semantics-preserving preprocessing function convert_multiway (omitted), which rewrites nested binary conjunctions into Ands.

6 Sliding Window Algorithm

To evaluate the temporal operators S and U, VeriMon computes the union of tables that are associated with time-stamps within the operator's interval. These sets of time-stamps often overlap between consecutive monitor steps. The *sliding window algorithm* (SWA) [10] is an efficient algorithm for combining the elements of overlapping sequences with an associative operator. It improves over the naive approach that recomputes the combination (here, the union) from scratch for every sequence. MonPoly uses SWA for the special cases $\blacklozenge_I \beta = \mathsf{TT}\ \mathsf{S}_I\ \beta$ and $\Diamond_I \beta = \mathsf{TT}\ \mathsf{U}_I\ \beta$, where $\mathsf{TT} = \exists x.\ x \approx x$. However, SWA was not designed for the evaluation of arbitrary S and U operators. For these, the tables in the sequence must be joined with the left subformula's results in every monitor step. In a separate work [27,28], we formally verified SWA's functional correctness (but not its optimality) and extended it with a join operation to support arbitrary S and U operators.

SWA is overly general: it supports any associative operator, not just the union of tables. We conjecture that the generic SWA algorithm is not optimal in the special case needed for S and U. To optimize the evaluation of the S and U operators in VeriMon+, we abstracted the individual steps of their evaluation in one locale for each of them (Sect. 6.1). We then instantiated the locales with specialized sliding window algorithms (Sect. 6.2). Due to space limitations, we only describe the optimization for the Since operator here.

```
locale msaux = fixes valid_msaux :: args ⇒ ts ⇒ 'msaux ⇒ mslist ⇒ bool
  and init_msaux :: args ⇒ 'msaux
  and add_new_ts :: args ⇒ ts ⇒ 'msaux ⇒ 'msaux
  and join_msaux :: args ⇒ table ⇒ 'msaux ⇒ 'msaux
  and add_new_table :: args ⇒ table ⇒ 'msaux ⇒ 'msaux
  and result_msaux :: args ⇒ 'msaux ⇒ table
```

Fig. 9. The locale for evaluating the Since operator (assumptions omitted)

6.1 Integration into the Monitor

Recall the evaluation of Since in VeriMon (Sect. 2). First, VeriMon updates the operator's state with a new time-stamp τ' and the satisfactions R_α and R_β for the subformulas α and β. Second, it evaluates the state to obtain the satisfactions for $\alpha \, \mathsf{S}_I \, \beta$.

Let *mslist* denote the type of the S operator's state in VeriMon. In VeriMon+, we define a locale msaux that abstracts the update and evaluation of an optimized state *'msaux* and relates the optimized state to VeriMon's original *mslist* state (Fig. 9). We provide additional constant arguments for evaluating the S operator in a record *args*. It consists of the S operator's interval, the arguments to wf_tuple characterizing the satisfactions of the two subformulas, and a Boolean value denoting whether the left subformula occurs negated. The predicate valid_msaux relates an optimized state to VeriMon's state with respect to the given *args* and a current time-stamp. The function init_msaux returns an initial optimized state. The next three functions add_new_ts, join_msaux, and add_new_table correspond to the three steps in which VeriMon's state is updated (Sect. 2), except that now they act on the optimized state. Finally, result_msaux evaluates the optimized state to obtain the satisfactions of the S operator at the current time-point. The omitted locale assumptions state that all operations preserve valid_msaux and that result_msaux returns the union computed on any VeriMon state related by valid_msaux.

6.2 The Specialized Algorithm

VeriMon's state for the S operator consists of a list of tables T_c with the satisfactions of formulas $\alpha \, \mathsf{S}_{[c,\,c]} \, \beta$, along with the corresponding time-stamps. VeriMon stores the satisfactions T_c (and time-stamps) for all c that do not exceed the interval's upper bound.

In our optimized state, we partition the list of tables T_c into a list data_prev for time-stamps that are not yet in the interval and a list data_in for time-stamps that are already in the interval. The state also contains a mapping tuple_in that assigns to each tuple occurring in some table T_c in the interval the latest time-stamp in the interval for which this tuple occurs in the respective table. Finally, the state contains a mapping tuple_since that assigns to each tuple occurring in some table T_c in the entire state the *earliest* time-stamp for which this tuple occurs in the respective table. (For efficiency, we delete tuples from tuple_since

ts	*db*	step	data_prev	data_in	tuple_in	tuple_since
		init_msaux	[]	[]	{}	{}
1	$\{Q(a), Q(b), Q(c)\}$	add_new_table	$[(1, \{a,b,c\})]$	[]	{}	$\{a \mapsto 1, b \mapsto 1, c \mapsto 1\}$
2	$\{P(b), P(c)\}$	join_msaux	$[(1, \{a,b,c\})]$	[]	{}	$\{b \mapsto 1, c \mapsto 1\}$
		add_new_table	$[(1, \{a,b,c\}), (2, \{\})]$	[]	{}	$\{b \mapsto 1, c \mapsto 1\}$
3	$\{P(b), P(c), Q(a), Q(b)\}$	add_new_ts	$[(2, \{\})]$	$[(1, \{a,b,c\})]$	$\{b \mapsto 1, c \mapsto 1\}$	$\{b \mapsto 1, c \mapsto 1\}$
		add_new_table	$[(2, \{\}), (3, \{a,b\})]$	$[(1, \{a,b,c\})]$	$\{b \mapsto 1, c \mapsto 1\}$	$\{a \mapsto 3, b \mapsto 1, c \mapsto 1\}$
7	$\{P(a)\}$	add_new_ts	[]	$[(3, \{a,b\})]$	$\{a \mapsto 3, b \mapsto 3\}$	$\{a \mapsto 3, b \mapsto 1, c \mapsto 1\}$
		join_msaux	[]	$[(3, \{a,b\})]$	$\{a \mapsto 3\}$	$\{a \mapsto 3\}$
		add_new_table	$[(7, \{\})]$	$[(3, \{a,b\})]$	$\{a \mapsto 3\}$	$\{a \mapsto 3\}$

Fig. 10. An example of updating the optimized state for the formula $P(x)\ \mathsf{S}_{[2,4]}\ Q(x)$

lazily, i.e., only at defined garbage collection points, such that the mapping may even contain tuples from some T_c that already has fallen out of the interval.)

The state is initialized via init_msaux to consist of empty lists and empty mappings. The function add_new_ts drops tables from data_in that fall out of the interval based on a newly received time-stamp. It also removes those tuples from tuple_in whose latest occurrence (which is stored in this mapping) has fallen out of the interval. Then it moves tables that newly enter the interval from data_prev to data_in, and updates the tuples from these moved tables in tuple_in to the most recent time-stamp τ for which they now occur in the interval, but only if tuple_since maps the tuple to a time-stamp that is at most τ.

The function join_msaux only modifies the mappings tuple_since and tuple_in by removing tuples that are not matched by any tuple in the given table R_α. The function add_new_table appends the new table R_β to data_prev (or directly data_in, if $0 \in_{\mathcal{I}} I$), adds the tuples from R_β that were not in tuple_since to that mapping, and, if $0 \in_{\mathcal{I}} I$, updates the tuples from R_β in the mapping tuple_in to the current time-stamp. Finally, result_msaux returns the keys of the mapping tuple_in, in particular without computing any unions. In other words, tuple_in contains precisely the tuples that are in the interval and have not been removed by joins. Crucially, and unlike in VeriMon's state, the join operation does not change the tables T_c in our optimized state. This functionality is implemented more efficiently by filtering the two mappings tuple_since and tuple_in.

Example. Figure 10 shows how the optimized state for the formula $P(x)\ \mathsf{S}_{[2,4]}\ Q(x)$ is updated. In total, four time-points are processed. The first two columns show the time-stamp and database for each time-point. The other columns show

the state after applying the step named in the third column. Each step corresponds to a function in the msaux locale. The satisfactions $\{\}, \{\}, \{b, c\}, \{a\}$ returned by result_msaux can be read off from the mapping tuple_in after each time-point's last step. We omit steps that do not change the state.

The first row shows the initial state. For the first time-point, the steps add_new_ts with time-stamp 1 and join_msaux with the table $\{\}$ (as there are no P events) do not change the initial state. Then, add_new_table appends the table $\{a, b, c\}$ with the parameters of the Q events to data_prev (as $0 \notin_{\mathcal{I}} [2, 4]$) and adds its elements to tuple_since.

For the second time-point, VeriMon+ applies add_new_ts with time-stamp 2. Again this step has no effect: data_prev's first entry is not moved to data_in as the difference $2-1$ to the current time-stamp is not in $[2, 4]$. Next, join_msaux with the table $\{b, c\}$ (from the P events) removes a from tuple_since, but not from data_prev. Finally, add_new_table appends the table $\{\}$ (as there are no Q events) to data_prev.

For the third time-point, add_new_ts moves data_prev's first entry to data_in because the time-stamp difference $3-1$ is in $[2, 4]$. The values b, c of that entry are added to tuple_in because tuple_since maps them to a time-stamp ≤ 1. Note that a is not added, as it is not contained in tuple_since. The join_msaux step with the table $\{b, c\}$ does not change the state. The add_new_table step appends $\{a, b\}$ to data_prev. Now, a is added to tuple_since, whereas b is already contained in tuple_since and its value is not updated.

When the fourth time-point is processed, the first two observed time-stamps fall out of the interval and add_new_ts discards their entries from data_prev and data_in, and their values from tuple_in but not from tuple_since. As before, the last table $\{a, b\}$ in data_prev is moved to data_in and its elements are added to tuple_in. As time has progressed by more than the upper bound of the interval $[2, 4]$, join_msaux triggers garbage collection, which removes the key c from tuple_since. The join operation further removes b from tuple_in and tuple_since. Finally, add_new_table appends $\{\}$ to data_prev.

7 Evaluation

We perform two kinds of experiments. First, we carry out differential testing [35] of VeriMon+ against three (unverified) state-of-the-art monitors: MonPoly [9], Aerial [11], and Hydra [43]. Second, we compare VeriMon+'s performance to these monitors on representative formulas. VeriMon+ reuses MonPoly's log and formulas parsers and user interface. The verified monitor's code extracted from Isabelle is integrated with these unverified components in about 170 lines of unverified OCaml code. Our implementation and our experiments are available [5]. Of the above monitors, only VeriMon+ supports full MFODL. MonPoly supports a monitorable fragment of MFOTL with bounded future operators and aggregations. Aerial and Hydra support the propositional fragment of MFODL.

Differential Testing. To validate the results produced by unverified monitors, we generate random stream prefixes and formulas, invoke the monitors, and

compare their results to VeriMon+'s. For this purpose, we developed a random stream and formula generator. It takes as parameters the formula size (in terms of number of operators) and the number of free variables that occur in the formula. The generator can be configured to generate formulas within the fragments of MFODL supported by the different monitors we evaluate.

Our tests uncovered several classes of inputs where MonPoly's output deviated from VeriMon+'s. Here, we show one example and refer to our extended report [6] for a comprehensive overview. Namely, formulas of the form $m \leftarrow \Omega\ x; x.\ \blacklozenge_I \alpha$, where $\mathsf{fv}\ \alpha = \{x, y\}$, $\Omega \in \{\mathsf{Min}, \mathsf{Max}\}$, and $0 \notin_{\mathcal{I}} I$, were evaluated in MonPoly using a specialized algorithm, which incorrectly updated the satisfactions of α when they fell out of the interval I.

Aerial's and Hydra's output mostly coincided with VeriMon+'s. However, we noticed that Hydra's output is not as eager as it could be at the end of the stream prefix. For example, $\triangleright_{[1,1]}\ (\star \cdot (\mathsf{TT} \vee \triangleright_{[1,1]}\ \star)?)$ is satisfied at time-point 0 of the prefix $(\{\}, 0), (\{\}, 1)$ due to the existence of time-point 1, where TT can be evaluated. The subformula $\triangleright_{[1,1]}\ \star$ cannot be evaluated at time-point 1. This prevents Hydra from outputting this verdict at 0.

Performance Evaluation. To assess VeriMon+'s performance, we selected four formulas, shown in Fig. 11, which exercise the optimizations (multi-way join and sliding window) and the language features (aggregations and regular expressions) we have introduced. The formula $\mathsf{Star}(N)$ is derived from the *star* conjunctive query, commonly used as a benchmark for joins [14]. We use it to evaluate our multi-way join (for $N = 10$) and sliding window (for $N = 30$) implementations. The formula Top is a commonly used aggregation query, which computes the most frequently occurring value of the event P's second parameter. Finally, Alt checks if events P and Q alternate over the last 10 time units.

We generate random stream prefixes with a time span of 60 time units containing events P, Q, and R, each with two integer parameters sampled uniformly at random from the set $\{1, 2, \ldots, 10^9\}$. Our stream generator is parametrized by the event rate (i.e., by the number of events with the same time-stamp). Since VeriMon+ reuses MonPoly's formula and log parsing infrastructure, there is an additional (conceptually unnecessary) overhead caused by converting the data structures to match the appropriate interfaces. In cases where the monitoring task is easy, this becomes the bottleneck and MonPoly performs better than VeriMon+. To make the monitoring task difficult for $\mathsf{Star}(10)$, we sample the value of the first parameter of each event (the common variable x) using the Zipf distribution. Thus, some parameter values occur frequently. This results in large intermediate tables, which are problematic for binary joins.

Figure 11 shows that VeriMon+ outperforms MonPoly on the $\mathsf{Star}(N)$ formulas. The results confirm the feasibility of monitoring aggregations and regular expressions with VeriMon+. Specialized algorithms remain more performant on problems in their domain.

Event rate	Star(10)			Star(30)			Top		Alt		
Formula / Monitor	MonPoly	VeriMon+	VeriMon	MonPoly	VeriMon+	VeriMon	MonPoly	VeriMon+	Aerial	Hydra	VeriMon+
50	0.0/6.2	0.1/9.3	0.2/9.5	0.1/6.4	0.2/12.0	3.0/12.4	0.2/8.9	6.0/10.4	0.3/5.8	0.2/3.2	2.1/8.6
100	0.1/7.0	0.2/12.0	0.9/16.5	0.1/7.0	0.3/13.4	10.7/24.2	0.3/10.0	29.9/12.6	0.4/5.8	0.2/3.2	3.4/8.6
200	0.5/9.1	0.3/12.1	6.2/47.4	0.4/9.2	0.7/18.7	50.1/48.5	0.9/9.9	to	0.6/6.0	0.2/3.1	7.2/8.8
500	6.0/9.8	1.3/16.3	so	2.5/12.9	1.9/32.5	so	2.9/13.5	to	1.1/6.3	0.2/3.2	18.0/8.7
1000	38.0/12.8	2.5/22.2	so	11.6/17.9	5.2/58.0	so	10.9/22.4	to	1.7/6.4	0.3/3.1	34.1/8.8
2000	to	5.9/36.5	so	to	11.9/106.7	so	22.0/34.2	to	3.1/6.3	0.4/3.2	to
4000	to	15.0/65.0	so	to	22.8/206.0	so	50.8/62.1	to	5.0/6.5	0.6/3.3	to

$\mathsf{Star}(N) \equiv (\blacklozenge_{[0,N]} P(x,y) \wedge Q(x,z)) \wedge \Diamond_{[0,N]} R(x,w)$ $\quad \mathsf{Alt} \equiv \blacktriangleleft_{[10,10]} (P? \cdot \star \cdot Q? \cdot \star)^*$

$\mathsf{Top} \equiv (m \leftarrow \mathsf{Max}\ s; v.\ (s \leftarrow \mathsf{Cnt}\ id; id.\blacklozenge P(id,v))) \wedge (m \leftarrow \mathsf{Cnt}\ id; id.\blacklozenge P(id,v))$

Fig. 11. Time (s)/memory (MB) usage of the monitors (to = timeout of 60s, so = stack overflow)

8 Conclusion

We have presented a verified monitor, competitive with the state-of-the-art, for the expressive specification language metric first-order dynamic logic. Our formalization comprises roughly 15 000 lines of Isabelle code, distributed over the four features we presented: regular expressions (2 000), terms and aggregations (750), multi-way join (3 300), and the sliding window algorithm (3 000). Isabelle extracts a 7 500 line OCaml program from our formalization. This code includes efficient libraries representing sets and mappings via red–black trees introduced transparently into the formalization via the Containers framework [32]. We also use and extend a formalization of IEEE floating point numbers [50].

We have made additional contributions from the algorithmic perspective. Our monitor is the first monitoring algorithm for MFODL with aggregations. Moreover, our specialized sliding window algorithm improves over the existing generic algorithm [10]. Our usage of multi-way joins in the context of first-order monitoring is also novel, as is our extension of the multi-way join algorithm to handle anti-joins. It would be interesting to investigate the optimality of this extension and further consider a multi-way-like evaluation of an arbitrary Boolean combination of finite tables.

Our focus was on extending the verified monitor's specification language and improving its algorithms. As next steps, we plan to further improve performance by refining our algorithms to imperative data structures following Lammich's methodology [29,30].

Acknowledgment. We thank the anonymous IJCAR reviewers for their helpful comments. This research is supported by the US Air Force grant "Monitoring at Any Cost" (FA9550-17-1-0306) and by the Swiss National Science Foundation grant "Big Data Monitoring" (167162). The authors are listed in alphabetical order.

References

1. Alur, R., Fisman, D., Raghothaman, M.: Regular programming for quantitative properties of data streams. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 15–40. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_2

2. Antimirov, V.M.: Partial derivatives of regular expressions and finite automaton constructions. Theoret. Comput. Sci. **155**(2), 291–319 (1996). https://doi.org/10.1016/0304-3975(95)00182-4
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_9
4. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics. LNCS, vol. 10457. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5
5. Basin, D., et al.: VeriMon+: implementation and case study associated with this paper (2020). https://bitbucket.org/jshs/monpoly/downloads/verimonplus.zip
6. Basin, D., et al.: A formally verified, optimized monitor for metric first-order dynamic logic (extended report) (2020). https://people.inf.ethz.ch/trayteld/papers/ijcar20-verimonplus/verimonplus_report.pdf
7. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. Form Methods Syst. Des. **46**(3), 262–285 (2015). https://doi.org/10.1007/s10703-015-0222-7
8. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015). https://doi.org/10.1145/2699444
9. Basin, D., Klaedtke, F., Zălinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017)
10. Basin, D., Klaedtke, F., Zălinescu, E.: Greedily computing associative aggregations on sliding windows. Inf. Process. Lett. **115**(2), 186–192 (2015). https://doi.org/10.1016/j.ipl.2014.09.009
11. Basin, D., Krstić, S., Traytel, D.: AERIAL: almost event-rate independent algorithms for monitoring metric regular properties. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 29–36. EasyChair (2017)
12. Basin, D., Bhatt, B.N., Krstić, S., Traytel, D.: Almost event-rate independent monitoring. Form. Methods Syst. Des. **54**(3), 449–478 (2019). https://doi.org/10.1007/s10703-018-00328-3
13. Bauer, A., Küster, J.-C., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 59–75. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_4
14. Beame, P., Koutris, P., Suciu, D.: Communication steps for parallel query processing. J. ACM **64**(6), 40:1–40:58 (2017). https://doi.org/10.1145/3125644
15. Benzaken, V., Contejean, É., Keller, C., Martins, E.: A Coq formalisation of SQL's execution engines. In: Avigad, J., Mahboubi, A. (eds.) ITP 2018. LNCS, vol. 10895, pp. 88–107. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94821-8_6
16. Blech, J.O., Falcone, Y., Becker, K.: Towards certified runtime verification. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 494–509. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_34
17. Bohrer, B., Tan, Y.K., Mitsch, S., Myreen, M.O., Platzer, A.: VeriPhy: verified controller executables from verified cyber-physical system models. In: Foster, J.S., Grossman, D. (eds.) PLDI 2018, pp. 617–630. ACM (2018). https://doi.org/10.1145/3192366.3192406
18. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964). https://doi.org/10.1145/321239.321249

19. D'Angelo, B., et al.: LOLA: runtime monitoring of synchronous systems. In: TIME 2005, pp. 166–174. IEEE Computer Society (2005). https://doi.org/10.1109/TIME.2005.26
20. Dardinier, T.: Formalization of multiway-join algorithms. Archive of Formal Proofs (2019). https://isa-afp.org/entries/Generic_Join.html
21. Dardinier, T., Heimes, L., Raszyk, M., Schneider, J., Traytel, D.: Formalization of an optimized monitoring algorithm for metric first-order dynamic logic with aggregations. Archive of Formal Proofs (2020). https://isa-afp.org/entries/MFODL_Monitor_Optimized.html
22. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Rossi, F. (ed.) IJCAI 2013, pp. 854–860. IJCAI/AAAI (2013)
23. Havelund, K.: Rule-based runtime verification revisited. STTT **17**(2), 143–170 (2015). https://doi.org/10.1007/s10009-014-0309-2
24. Havelund, K., Leucker, M., Reger, G., Stolz, V.: A shared challenge in behavioural specification (Dagstuhl Seminar 17462). Dagstuhl Rep. **7**(11), 59–85 (2017). https://doi.org/10.4230/DagRep.7.11.59
25. Havelund, K., Peled, D.: Efficient runtime verification of first-order temporal properties. In: Gallardo, M.M., Merino, P. (eds.) SPIN 2018. LNCS, vol. 10869, pp. 26–47. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94111-0_2
26. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
27. Heimes, L.: Extending and optimizing a verified monitor for metric first-order temporal logic. Bachelor's thesis, Department of Computer Science, ETH Zürich (2019)
28. Heimes, L., Schneider, J., Traytel, D.: Formalization of an algorithm for greedily computing associative aggregations on sliding windows. Archive of Formal Proofs (2020). https://isa-afp.org/entries/Sliding_Window_Algorithm.html
29. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) ITP 2019. LIPIcs, vol. 141, pp. 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.22
30. Lammich, P.: Refinement to imperative HOL. J. Autom. Reasoning **62**(4), 481–503 (2019). https://doi.org/10.1007/s10817-017-9437-1
31. Laurent, J., Goodloe, A., Pike, L.: Assuring the guardians. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 87–101. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_6
32. Lochbihler, A.: Light-weight containers for Isabelle: efficient, extensible, nestable. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 116–132. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_11
33. Malecha, J.G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL 2010, pp. 237–248. ACM (2010. https://doi.org/10.1145/1706299.1706329
34. Mamouras, K., Raghothaman, M., Alur, R., Ives, Z.G., Khanna, S.: StreamQRE: modular specification and efficient evaluation of quantitative queries over streaming data. In: Cohen, A., Vechev, M.T. (eds.) PLDI 2017, pp. 693–708. ACM (2017). https://doi.org/10.1145/3062341.3062369
35. McKeeman, W.M.: Differential testing for software. Digit. Tech. J. **10**(1), 100–107 (1998)

36. Mitsch, S., Platzer, A.: ModelPlex: verified runtime validation of verified cyber-physical system models. Form. Methods Syst. Des. **49**(1–2), 33–74 (2016). https://doi.org/10.1007/s10703-016-0241-z
37. Ngo, H.Q., Porat, E., Ré, C., Rudra, A.: Worst-case optimal join algorithms: [extended abstract]. In: Benedikt, M., Krötzsch, M., Lenzerini, M. (eds.) PODS 2012, pp. 37–48. ACM (2012). https://doi.org/10.1145/2213556.2213565
38. Ngo, H.Q., Porat, E., Ré, C., Rudra, A.: Worst-case optimal join algorithms. J. ACM **65**(3), 16:1–16:40 (2018). https://doi.org/10.1145/3180143
39. Ngo, H.Q., Ré, C., Rudra, A.: Skew strikes back: new developments in the theory of join algorithms. SIGMOD Rec. **42**(4), 5–16 (2013). https://doi.org/10.1145/2590989.2590991
40. Peycheva, G.: Real-time verification of datacenter security policies via online log analysis. Master's thesis, ETH Zürich (2018)
41. Pike, L., Niller, S., Wegmann, N.: Runtime verification for ultra-critical systems. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 310–324. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_23
42. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Experience report: a do-it-yourself high-assurance compiler. In: Thiemann, P., Findler, R.B. (eds.) ICFP 2012, pp. 335–340. ACM (2012). https://doi.org/10.1145/2364527.2364553
43. Raszyk, M., Basin, D., Krstić, S., Traytel, D.: Multi-head monitoring of metric temporal logic. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 151–170. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_9
44. Rizaldi, A., et al.: Formalising and monitoring traffic rules for autonomous vehicles in Isabelle/HOL. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 50–66. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_4
45. Schneider, J., Basin, D., Krstić, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 310–328. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_18
46. Thati, P., Rosu, G.: Monitoring algorithms for metric temporal logic specifications. Electron. Notes Theoret. Comput. Sci. **113**, 145–162 (2005). https://doi.org/10.1016/j.entcs.2004.01.029
47. Ulus, D.: Montre: a tool for monitoring timed regular expressions. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 329–335. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_16
48. Veldhuizen, T.L.: Triejoin: a simple, worst-case optimal join algorithm. In: Schweikardt, N., Christophides, V., Leroy, V. (eds.) ICDT 2014, pp. 96–106. OpenProceedings.org (2014). https://doi.org/10.5441/002/icdt.2014.13
49. Völlinger, K.: Verifying the output of a distributed algorithm using certification. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 424–430. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_29
50. Yu, L.: A formal model of IEEE floating point arithmetic. Archive of Formal Proofs (2013). https://isa-afp.org/entries/IEEE_Floating_Point.html