From Harbin with Love: Freezing MFOTL

Srđan Krstić and Dmitriy Traytel

Department of Computer Science, University of Copenhagen, Copenhagen, Denmark,
{srdan.krstic,traytel}@di.ku.dk

Abstract. We introduce the *freeze* operator in metric first-order temporal logic (MFOTL). It records a relation and provides means to refer to it independently of temporal operators surrounding the reference. As such, it generalizes the (half-order) freeze quantifier available in timed propositional temporal logic to a first-order operator. Semantically, the freeze operator is similar to the non-recursive let operator, but with a subtle yet profound difference. We provide a monitoring algorithm for formulas with freeze operator and implement it in the MonPoly and VeriMon monitors. The VeriMon implementation is accompanied by a correctness proof checked by the Isabelle proof assistant. We evaluate the performance of both implementations and demonstrate how the freeze operator can be used in rewrite rules that aim to bring MFOTL formulas into relational algebra normal form without affecting the monitor's progress. This yields a strict semantic improvement over previous rewrites, although not always performance-wise.

Keywords: Freeze operator · Monitoring · Temporal logic.

1 Introduction

David Basin and colleagues introduced metric first-order temporal logic (MFOTL) [5], an expressive specification language for monitoring systems at runtime (Section 2). MFOTL is the specification language of multiple monitoring tools, including MonPoly [6], VeriMon [20], TimelyMon [19], and WhyMon [15], and subsumes other first-order languages, including the past-only variant used by DejaVu [12].

Over the years, new features have been integrated into MFOTL and some of its monitors: aggregations [4], regular expressions [3], and non-recursive and past-recursive definitions [21]. VeriMon supports all four extensions, while MonPoly supports all but past-recursive definitions. A recent retrospective [8] summarizes the grown logic.

A natural question arises: *Is something still missing?* Some indications to the affirmative come from the study of its rewriting into relational algebra normal form (RANF) [1, 8]. To work with finite relations, MonPoly and VeriMon preprocess MFOTL into RANF, following heuristic and incomplete rewriting rules proposed by Müller [16]. Raszyk [17] develops a similar but complete RANF translation, and Christensen [9] further investigates the interaction of Müller's rules with the monitor's progress, i.e., the index up to which the monitor can emit outputs after reading a given trace prefix. A key observation is that RANF rewriting may turn past-only formulas into ones that depend on the future, thus negatively affecting the monitor's progress and introducing delays. Consider the rule that pushes a subformula α into the past operator $\blacklozenge_I$ (Müller's rule 14 [16, page 81]):

$$\alpha \wedge \blacklozenge_I \beta \quad \longrightarrow \quad \alpha \wedge \blacklozenge_I ((\lozenge_I \alpha) \wedge \beta)$$

This rewrite may be useful in case β's topmost operator is a negation, e.g. $\beta = \neg p(x)$, which is unsupported by RANF as there are typically infinitely many values for x violating the predicate p. Then α can provide the needed guard, e.g., $\alpha = q(x,y)$. For our example α and β, the original formula is past-only but not in RANF. The rewritten formula is in RANF, but refers to the future. The intervals do not cancel: evaluating the formula at i evaluates $\blacklozenge_I$'s argument at some past $j \leq i$, from which $\Diamond_I$ may reach beyond i. This is a semantically safe overapproximation: we would be happy with only referring to valuations of α at precisely i—unfortunately, under temporal operators we have no way of doing so in MFOTL. This overapproximation is not helpful when I is unbounded, as monitors typically do not support unbounded future intervals that the rewrite would introduce.

Can we refer to a formula's evaluation result at an index *regardless* of the surrounding temporal operators? To this end, the *freeze quantifier* was introduced in timed propositional temporal logic (TPTL) [2], where it binds a variable to the current time. Demri and Lazić [10] study the connection of the freeze quantifier to register automata. Basin et al. [7] extend freeze to metric temporal logic and use it in an out-of-order monitor.

We generalize freeze to the first-order case, in which the freeze operator captures an *entire table* (a finite relation) rather than a single value, making it an operator rather than a quantifier. Semantically, our Frz operator is a minimal modification of the Let operator for non-recursive definitions (Section 3): Let defines a predicate that re-evaluates its defining subformula at the queried index, Frz ignores the queried index and always returns the subformula's value at the *outer* index. The two coincide when the frozen predicate is accessed only at the current index; the difference manifests only under temporal operators.

Despite the minimal semantic change, the monitoring algorithm for Frz we develop (Section 3) differs substantially from that of Let: Freezing decouples the index at which α is evaluated from the index at which it is queried inside β, so the algorithm must manage this decoupling explicitly. We implemented our Frz algorithm in VeriMon first, along with a correctness proof checked by the Isabelle proof assistant. Our design here follows the motto of "the simplest computation that does the job."

Equipped with the freeze operator, we revisit RANF rewriting (Section 4). To this end, we introduce new Frz-based rewrite rules that replace the temporal conjuncts with a frozen snapshot at the current time-point, preserving the monitor's progress.

We extract executable OCaml code from the Isabelle formalization, yielding VeriMon, and reimplement the algorithm in MonPoly, where we further optimize the implementation with the assistance of large language models (Section 5). We validate the optimizations against VeriMon using differential testing and reflect on LLM-assisted programming safeguarded by a verified oracle. Finally, we evaluate Frz's performance and the Frz-based rewrites (Section 6). Our experiments quantify the cost of Frz as a function of trace size and formula complexity, and compare the Frz-based rewrites to the original rewrites.

In summary, this paper makes the following contributions:

- We introduce the freeze operator Frz in MFOTL and extend a verified monitor to support Frz, together with a machine-checked correctness proof of the extension;
- We devise Frz-based rewrite rules that avoid the temporal shifts of existing rewrites;
- We implement our algorithm in VeriMon (verified) and MonPoly (AI-assisted); and
- We evaluate the performance of our implementations.

All work is available in VeriMon's development repository [22].

datatype $data = \mathsf{Int}\ int \mid \mathsf{Flt}\ double \mid \mathsf{Str}\ string$ **type_synonym** $ts = nat$
type_synonym $db = (string \times data\ list)\ set$ **typedef** $trace = \{s :: (db \times ts)\ stream.\ \mathsf{trace}\ s\}$
datatype $trm = \mathsf{V}\ nat \mid \mathsf{C}\ data \mid trm + trm \mid \ldots$ **typedef** $\mathcal{I} = \{(a :: nat, b :: enat).\ a \leq b\}$
datatype $frm = string(trm\ list) \mid trm \circ trm \mid \neg\ frm \mid \exists\ frm \mid frm \vee frm \mid frm \wedge frm$
$\mid \bullet_{\mathcal{I}}\ frm \mid \bigcirc_{\mathcal{I}}\ frm \mid frm\ \mathsf{S}_{\mathcal{I}}\ frm \mid frm\ \mathsf{U}_{\mathcal{I}}\ frm \mid \mathsf{Let}\ name := frm\ \mathsf{in}\ frm$
fun $\mathsf{et} :: data\ list \Rightarrow trm \Rightarrow data$ **where**
$\mathsf{et}\ v\ (\mathsf{V}\ x) = v\,!\,x \mid \mathsf{et}\ v\ (\mathsf{C}\ x) = x \mid \mathsf{et}\ v\ (t_1 + t_2) = \mathsf{et}\ v\ t_1 + \mathsf{et}\ v\ t_2 \mid \ldots$
fun $\mathsf{sat} :: trace \Rightarrow (name \times nat \rightharpoonup data\ list \Rightarrow nat \Rightarrow bool) \Rightarrow data\ list \Rightarrow nat \Rightarrow frm \Rightarrow bool$ **where**
$\mathsf{sat}\ \sigma\ V\ v\ i\ (p(as)) = (\underline{\mathsf{let}}\ t = \mathsf{map}\ (\mathsf{et}\ v)\ as\ \underline{\mathsf{in}}\ \underline{\mathsf{case}}\ V\ (p, |as|)\ \underline{\mathsf{of}}\ \lfloor f \rfloor \Rightarrow f\ t\ i \mid \bot \Rightarrow (p, t) \in \Gamma\ \sigma\ i)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (t_1 \circ t_2) = (\mathsf{et}\ v\ t_1 \circ \mathsf{et}\ v\ t_2)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\neg\alpha) = (\neg \mathsf{sat}\ \sigma\ V\ v\ i\ \alpha) \mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\alpha \wedge \beta) = (\mathsf{sat}\ \sigma\ V\ v\ i\ \alpha \wedge \mathsf{sat}\ \sigma\ V\ v\ i\ \beta)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\alpha \vee \beta) = (\mathsf{sat}\ \sigma\ V\ v\ i\ \alpha \vee \mathsf{sat}\ \sigma\ V\ v\ i\ \beta) \mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\exists\alpha) = (\exists z.\ \mathsf{sat}\ \sigma\ V\ (z\#v)\ i\ \alpha)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\bullet_I\ \alpha) = (\underline{\mathsf{case}}\ i\ \underline{\mathsf{of}}\ 0 \Rightarrow \mathsf{False} \mid j+1 \Rightarrow \mathsf{T}\ \sigma\ i - \mathsf{T}\ \sigma\ j \in I \wedge \mathsf{sat}\ \sigma\ V\ v\ j\ \alpha)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\bigcirc_I\ \alpha) = (\mathsf{T}\ \sigma\ (i+1) - \mathsf{T}\ \sigma\ i \in I \wedge \mathsf{sat}\ \sigma\ V\ v\ (i+1)\ \alpha)$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\alpha\ \mathsf{S}_I\ \beta) = (\exists j \leq i.\ \mathsf{T}\ \sigma\ i - \mathsf{T}\ \sigma\ j \in I \wedge \mathsf{sat}\ \sigma\ V\ v\ j\ \beta \wedge (\forall k \in \{j <..\ i\}.\ \mathsf{sat}\ \sigma\ V\ v\ k\ \alpha))$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\alpha\ \mathsf{U}_I\ \beta) = (\exists j \geq i.\ \mathsf{T}\ \sigma\ j - \mathsf{T}\ \sigma\ i \in I \wedge \mathsf{sat}\ \sigma\ V\ v\ j\ \beta \wedge (\forall k \in \{i\ ..<\ j\}.\ \mathsf{sat}\ \sigma\ V\ v\ k\ \alpha))$
$\mid \mathsf{sat}\ \sigma\ V\ v\ i\ (\mathsf{Let}\ p := \alpha\ \mathsf{in}\ \beta) = \mathsf{sat}\ \sigma\ (V[(p, \mathsf{nfv}\ \alpha) \mapsto \lambda w\ j.\ \mathsf{sat}\ \sigma\ V\ w\ j\ \alpha])\ v\ i\ \beta$

Fig. 1. Formal syntax and semantics of MFOTL with Let, where $\circ \in \{=, <, \leq\}$

2 Preliminaries

We recall the fragment $\mathcal{L}_{\mathsf{MFOTL}}$ of metric first-order temporal logic (MFOTL) used in the rest of this paper and sketch the overall structure of VeriMon's monitoring algorithm.

2.1 Metric First-Order Temporal Logic

A trace σ is an infinite sequence of time-stamped databases over a fixed data domain, $\Gamma\ \sigma\ i$ denotes the i-th database and $\mathsf{T}\ \sigma\ i$ its time-stamp. Time-stamps are assumed to be monotone and eventually increasing (predicate trace). Terms and formulas are defined by the *trm* and *frm* datatypes in Fig. 1: here, variables are represented by De Bruijn indices, and intervals are non-empty contiguous subsets of $\mathbb{N}$; we write $\in$ for interval membership.

Fig. 1 also defines the semantics via term evaluation (*et* function) and an inductive satisfaction relation (*sat* Boolean function). Notation-wise, $\lfloor\_\rfloor$ and $\bot$ are the option datatype constructors, # is the list constructor, $\mathsf{nfv}\ \alpha$ is the number of α's free variables, and $\{i\ ..<\ j\}$ and $\{i <..\ j\}$ denote $\{i, \ldots, j-1\}$ and $\{i+1, \ldots, j\}$, respectively.

We use a valuation v to interpret free variables and a partial predicate environment V for Let-bound (and later Frz-bound) predicates. For a predicate in V's domain, its interpretation is given by V; otherwise, the relation is read from the trace. The semantics of Let extends V with a new interpretation, potentially shadowing previously present bindings and hiding any homonymous same-arity predicates present in the trace.

To work with tables, we restrict our attention to the RANF fragment of MFOTL [8]. In VeriMon, the fragment is given by the predicate $\mathsf{safe\_formula} :: frm \Rightarrow bool$ (omitted), which places restrictions on the formula's structure to ensure that all subformulas have finitely many satisfying valuations. For example, a disjunction $\alpha \vee \beta$ is in RANF if both α and β are in RANF and have the same free variables ($\mathsf{fv}\ \alpha = \mathsf{fv}\ \beta$). Negation is only permitted in a conjunction-guarded form: $\alpha \wedge \neg\beta$ and all free variables of β must occur in α. Finally, for the non-recursive let operator $\mathsf{Let}\ p := \alpha\ \mathsf{in}\ \beta$, both α and β must be in RANF, and the free variables of α must be exactly the first $\mathsf{nfv}\ \alpha$ De Bruijn indices.

2.2 Monitoring Algorithm

We sketch VeriMon's online algorithm for monitoring RANF formulas, with a special focus on the Let operator [21]. VeriMon processes the trace via two functions: init :: $frm \Rightarrow state$, which builds the initial state, and step :: $db \Rightarrow ts \Rightarrow state \Rightarrow (nat \times table)\ list \times state$, which consumes a time-stamped database and returns the new outputs—each a pair of time-point and $table = data\ list\ set$ (a finite set of satisfying valuations)—together with the updated state. Because future operators may delay verdicts, a single call to step may emit answers for several time-points at once, sorted by time-point in increasing order.

The outer *state* is a wrapper around the inductive internal state $s_\alpha : sfrm$ that mirrors the monitored formula's syntax tree. Each *sfrm* constructor stores the auxiliary data needed to resume monitoring the formula it mirrors: buffers for binary operators, cached tables for binding constructs, and sub-states for all recursive operators. The outer state adds bookkeeping information such as the processed prefix length j and the output arity n.

The main workhorses of init and step are thus the recursive minit :: $frm \to sfrm$ and meval :: $nat \Rightarrow nat \Rightarrow ts\ list \Rightarrow ((name \times nat) \rightharpoonup table\ list) \Rightarrow sfrm \Rightarrow table\ list \times sfrm$. The latter works with batches of new data. Given the prefix length, the output arity, the newly received batch of time-stamps and data, and an *sfrm* state, it returns the new verdict tables together with the updated state. For Let $p := \alpha$ in β, the state SLet $p\ m\ s_\alpha\ s_\beta$ stores the predicate's name and arity and two recursive substates. The full case of meval is:

$$\begin{aligned}&\mathsf{meval}\ j\ n\ ts\ db\ (\mathsf{SLet}\ p\ m\ s_\alpha\ s_\beta) = \underline{\mathsf{let}}\ (xs,\ s'_\alpha) = \mathsf{meval}\ j\ m\ ts\ db\ s_\alpha;\\ &\quad (ys,\ s'_\beta) = \mathsf{meval}\ j\ n\ ts\ (db[(p,m) \mapsto xs])\ s_\beta\ \underline{\mathsf{in}}\ (ys,\ \mathsf{SLet}\ p\ m\ s'_\alpha\ s'_\beta)\end{aligned}$$

The algorithm first evaluates α's sub-monitor on the current database. Then, it evaluates β's sub-monitor with the database updated to hold the freshly computed α verdicts for the predicate p. In general, meval evaluates the sub-monitors, threads their updated states through, and combines new verdicts according to the operator's semantics.

To express correctness, we track each operator's *progress* prog $\sigma\ P\ \varphi\ j$: the index up to which the monitor can soundly evaluate φ after reading the first j databases of σ, relative to a progress map P for Let-bound predicates. Future operators in φ can force prog to be strictly below j. For the Let $p := \alpha$ in β operator, prog is:

$$\mathsf{prog}\ \sigma\ P\ (\mathsf{Let}\ p := \alpha\ \mathsf{in}\ \beta)\ j = \mathsf{prog}\ \sigma\ (P[(p,m) \mapsto \mathsf{prog}\ \sigma\ P\ \alpha\ j])\ \beta\ j$$

That is, β's progress is computed under the progress map extended so that the let-bound predicate p is stalled at α's own progress. Moreover, we use an invariant to tie each operator's state to the semantics sat. The inductive invariant wf_mformula $\sigma\ j\ P\ V\ n\ s_\alpha\ \varphi$ states that the state s_α faithfully represents monitoring φ after reading the first j elements of σ. The main correctness theorem is stated on meval. Given the invariant after j steps and a well-formed environment wf_envs $\sigma\ j\ \delta\ P\ P'\ V\ db$ (the progress maps P and P' are consistent with the supplied database db, and the database covers the trace prefix $[j..<j+\delta]$), meval preserves the invariant and produces exactly the tables containing satisfying valuations (predicate tab) whose indices fall between the old and the new progress values:

$$\begin{aligned}&\mathsf{wf\_mformula}\ \sigma\ j\ P\ V\ n\ s_\varphi\ \varphi \implies \mathsf{wf\_envs}\ \sigma\ j\ \delta\ P\ P'\ V\ db \Longrightarrow\\ &\mathsf{meval}\ (j+\delta)\ n\ (\mathsf{map}\ (\mathsf{T}\ \sigma)\ [j\ ..<\ j+\delta])\ db\ s_\varphi\ =\ (xs,\ s'_\varphi) \Longrightarrow\\ &\mathsf{wf\_mformula}\ \sigma\ (j+\delta)\ P'\ V\ n\ s'_\varphi\ \varphi \wedge\\ &\mathsf{list\_all2}\ (\lambda i.\ \mathsf{tab}\ n\ (\mathsf{fv}\ \varphi)\ (\lambda v.\ \mathsf{sat}\ \sigma\ V\ v\ i\ \varphi))\ [\mathsf{prog}\ \sigma\ P\ \varphi\ j\ ..<\ \mathsf{prog}\ \sigma\ P'\ \varphi\ (j+\delta)]\ xs\end{aligned}$$

3 Freeze Operator

We introduce the freeze operator Frz, which is similar to the non-recursive Let with a small change in the semantics. To this end, we add a new constructor to the *frm* datatype:

datatype $frm = \cdots \mid \mathsf{Frz}\ name := frm\ \mathsf{in}\ frm$

Semantics. Recall Let's semantics from Fig. 1. The semantics of Frz is obtained through a highlighted one character change: j becomes i when referring to α's satisfactions:

$$\mathsf{sat}\ \sigma\ V\ v\ i\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta) = \mathsf{sat}\ \sigma\ (V[(p, \mathsf{nfv}\ \alpha) \mapsto \lambda w\ j.\ \mathsf{sat}\ \sigma\ V\ w\ \boxed{i}\ \alpha])\ v\ i\ \beta$$

Thus, the Frz operator freezes the relation for p at the *current* time-point i instead of letting it vary with the query index j. When viewing the assigned function from valuation w and index j to the Boolean satisfaction $\mathsf{sat}\ \sigma\ V\ w\ i\ \alpha$ as a stream of satisfying valuations for α, note that Frz thus generates a constant stream (whereas Let's stream was not constant, but dependent on j). In other words, Frz-bound predicates *ignore* their own index argument j and always reuses the outer time i. Moreover, β is still evaluated at index i.

Frz and Let coincide when β accesses p only at the current time-point, since both would evaluate p at i in this case. The fundamental difference emerges with temporal operators: Let re-evaluates α at each queried time-point, while Frz uses a frozen α from the outer time i throughout, regardless of which time-point is being queried inside β.

Runtime State. For the non-recursive Let, VeriMon uses the state $\mathsf{SLet}\ p\ m\ s_\alpha\ s_\beta$, where a single shared s_β is advanced *incrementally* across monitor steps: as s_α produces new satisfying valuations, V is extended with them, and s_β reads the current binding of p when it queries. This works for Let because p's relation at any queried index j is simply α's satisfaction at j: the indices align perfectly.

For Frz the situation is rather different: the frozen predicate at any queried index j is the *constant* α's satisfaction at k where k is the outer index. We thus must re-evaluate the β sub-monitor separately for every k. Furthermore, the α and β sub-monitors advance independently. When the α sub-monitor is ahead, the excess α-snapshots must be buffered because the β sub-monitor is not ready to receive them yet. The opposite case, where the β sub-monitor is ahead, is a corner case that arises when β does not reference the frozen predicate. The new constructor SFrz therefore extends the Let state with the buffer αs and additionally records three progress counters: $\mathsf{SFrz}\ p\ m\ s_\alpha\ s_\beta\ p_{\mathsf{Frz}}\ p_\beta\ p_\alpha\ \alpha s$. The fields are:

- p: the frozen predicate name,
- $m = \mathsf{nfv}\ \alpha$: arity of the frozen predicate,
- s_α: the sub-monitor for α,
- s_β: the sub-monitor for β evaluated under the degenerate database $db[(p,m) \mapsto []]$—this is the only part that is shared across different β sub-monitor evaluations,
- p_{Frz}: the progress of the full Frz formula, i.e., $\mathsf{prog}\ \sigma\ P\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta)\ j$, which we are yet to define—our algorithm simply records how many outputs were produced,
- p_β: the progress of β under degenerate p-progress, i.e., $\mathsf{prog}\ \sigma\ (P[(p,m) \mapsto 0])\ \beta\ j$,
- p_α: the progress of α, i.e., $\mathsf{prog}\ \sigma\ P\ \alpha\ j$,
- αs: the queue of pending α-snapshots that have not yet been passed to β corresponding to α-satisfactions at indices k from the interval $[p_{\mathsf{Frz}}\ ..<\ p_\alpha]$.

The initial state is $\mathsf{SFrz}\ p\ (\mathsf{nfv}\ \alpha)\ (\mathsf{minit}\ \alpha)\ (\mathsf{minit}\ \beta)\ 0\ 0\ 0\ []$, where all three progress counters start at 0 and the queue is empty.

Evaluation algorithm. Our algorithm for SFrz relies on the auxiliary freeze_meval, which we refer to as *diagonal replay*. Let $\alpha s = [\alpha_k, \alpha_{k+1}, \ldots]$ be the queue of frozen α-tables starting from index k. Given an evaluation function *eval* that, for a frozen table, returns a list of β-tables, freeze_meval *eval k* αs picks out the list's k-th output and recurses.

freeze_meval *eval k* [] = ([], [])
freeze_meval *eval k* (α#αs) = let (*ys*, _) = *eval* α in
  case drop *k ys* of [] ⇒ ([], α#αs)
  | *y*#_ ⇒ let (*zs*, $\alpha s'$) = freeze_meval *eval* (*k*+1) αs in (*y*#*zs*, $\alpha s'$)

For the first snapshot in the queue we thus get the k-th table from *eval*'s output; for the second the table at $k+1$; and so on. The reason we discard the earlier outputs of each *eval* call is that these calls are independent: the i-th run, applied to α_{k+i}, has not seen any of the earlier frozen tables, and only its $(k+i)$-th output is the verdict for outer index $k+i$ that this snapshot contributes. The pattern of taking the output at k, $k+1$, $k+2$, etc. across successive snapshots α_k, α_{k+1}, α_{k+2} is what gives the construction its diagonal shape. The recursion stops as soon as one call to *eval* does not yet yield that necessary table, leaving the last tried and remaining snapshots queued.

The full meval step for SFrz proceeds as follows:

meval *j n ts db* (SFrz *p m* s_α s_β p_{Frz} p_β p_α αs) =
  let (*xs*, s'_α) = meval *m ts db* s_α; $p'_\alpha = p_\alpha + |xs|$;
    (*zs*, s'_β) = meval *n ts* (*db*[(*p*, *m*) ↦ []]) s_β; $p'_\beta = p_\beta + |zs|$;
    zs_{out} = drop ($p_{\mathsf{Frz}} - p_\beta$) *zs*;
    (*ys*, $\alpha s'$) = freeze_meval
      ($\lambda\alpha$. meval *j n* [] ((map_values (λ__. []) *db*)[(*p*, *m*) ↦ replicate *j* α]) s'_β)
      ($p_{\mathsf{Frz}} - p'_\beta$) (drop ($p_{\mathsf{Frz}} - p_\alpha + |zs_{\text{out}}|$) ($\alpha s$ @ *xs*));
  in (zs_{out} @ *ys*, SFrz *p m* s'_α s'_β ($p_{\mathsf{Frz}} + |zs_{\text{out}}| + |ys|$) p'_β p'_α $\alpha s'$)

In words, the evaluation performs the following steps in sequence:

1. *Advance* α. Evaluate s_α on the current input batch, yielding new α-snapshots *xs* for the range $[p_\alpha \mathinner{..<} p'_\alpha]$.
2. *Shared* β *evaluation.* Evaluate s_β under the degenerate database $db[(p,m) \mapsto []]$ (which corresponds to progress map $P[(p,m) \mapsto 0]$), yielding tables *zs* for $[p_\beta \mathinner{..<} p'_\beta]$. Drop the prefix below p_{Frz} (already output) to obtain zs_{out}, the newly emitted shared output tables for $[p_{\mathsf{Frz}} \mathinner{..<} p_\beta{}']$. Frequently, this list will be empty, but it supports the corner case where p is unused in β.
3. *Trim the snapshot queue.* Extend the queue with the new snapshots *xs* and drop these snapshots for these indices k, which we do not happen to need to evaluate β at k. The drop amount is $p_{\mathsf{Frz}} - p_\alpha + |zs_{\text{out}}|$, i.e., the old "debt" $p_{\mathsf{Frz}} - p_\alpha$ plus the newly emitted shared tables. The remaining queue covers $[p_{\mathsf{Frz}} + |zs_{\text{out}}| \mathinner{..<} p'_\alpha]$.
4. *Diagonal replay.* For each α_i in the remaining queue, evaluate s'_β with p bound to the *constant* list replicate *j* α_i (i.e., the same frozen table at every index up to the current maximum *j*), using an empty time-stamp batch and with the other database keys preserved but emptied (this data has been already incorporated into s'_β in step 2). Then, freeze_meval picks the diagonal tables (from offset $p_{\mathsf{Frz}} - p'_\beta$), yielding *ys*.
5. *Compose the new state and output.* Return zs_{out} @ *ys*, the updated states s'_α and s'_β and the progress counters $p'_{\mathsf{Frz}} = p_{\mathsf{Frz}} + |zs_{\text{out}}| + |ys|$, $p'_\beta = p_\beta + |zs|$, $p'_\alpha = p_\alpha + |xs|$.

Progress. Mirroring the algorithm, the progress function for Frz differs from that of Let. Let $pn = (p, \mathsf{nfv}\ \alpha)$. The progress of $\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta$ is defined as:

$$\begin{aligned}&\mathsf{prog}\ \sigma\ P\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta)\ j = \underline{\mathsf{let}}\ p_\alpha = \mathsf{prog}\ \sigma\ P\ \alpha\ j\ \underline{\mathsf{in}}\\ &\quad \mathsf{LEAST}\ i.\ i \le j \wedge \mathsf{prog}\ \sigma\ (P[pn \mapsto \mathsf{if}\ i < p_\alpha\ \mathsf{then}\ j\ \mathsf{else}\ 0])\ \beta\ j \le i\end{aligned}$$

Thus, for a candidate frontier i below our prefix length j, we calculate:

- if $i < p_\alpha$, then corresponding α-snapshot at i is available up to j, so β's progress is computed using $P[pn \mapsto j]$ (i.e., the frozen predicate is fully usable);
- else, the α-snapshot is unavailable, and β's progess is computed under $P[pn \mapsto 0]$.

The least i where β no longer results in more progress than i is the Frz progress.

From this definition, we obtain three facts that govern the offset arithmetic in the evaluation algorithm. First, $\mathsf{prog}\ \sigma\ (P[pn \mapsto 0])\ \beta\ j \le \mathsf{prog}\ \sigma\ P\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta)\ j$, which justifies the drop offset $p_{\mathsf{Frz}} - p_\beta$ used to extract zs_{out} in phase 2 of the algorithm. Second, if $\mathsf{prog}\ \sigma\ P\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta)\ j > \mathsf{prog}\ \sigma\ P\ \alpha\ j$ then $\mathsf{prog}\ \sigma\ P\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta)\ j = \mathsf{prog}\ \sigma\ (P[pn \mapsto 0])\ \beta\ j$: when Frz progress has outpaced the α-snapshots, all further output must come from the shared β-run—the diagonal replay has nothing to work with. Third, the operational progress conservation equation $\mathsf{prog}\ \sigma\ P\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta)\ (j+\delta) = \mathsf{prog}\ \sigma\ P\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta)\ j + |zs_{\text{out}}| + |ys|$ for zs_{out} and ys as defined in the algorithm confirms that our count updates performed in the algorithm are correct.

Relational Algebra Normal Form. The RANF condition for Frz is the same as for Let:

$$\mathsf{safe\_formula}\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta) = \{0\ ..{<}\ \mathsf{nfv}\ \alpha\} \subseteq \mathsf{fv}\ \alpha \wedge \mathsf{safe\_formula}\ \alpha \wedge \mathsf{safe\_formula}\ \beta$$

The requirement $\{0\ ..{<}\ \mathsf{nfv}\ \alpha\} \subseteq \mathsf{fv}\ \alpha$ ensures that α's free variables have no gaps, so the resulting tables expose all positions of the frozen tuple and can be passed directly as the frozen table for p. No additional syntactic restriction on how p appears in β is necessary.

Correctness Invariant. We extend the state invariant $\mathsf{wf\_mformula}\ \sigma\ j\ P\ V\ n\ s_\alpha\ \varphi$ with a case for SFrz. One auxiliary predicate is used to express that the α-snapshot queue covers exactly the α-satisfactions (as tables) in the interval $[p_{\mathsf{Frz}}\ ..{<}\ p_\alpha]$:

$$\begin{aligned}&\mathsf{wf\_frz\_state}\ \sigma\ j\ P\ V\ p\ m\ \alpha\ \beta\ p_{\mathsf{Frz}}\ p_\alpha\ \alpha s =\\ &\quad p_\alpha = \mathsf{prog}\ \sigma\ P\ \alpha\ j \wedge p_{\mathsf{Frz}} = \mathsf{prog}\ \sigma\ P\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta)\ j \wedge\\ &\quad \mathsf{list\_all2}\ (\lambda k.\ \mathsf{tab}\ m\ (\mathsf{fv}\ \alpha)\ (\lambda v.\ \mathsf{sat}\ \sigma\ V\ v\ k\ \alpha))\ [p_{\mathsf{Frz}}\ ..{<}\ p_\alpha]\ \alpha s\end{aligned}$$

The full inductive invariant for SFrz is:

$$\frac{\begin{array}{c}\mathsf{wf\_mformula}\ \sigma\ j\ P\ V\ m\ s_\alpha\ \alpha\\ \forall i.\ \mathsf{wf\_mformula}\ \sigma\ j\ (P[pn \mapsto 0])\ (V[pn \mapsto \lambda w\ j.\ \mathsf{sat}\ \sigma\ V\ w\ i\ \alpha])\ n\ s_\beta\ \beta\\ \mathsf{wf\_frz\_state}\ \sigma\ j\ P\ V\ p\ m\ \alpha\ \beta\ p_{\mathsf{Frz}}\ p_\alpha\ \alpha s \qquad p_\beta = \mathsf{prog}\ \sigma\ (P[pn \mapsto 0])\ \beta\ j\\ \{0\ ..{<}\ m\} \subseteq \mathsf{fv}\ \alpha \qquad m = \mathsf{nfv}\ \alpha \qquad \mathsf{safe\_formula}\ \alpha \qquad \mathsf{safe\_formula}\ \beta\end{array}}{\mathsf{wf\_mformula}\ \sigma\ j\ P\ V\ n\ (\mathsf{SFrz}\ p\ m\ s_\alpha\ s_\beta\ p_{\mathsf{Frz}}\ p_\beta\ p_\alpha\ \alpha s)\ (\mathsf{Frz}\ p := \alpha\ \mathsf{in}\ \beta)}$$

The first premise constrains s_α exactly as for SLet. The second—the essential new premise—states that s_β satisfies the well-formedness invariant *for every frozen valuation*: for all i, s_β is a valid β-monitor under the valuation-map extension that binds pn to $\lambda w\ j.\ \mathsf{sat}\ \sigma\ V\ w\ i\ \alpha$ and under the degenerate progress map $P[pn \mapsto 0]$. A crucial observation on which our correctness proof relies here is V-insensitivity—the fact that

at time-points below $\mathsf{prog}\ \sigma\ (P[pn \mapsto 0])\ \beta\ j$, the actual value of $V(pn)$ does not affect sat. The third premise records the snapshot queue invariant and together with the fourth premise checks that the stored values p_{Frz}, p_β, and p_α are equal to the corresponding semantic progress values. The arity and safety side conditions match those required by safe_formula. We extend the proof of the correctness theorem of Section 2 to the new Frz case, demonstrating that our invariant is preserved by the meval step of the evaluation algorithm and that meval produces correct verdicts for Frz at each time-point.

4 Rewriting Towards Relational Algebra Normal Form

When a given formula is not in RANF, VeriMon and MonPoly attempt to heuristically rewrite it by applying a set of syntactic rules [16] with the goal of reaching RANF. For a formula α, the monitors compute free variables $\mathsf{fv}\ \alpha$ and the subset of variables that are *range restricted* (or guarded), written $\mathsf{rr}\ \alpha$. The latter can help with guarding variables that would otherwise cause an RANF violation and are propagated into subformulas to produce a semantically equivalent formula, which is "closer" to RANF. Whenever $\mathsf{rr}\ \alpha \cap (\mathsf{fv}\ \beta \setminus \mathsf{rr}\ \beta) \neq \emptyset$, i.e., some subformula α range-restricts some variable that is free but not range-restricted in another subformula β, the monitors copy α next to β using conjunction (often under some temporal operators).

Existing rules. The following rules are used in VeriMon and MonPoly. They push a range restriction from an outer conjunct α into a temporal subformula by adding an "opposite" temporal operator. The opposite operator overapproximates α's satisfactions: it ensures that the outer time-point lies within the temporal window of the added operator, so α is captured at the current time. We write $I = [a,b]$ for an arbitrary bounded interval, $J = [0,b]$ for the corresponding initial interval, and use standard operators once $\blacklozenge_I$, eventually $\Diamond_I$, historically $\blacksquare_I$, and always $\Box_I$ derived from S_I and U_I (and $\neg$).

$$
\begin{array}{rlcl}
1. & \alpha \wedge (\beta\, \mathsf{S}_I\, \gamma) & \to & \alpha \wedge (((\Diamond_J\, \alpha) \wedge \beta)\, \mathsf{S}_I\, \gamma) \\
2. & \alpha \wedge (\beta\, \mathsf{U}_I\, \gamma) & \to & \alpha \wedge (((\blacklozenge_J\, \alpha) \wedge \beta)\, \mathsf{U}_I\, \gamma) \\
3. & \alpha \wedge (\gamma\, \mathsf{S}_I\, \beta) & \to & \alpha \wedge (\gamma\, \mathsf{S}_I\, ((\Diamond_I\, \alpha) \wedge \beta)) \\
4. & \alpha \wedge (\gamma\, \mathsf{U}_I\, \beta) & \to & \alpha \wedge (\gamma\, \mathsf{U}_I\, ((\blacklozenge_I\, \alpha) \wedge \beta)) \\
5. & \alpha \wedge \blacklozenge_I\, \beta & \to & \alpha \wedge \blacklozenge_I\, ((\Diamond_I\, \alpha) \wedge \beta) \\
6. & \alpha \wedge \Diamond_I\, \beta & \to & \alpha \wedge \Diamond_I\, ((\blacklozenge_I\, \alpha) \wedge \beta) \\
7. & \alpha \wedge \blacksquare_I\, \beta & \to & \alpha \wedge \blacksquare_I\, ((\Diamond_I\, \alpha) \wedge \beta) \\
8. & \alpha \wedge \Box_I\, \beta & \to & \alpha \wedge \Box_I\, ((\blacklozenge_I\, \alpha) \wedge \beta) \\
9. & \alpha \wedge \bullet_I\, \beta & \to & \alpha \wedge \bullet_I\, ((\bigcirc_I\, \alpha) \wedge \beta) \\
10. & \alpha \wedge \bigcirc_I\, \beta & \to & \alpha \wedge \bigcirc_I\, ((\bullet_I\, \alpha) \wedge \beta)
\end{array}
$$

We focus on a subset of rules proposed by Müller [16] and our numbering deviates from his. His other rules either lack temporal operator altogether, or have α appear *under* a temporal operator, e.g., in the second argument of S_I, where it is copied into the first argument. Müller further distinguishes temporal and strict temporal operators, which force the operator to reach into the past or future. (We could model this by requiring the interval I to exclude 0, i.e., $a > 0$.) All these aspects are not relevant for our discussion, though.

Alternative rules using Frz. Each rule above has an equivalent formulation using the Frz operator. Instead of adding a temporal conjunct, we *freeze* α at the outer time-point into a fresh predicate q, and replace the temporal conjunct by q inside the operator. (Strictly speaking, we would need to write $q(x,y,z)$ if $\mathsf{fv}\,\alpha = \{x,y,z\}$.)

$$
\begin{array}{rlcl}
1'. & \alpha \wedge (\beta\, \mathsf{S}_I\, \gamma) & \rightarrow & \mathsf{Frz}\, q := \alpha \,\mathsf{in}\, \alpha \wedge ((q \wedge \beta)\, \mathsf{S}_I\, \gamma) \\
2'. & \alpha \wedge (\beta\, \mathsf{U}_I\, \gamma) & \rightarrow & \mathsf{Frz}\, q := \alpha \,\mathsf{in}\, \alpha \wedge ((q \wedge \beta)\, \mathsf{U}_I\, \gamma) \\
3'. & \alpha \wedge (\gamma\, \mathsf{S}_I\, \beta) & \rightarrow & \mathsf{Frz}\, q := \alpha \,\mathsf{in}\, \alpha \wedge (\gamma\, \mathsf{S}_I\, (q \wedge \beta)) \\
4'. & \alpha \wedge (\gamma\, \mathsf{U}_I\, \beta) & \rightarrow & \mathsf{Frz}\, q := \alpha \,\mathsf{in}\, \alpha \wedge (\gamma\, \mathsf{U}_I\, (q \wedge \beta)) \\
5'. & \alpha \wedge \blacklozenge_I\, \beta & \rightarrow & \mathsf{Frz}\, q := \alpha \,\mathsf{in}\, \alpha \wedge \blacklozenge_I\, (q \wedge \beta) \\
6'. & \alpha \wedge \lozenge_I\, \beta & \rightarrow & \mathsf{Frz}\, q := \alpha \,\mathsf{in}\, \alpha \wedge \lozenge_I\, (q \wedge \beta) \\
7'. & \alpha \wedge \blacksquare_I\, \beta & \rightarrow & \mathsf{Frz}\, q := \alpha \,\mathsf{in}\, \alpha \wedge \blacksquare_I\, (q \wedge \beta) \\
8'. & \alpha \wedge \square_I\, \beta & \rightarrow & \mathsf{Frz}\, q := \alpha \,\mathsf{in}\, \alpha \wedge \square_I\, (q \wedge \beta) \\
9'. & \alpha \wedge \bullet_I\, \beta & \rightarrow & \mathsf{Frz}\, q := \alpha \,\mathsf{in}\, \alpha \wedge \bullet_I\, (q \wedge \beta) \\
10'. & \alpha \wedge \circ_I\, \beta & \rightarrow & \mathsf{Frz}\, q := \alpha \,\mathsf{in}\, \alpha \wedge \circ_I\, (q \wedge \beta)
\end{array}
$$

The Frz formulation is arguably more readable: q is explicitly the snapshot of α at the current time-point, and the temporal operator is applied to the unmodified β conjoined with that snapshot—no overapproximation needed. Unlike the existing rules, the Frz formulation also does not change the monitor's overall progress for the rewritten formulas compared to the original formulas. However, while both rule sets produce equivalent formulas in RANF, Section 6 shows that the Frz variants may incur a significant runtime cost.

5 Implementation

The support for the Frz operator is implemented in VeriMon and MonPoly. VeriMon follows the algorithm of Section 3 verbatim. The implementation is formalized in Isabelle and we extract OCaml code from it via Isabelle's code generator [11].

The MonPoly implementation evaluates the same specification but introduces several algorithmic refinements that are not present in the verified algorithm. At formula-translation time it inspects the body β and dispatches to one of two modes.

Atemporal mode. If β contains no temporal operator, Frz coincides with Let. Moreover, the frozen relation can be computed once and used without any complication in β whenever needed. Thus, the monitor only installs the frozen relation as a constant binding for the predicate symbol (p,m), evaluates β once on the current database, and removes the binding. The snapshot queue and diagonal replay machinery of Section 3 is bypassed.

Temporal mode. If β contains a temporal operator, the frozen relation must remain visible to β at all queried past or future time-points. Here the native monitor departs from the verified algorithm: instead of maintaining one shared β-monitor and replaying each pending snapshot α_i through it from scratch, it allocates a *separate persistent* β*-monitor* s_β^i for each outer time-point k_i and advances s_β^i incrementally as the trace grows. The diagonal verdict—the output of s_β^i at the time-point k_i itself—is collected as

soon as s^i_β reaches k_i; at all other positions the monitor discards the computed relation but advances its temporal state. When β contains only bounded-past operators, a static analysis computes the maximum temporal depth d — the furthest time-stamp-distance β can look into the past. Each s^i_β then replays only the trace suffix that falls within the relevant time-stamp window $[\mathsf{T}\,\sigma\,k_i - d, \mathsf{T}\,\sigma\,k_i]$.

Differential testing. These optimizations are not verified in Isabelle. To gain confidence in their correctness, we perform differential testing against the verified, but less efficient VeriMon algorithm. To this end, every formula and trace in our test suite is evaluated by both algorithms on the same input and the two output streams are compared for exact equality. The suite comprises 79 cases organized into five families: atemporal freezing; frozen relation used under the temporal operators S, U, $\blacklozenge$, and $\lozenge$ (exercising the temporal mode); nesting (double and triple nesting of Frz, as well as mixtures with Let); freezing complex formulas; and regression tests tracing specific issues found during development. All tests pass; maintaining this is an invariant of our development process.

LLM-assisted development. Both algorithms were developed with the assistance of state-of-the-art large-language models (LLMs), notably Claude Sonnet 4.6, DeepSeek V4 Flash, and Claude Opus 4.8. For VeriMon, we had specified the semantics of Frz and had supplied the core structure of the algorithm, but left figuring out the progress bookkeeping details to the LLM. Moreover, VeriMon's overall correctness statement was fixed in prior work. The LLM generated most of the Isabelle proof for the new Frz case and proposed and proved key auxiliary lemmas—notably for progress arithmetic. The work was interactive: the human guided and oversaw, while the model filled in the details.

This setup provides a form of *logical safety*: the proof assistant guarantees that the logical reasoning steps are correct, while the LLM supplies productivity. Naturally, one needs to validate manually that the correctness statement is the desired monitor specification; however, this statement has not changed compared to prior work and has been carefully scrutinized as part of that work, leaving no room for LLM hallucinations.

The MonPoly implementation was developed largely autonomously by Claude Opus 4.8, starting from the verified VeriMon algorithm but soon departing from it to improve efficiency. The two modes, the differential test suite, and the integration with MonPoly's existing rewriting and parsing infrastructure were all produced through iterative prompting—a development style sometimes called *vibe coding*—with the instruction to use VeriMon as a verified oracle. Whenever the LLM proposed a change, the test suite ran both kernels side by side and caught any mismatches before they reached the codebase. While the guarantees here are far from logical safety, differential testing and manual code review provide some assurance that the MonPoly implementation is correct.

6 Evaluation

We evaluate VeriMon's and MonPoly's algorithmic extensions for the freeze operator using MonitoringFace [18] platform designed for portable and reproducible evaluation of runtime monitors. The four MonitoringFace configuration files used to obtain the results from this section are publicly available [14]. Each file corresponds to an experiment designed to answer one of our research questions (RQs).

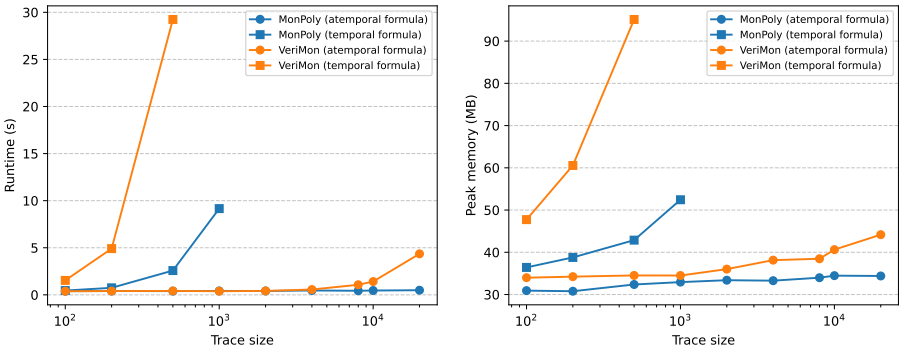

Fig. 2. RQ1: Runtime and memory usage vs. trace size

Experimental Setup. We fix two unary events $a(x)$ and $b(x)$ both with an integer parameter in all our experiments. MonitoringFace provides a trace generator [13] that takes the event description and generates a random trace with consecutive time-stamps, and configurable event rate (10 in our case) and overall trace size. It determines parameter values by maintaining a set of unique most recently sampled values with configurable size (100 in our case) and samples from this set with a configurable probability (0.1 in our case) ensuring some common parameter values across events. As no other tool supports the (first-order) freeze operator, we focus only on our extensions and compare them.

MonitoringFace measures the monitors' runtime in sections (s) and peak memory usage in megabytes (MB) using GNU `time` (`%e` and `%M` fields). We have set a timeout for all the monitors to 30 seconds and we mark such runs with an $\times$ symbol. We ran all the experiments on an Apple M3 Pro 12-core CPU with 36 GB RAM.

RQ1: How does the performance of Frz *scale with the trace size?* To answer we monitor the atemporal Frz $q(x) := a(x)$ in $q(x)$ and the temporal Frz $q(x) := a(x)$ in $a(x) \wedge ((q(x) \wedge b(y) \wedge x > 0)\, \mathsf{U}_{[0,5]}\, (a(x) \wedge b(x)))$ formula on traces with 1K to 20K time-points.

Fig. 2 shows the results: except for the MonPoly runs on atemporal formula, time grows roughly quadratically for all runs: doubling the trace size approximately quadruples the monitoring time. VeriMon also times out for traces with more than 40K time-points. This is a consequence of the Frz evaluation scheme: at each time-point i the frozen snapshot $q := a$ at i changes, so the β-monitor cannot reuse state from the previous step and performs an amount of work proportional to the current progress.

RQ2: How does the performance scale with formula complexity? We fix the trace size to 1K time-points and vary the formula: *simple* (Frz $q(x) := a(x)$ in $q(x)$), *medium* (Frz $q(x) := a(x)$ in $q(x) \wedge b(x)$), and *complex* (Frz $q(x) := a(x)$ in $\blacklozenge_{[0,5]}\, q(x)$).

The results are shown in Fig. 3: adding another conjunct inside β (*medium*) already significantly increases the VeriMon's runtime compared to the formula with the trivial body (*simple*). Nesting β inside a bounded $\blacklozenge$ operator (*complex*) even further increases the runtime overhead, as the operator must replay the frozen snapshot over a sliding window.

RQ3: How does Frz *compare to other temporal operators?* To answer this question we monitor the formula $op(x) \wedge b(x)$ over traces ranging from 100 to 2K time-points and

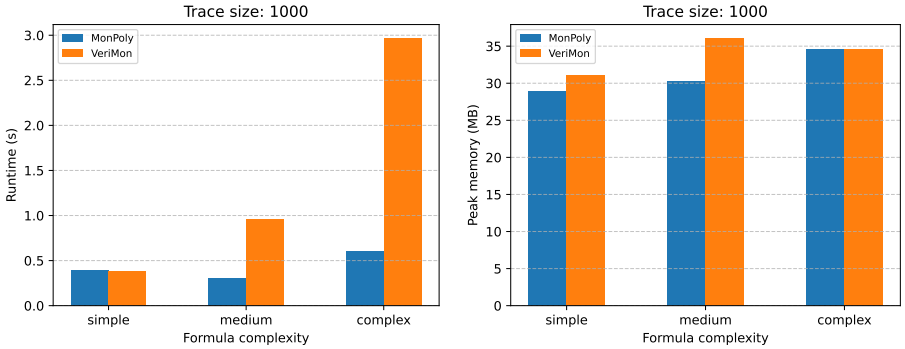

Fig. 3. RQ2: Runtime and memory usage vs. formula complexity.

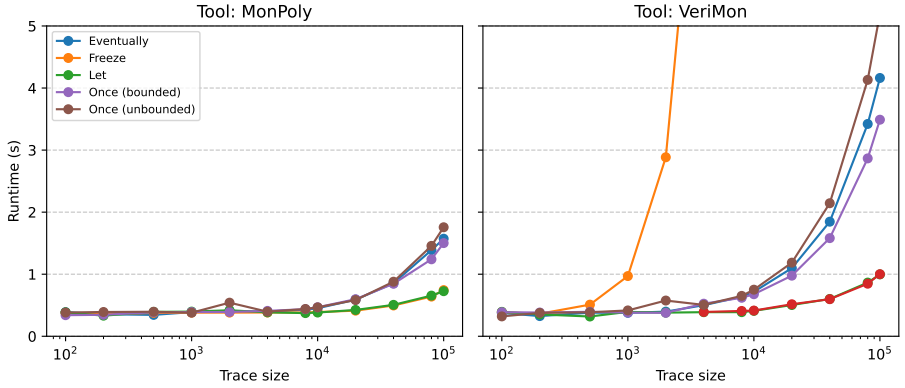

Fig. 4. RQ3: Runtime for five operator variants vs. trace size.

select the *op* operator from one of $\Diamond_{[0,100]}$, Frz, Let, $\blacklozenge$, and $\blacklozenge_{[0,100]}$ applied on the $a(x)$ predicate. The Frz and Let operators simply bind the predicate to a fresh one.

Fig. 4 shows plots for each tool separately focusing on the monitor runtimes only. All operators except VeriMon's Frz scale linearly (note the log-scale x-axis) and finish in under 0.6 s even for large traces. To improve readability for other operators, Fig. 4's y-axis is bounded to 5 seconds omitting that VeriMonruns for 9.3 seconds for Frz on the trace with 4K time-point and subsequently times out.

RQ4: Is the Frz *rewrite more efficient than the state-of-the-art?* To assess this question, for each of the ten rule pairs (rules 1–10 and their Frz counterparts $1'$–$10'$ from Section 4), (1) we construct a non-RANF formula; (2) apply both rewrites; and (3) monitor the resulting formulas against the same trace. For step (1), we used the following patterns:

$$\mathsf{a}(x) \wedge \mathsf{OP}^1\ (\mathsf{b}(y) \wedge x > 0), \tag{1}$$

$$\mathsf{a}(x) \wedge (\mathsf{c}(x)\ \mathsf{OP}^2\ (\mathsf{b}(y) \wedge x > 0)), \text{ or} \tag{2}$$

$$\mathsf{a}(x) \wedge ((\mathsf{b}(y) \wedge x > 0)\ \mathsf{OP}^2\ \mathsf{c}(x)), \tag{3}$$

where $\mathsf{OP}^1 \in \{\blacklozenge_I, \Diamond_I, \blacksquare_I, \Box_I, \bullet_I, \circ_I\}$ is a unary operator, $\mathsf{OP}^2 \in \{\mathsf{S}_I, \mathsf{U}_I\}$ is a binary operator, and $\mathsf{b}(y) \wedge x > 0$ needs to be range restricted by $\mathsf{a}(x)$. Note that all formulas are

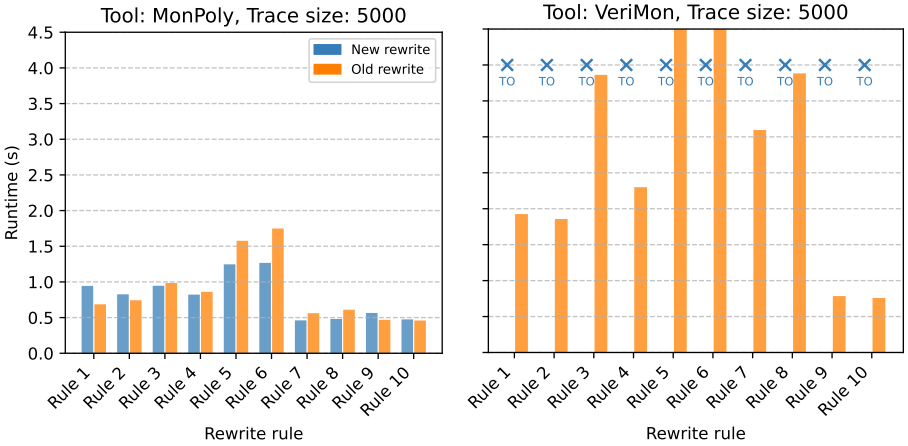

Fig. 5. RQ4: Comparing rewrites to RANF with MonPoly and Verimon.

initially not in RANF due to the unguarded variable x in the $x > 0$ subformula. When instantiated with the listed operators, these patterns cover preciselt the ten rules.

Figure 5 shows the runtime for each rule the runtime of monitoring the two alternatively-rewritten RANF formulas on a trace with size 5K. For MonPoly the Frz rewrite is comparable to the old rewrite and even more efficient for some rules.

7 Conclusion

We introduced the freeze operator Frz in MFOTL, as a minimal semantic modification of the non-recursive Let operator. Despite the small semantics change, the algorithmic implications are substantial. Our algorithm is implemented and proved correct in the Isabelle proof assistant, yielding VeriMon via code extraction, and reimplemented in MonPoly with performance optimizations.

Our evaluation shows that VeriMon's Frz algorithm scales quadratically with trace size, whereas MonPoly's optimizations fare better. We further introduce Frz-based rewrite rules for RANF that preserve the monitor's progress (unlike existing rules), at a runtime cost that is comparable to the original rewrites in most cases. Our development process itself demonstrates an emerging paradigm: large portions of the Isabelle proof and the entire MonPoly extension were produced with LLM assistance, safeguarded by Isabelle via proof checking and the verified VeriMon oracle via differential testing.

Several directions merit future investigation. First, the verified algorithm's diagonal replay can be optimized to avoid the wasteful replicate in favor of a constant relation marker, directly addressing a source of the quadratic cost. Second, MonPoly's bounded-past optimization (the temporal depth analysis) currently applies only to bodies without future operators; extending it to handle future operators is a natural engineering goal. Finally, a theoretical study of Frz's expressiveness and conciseness is worthwhile. Our rewrite rules already indicate that Frz can express constraints more directly, but a systematic study would clarify the role of Frz in the MFOTL landscape.

Acknowledgments. Traytel is supported by the Copilots for Isabelle project, funded by the AI for Math Fund, an initiative by Renaissance Philanthropy with support from XTX Markets. The Isabelle autoformalization of the freeze operator using LLMs has been carried out as a case study for the infrastructure developed in this project.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Alur, R., Henzinger, T.A.: A really temporal logic. J. ACM **41**(1), 181–204 (1994). https://doi.org/10.1145/174644.174651
3. Basin, D., Dardinier, T., Heimes, L., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12166, pp. 432–453. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_25
4. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal Methods Syst. Des. **46**, 262–285 (2015)
5. 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
6. 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). https://doi.org/10.29007/89HS
7. Basin, D., Klaedtke, F., Zălinescu, E.: Runtime verification of temporal properties over out-of-order data streams. In: Majumdar, R., Kuncak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 356–376. Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_18
8. Basin, D., Krstić, S., Schneider, J., Traytel, D.: Correct and efficient policy monitoring, a retrospective. In: ATVA 2023. pp. 3–30. Springer (2023)
9. Christensen, C.L.: Reducing VeriMon's Trusted Code Base. B.Sc. thesis, University of Copenhagen (2025)
10. Demri, S., Lazic, R.: LTL with the freeze quantifier and register automata. ACM Trans. Comput. Log. **10**(3), 16:1–16:30 (2009). https://doi.org/10.1145/1507244.1507246
11. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer (2010). https://doi.org/10.1007/978-3-642-12251-4_9
12. Havelund, K., Peled, D., Ulus, D.: Dejavu: A monitoring tool for first-order temporal logic. In: 3rd Workshop on Monitoring and Testing of Cyber-Physical Systems, MT@CPSWeek 2018, Porto, Portugal, April 10, 2018. pp. 12–13. IEEE (2018). https://doi.org/10.1109/MT-CPS.2018.00013
13. Krstić, S., Schneider, J.: A benchmark generator for online first-order monitoring. In: Deshmukh, J., Nickovic, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 482–494. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_27
14. Krstić, S., Traytel, D.: MonitoringFace configurations for the freeze operator extension. https://github.com/MonitoringFace-Benchmark/MonitoringFace/tree/main/Archive/Experiments/extensions (2026)
15. Lima, L., Huerta y Munive, J.J., Traytel, D.: WhyMon: A runtime monitoring tool with explanations as verdicts. In: Akshay, S., Niemetz, A., Sankaranarayanan, S. (eds.) ATVA 2024. LNCS, vol. 15055, pp. 76–90. Springer (2024). https://doi.org/10.1007/978-3-031-78750-8_4
16. Müller, S.: Theory and applications of runtime monitoring metric first-order temporal logic. Ph.D. thesis, ETH Zurich (2009). https://doi.org/10.3929/ethz-a-005932651

17. Raszyk, M.: Efficient, Expressive, and Verified Temporal Query Evaluation. Ph.D. thesis, ETH Zurich (2022). https://doi.org/10.3929/ethz-b-000553221
18. Reese, L., Hublet, F., Krstić, S., Traytel, D.: MonitoringFace: A portable platform for runtime monitoring. https://github.com/MonitoringFace-Benchmark/MonitoringFace (2026)
19. Reese, L., Silva, R.C.G., Traytel, D.: TimelyMon: A streaming parallel first-order monitor. In: Ábrahám, E., Abbas, H. (eds.) RV 2024. LNCS, vol. 15191, pp. 150–160. Springer (2024). https://doi.org/10.1007/978-3-031-74234-7_9
20. 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 (2019). https://doi.org/10.1007/978-3-030-32079-9_18
21. Zingg, S., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: Verified first-order monitoring with recursive rules. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 236–253. Springer (2022)
22. Zingg, S., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: VeriMon's development repository. https://bitbucket.org/jshs/monpoly/src/887b996966/thys/ (2021)