# Verified First-Order Monitoring with Recursive Rules

Sheila Zingg[1], Srđan Krstić[1](✉), Martin Raszyk[1](✉), Joshua Schneider[1](✉), and Dmitriy Traytel[2](✉)

[1] Institute of Information Security, Department of Computer Science, ETH Zürich, Zurich, Switzerland, {srdan.krstic,martin.raszyk,joshua.schneider}@inf.ethz.ch
[2] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, traytel@di.ku.dk

**Abstract.** First-order temporal logics and rule-based formalisms are two popular families of specification languages for monitoring. Each family has its advantages and only few monitoring tools support their combination. We extend metric first-order temporal logic (MFOTL) with a recursive let construct, which enables interleaving rules with temporal logic formulas. We also extend VeriMon, an MFOTL monitor whose correctness has been formally verified using the Isabelle proof assistant, to support the new construct. The extended correctness proof covers the interaction of the new construct with the existing verified algorithm, which is subtle due to the presence of the bounded future temporal operators. We demonstrate the recursive let's usefulness on several example specifications and evaluate our verified algorithm's performance against the DejaVu monitoring tool.

**Keywords:** Rule-based specifications · Monitoring · Formal verification.

## 1 Introduction

In runtime verification, a monitor observes events generated by a running system and analyzes the event streams for compliance with a given specification. Temporal specification languages for monitoring are often classified as operational or declarative [10]. Operational languages explicitly describe how the monitor's input should be transformed to obtain an output. Two important subclasses of operational languages are rule-based formalisms [2,13] and stream runtime verification (SRV) languages [6,8,11,20]. Both formulate the transformations as recursive equations. In contrast, declarative languages, such as first-order temporal logics [4,15], describe the output by composing high-level operators.

Operational and declarative languages have complementary advantages: declarative languages let specification authors focus on the "what" and not the "how", whereas operational languages offer the authors more control over the evaluation. Most runtime verification tools do not support mixing the paradigms, especially when it comes to parametric, i.e., first-order, specification languages. A notable exception is the recent addition of recursive rules to past-time first-order temporal logic (PFLTL), implemented in the DejaVu monitoring tool [14]. As another important benefit, recursive rules can express operations like transitive closure that are not expressible in first-order logics.

In this paper, we introduce recursion in metric first-order temporal logic (MFOTL) [4] in the form of a recursive let construct. We develop and implement an evaluation algorithm for MFOTL with recursion in VeriMon [3, 21], an MFOTL monitor whose correctness has been formally verified in the Isabelle proof assistant. To this end, we extend the formal correctness proof to cover the recursive let construct.

Unlike PFLTL, MFOTL supports bounded future temporal operators and aggregations (Section 2). The interaction of recursion with bounded future operators is subtle. To avoid non-termination, DejaVu requires all recursive occurrences to be guarded by a previous operator. We similarly require the recursive occurrences to be guarded in our monitor, but we relax the requirement on the guard to other past-time operators which ensure that their subformulas are evaluated strictly in the past. Moreover, we allow future operators in the recursive let construct, as long as no recursion takes place in the future operator's arguments. These restrictions ensure that the fixpoint given by the recursive let operator is well-defined. At the same time, they are permissive and allow us to formulate interesting examples, several of which are beyond what PFLTL with recursion can express.

Consider a specification that aims to secure hosts in a network that communicate with each other and with the outside world. A host is *tainted* by an address range iff there is a chain of communication from the address to the host and all hosts on the chain trigger an intrusion detection alert within one hour after communicating with the previous host. This specification can be expressed directly using our recursive let construct (to model chains of communication) and future temporal operators (to specify "within one hour after").

We start by extending MFOTL with a non-recursive let operator (Section 3). This special case is mainly of pedagogical value: aspects common to both let operators are easier to explain on the simpler non-recursive variant. Yet, this construct is useful in practice to structure complex formulas and improve monitoring performance by sharing common subformulas. Thus we extend VeriMon's algorithms and proofs with the non-recursive let.

We then introduce the recursive let operator (Section 4.1), exemplify its semantics with several specifications (Section 4.2), and develop the monitoring algorithm and sketch its correctness (Section 4.3). VeriMon's repository [24] contains complete formal proofs.

This work is part of the long-term effort to develop a trustworthy monitor that surpasses in expressiveness and efficiency other non-verified tools. In this work, our focus is on expressiveness (and trustworthiness). Nonetheless, we evaluate our algorithmic additions to VeriMon on a micro-benchmark and observe that even without further optimizations it exhibits an incomparable performance to DejaVu (Section 5). Moreover, we detected a problem in DejaVu's handling of variable names in recursive subformulas.

In summary, our main contribution is the extension of MFOTL with a recursive let operator and the design of an evaluation algorithm for it. Along the way, we introduce a non-recursive let operator, which proved essential when writing complex specifications. Our contributions are implemented as part of VeriMon and proved correct using Isabelle.

*Related Work.* Our work adds rule-based specification features [13] to a first-order specification language [16]. Above we describe our contribution's relationship to DejaVu and VeriMon, two monitors for first-order temporal specifications. VeriMon's algorithm [21], which we extend, is based on the algorithm used in the MonPoly monitor [5], although VeriMon has optimizations that are not present in MonPoly and vice versa [3]. VeriMon supports a more expressive specification language than MonPoly, and our introduction of the recursive let has increased the gap between the two. VeriMon's and MonPoly's algorithms work with finite relations. These tools are thus restricted to MFOTL's *monitorable fragment* [4], which ensures that all subformulas evaluate to finite results. In contrast, DejaVu finitely represents infinite relations using BDDs and thus supports the full PFLTL (but only closed formulas). Both DejaVu and our work restrict the recursive let syntactically.

**datatype** $data = \mathsf{Int}\ int \mid \mathsf{Flt}\ double \mid \mathsf{Str}\ string$ **type_synonym** $ts = nat$
**type_synonym** $db = string \Rightarrow data\ list\ set$ **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** $frm = string(trm\ list) \mid trm \circ trm \mid \neg\, frm \mid \exists\, frm \mid frm \vee frm \mid frm \wedge frm$
$\mid \bullet_{\mathcal{I}}\ frm \mid \bigcirc_{\mathcal{I}}\ frm \mid frm\ \mathsf{S}_{\mathcal{I}}\ frm \mid frm\ \mathsf{U}_{\mathcal{I}}\ frm \mid nat \leftarrow agg\_op(trm; nat)\ frm$

**fun** $\mathsf{etrm} :: data\ list \Rightarrow trm \Rightarrow data$ **where**
$\mathsf{etrm}\ v\ (\mathsf{V}\ x) = v\,!\,x \mid \mathsf{etrm}\ v\ (\mathsf{C}\ x) = x \mid \mathsf{etrm}\ v\ (t_1 + t_2) = \mathsf{etrm}\ v\ t_1 + \mathsf{etrm}\ v\ t_2 \mid \ldots$

**fun** $\mathsf{sat} :: trace \Rightarrow data\ list \Rightarrow nat \Rightarrow frm \Rightarrow bool$ **where**
$\mathsf{sat}\ \sigma\ v\ i\ (p(as)) = (\mathsf{map}\ (\mathsf{etrm}\ v)\ as \in \Gamma\ \sigma\ i\ p) \mid \mathsf{sat}\ \sigma\ v\ i\ (t_1 \circ t_2) = (\mathsf{etrm}\ v\ t_1 \circ \mathsf{etrm}\ v\ t_2)$
$\mid \mathsf{sat}\ \sigma\ v\ i\ (\neg\varphi) = (\neg \mathsf{sat}\ \sigma\ v\ i\ \varphi) \mid \mathsf{sat}\ \sigma\ v\ i\ (\exists\varphi) = (\exists z.\ \mathsf{sat}\ \sigma\ (z\#v)\ i\ \varphi)$
$\mid \mathsf{sat}\ \sigma\ v\ i\ (\alpha \vee \beta) = (\mathsf{sat}\ \sigma\ v\ i\ \alpha \vee \mathsf{sat}\ \sigma\ v\ i\ \beta) \mid \mathsf{sat}\ \sigma\ v\ i\ (\alpha \wedge \beta) = (\mathsf{sat}\ \sigma\ v\ i\ \alpha \wedge \mathsf{sat}\ \sigma\ v\ i\ \beta)$
$\mid \mathsf{sat}\ \sigma\ v\ i\ (\bullet_I \varphi) = (\underline{\mathsf{case}}\ i\ \underline{\mathsf{of}}\ 0 \Rightarrow \mathsf{False} \mid j+1 \Rightarrow \mathsf{T}\ \sigma\ i - \mathsf{T}\ \sigma\ j \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ v\ j\ \varphi)$
$\mid \mathsf{sat}\ \sigma\ v\ i\ (\bigcirc_I \varphi) = (\mathsf{T}\ \sigma\ (i+1) - \mathsf{T}\ \sigma\ i \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ v\ (i+1)\ \varphi)$
$\mid \mathsf{sat}\ \sigma\ v\ i\ (\alpha\, \mathsf{S}_I \beta) = (\exists j \leq i.\ \mathsf{T}\ \sigma\ i - \mathsf{T}\ \sigma\ j \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ v\ j\ \beta \wedge (\forall k \in \{j <.. i\}.\ \mathsf{sat}\ \sigma\ v\ k\ \alpha))$
$\mid \mathsf{sat}\ \sigma\ v\ i\ (\alpha\, \mathsf{U}_I \beta) = (\exists j \geq i.\ \mathsf{T}\ \sigma\ j - \mathsf{T}\ \sigma\ i \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ v\ j\ \beta \wedge (\forall k \in \{i ..< j\}.\ \mathsf{sat}\ \sigma\ v\ k\ \alpha))$
$\mid \mathsf{sat}\ \sigma\ v\ i\ (y \leftarrow \Omega(t; b)\ \varphi) = \big(\underline{\mathsf{let}}\ M = \{(x, \mathsf{card}^{\infty}\ Z) \mid x\ Z.$
$Z = \{z.\ \mathsf{length}\ z = b \wedge \mathsf{sat}\ \sigma\ (z\ @\ v)\ i\ \varphi \wedge \mathsf{etrm}\ (z\ @\ v)\ t = x\} \wedge Z \neq \{\}\}$
$\underline{\mathsf{in}}\ (M = \{\} \longrightarrow \mathsf{fv}\ \varphi \subseteq \{0 ..< b\}) \wedge v\,!\,y = \mathsf{eval\_agg\_op}\ \Omega\ M)$

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

Other rule-based [2,13] and SRV-based monitors [6,8,11,20] can express the temporal operators present in LTL, but struggle with extensions that introduce parameters. Even for the operators they can express, specialized algorithms that are carefully tuned for the operators tend to exhibit a better performance. Instead of encoding temporal operators, we take the opposite approach and enrich a monitor that uses specialized algorithms for temporal operators with general-purpose recursion.

Datalog [1] adds recursion to first-order logic, similarly to our addition of recursion to temporal logic. However, Datalog has no built-in notion of time and hence other measures must be taken to ensure that the fixpoints are well-defined, e.g., by restricting negation. Restricting the recursive occurrences to be strictly in the past is a natural and expressive alternative for monitoring, as we do not restrict negation beyond of what the monitorable fragment requires. Works on Datalog extensions with metric temporal operators [7,19,22] mostly study the decidability and complexity of computational problems related to these extensions, whereas we design, implement, and formally verify an executable algorithm.

## 2 Metric First-Order Temporal Logic

MFOTL extends linear temporal logic with first-order quantification, past-time operators, and interval bounds on the temporal operators [4]. The VeriMon monitor [3] supports a fragment of this logic. It also adds new features, specifically regular matching operators as in linear dynamic logic [9], which results in metric first-order dynamic logic (MFODL), as well as aggregations. Our extension of VeriMon with recursive rules retains the additional features of MFODL. However, the additional features are orthogonal to our extension and hence we base our presentation in this paper on MFOTL with aggregations.

We summarize MFOTL's syntax and semantics, as well as the monitorable fragment. The presentation generally follows the Isabelle formalization; however, we sometimes

deviate from Isabelle's concrete syntax for simplicity. We begin by defining some auxiliary types (top of Fig. 1). The logic's universe (type *data*) is fixed and infinite: it is a disjoint sum of integers, 64-bit IEEE floats, and strings of 8-bit characters. Databases (type *db*) encode first-order structures as functions from predicate names to relations over *data*. Relations are represented as sets of lists. A *trace* is a *stream* (an infinite sequence) of time-stamped databases. Time-stamps (type *ts*) are modeled as natural numbers (type *nat*). We write $\Gamma\ \sigma\ i$ for the $i$th database in $\sigma$, and $\mathsf{T}\ \sigma\ i$ for its time-stamp. The predicate trace enforces monotone and eventually increasing time-stamps, i.e., $\forall i \leq j.\ \mathsf{T}\ \sigma\ i \leq \mathsf{T}\ \sigma\ j$ and $\forall x.\ \exists i.\ x < \mathsf{T}\ \sigma\ i$. Non-empty intervals (type $\mathcal{I}$) are represented by their end-points. We write $[a,b]$ for the unique interval satisfying $n \in_{\mathcal{I}} [a,b]$ iff $a \leq n \leq b$, where $n \in_{\mathcal{I}} I$ denotes that $I$ contains the natural number $n$. The interval is unbounded from above if $b = \infty$, which the type *enat* adds to the natural numbers.

Terms (type *trm*) are constructed recursively from variables (represented by De Bruijn indices), constants, and arithmetic operators. We use named variables in examples and omit the V and C constructors. There are two kinds of atomic formulas (type *frm*): flexible predicates of the form $p(as)$, where $as$ is a list of terms, and rigid predicates $t_1 \circ t_2$ for $\circ \in \{=, <, \leq\}$, which have a fixed interpretation. Formally, the existential quantifier $\exists$ does not carry a variable name because of the De Bruijn encoding. We use $\mathsf{fv}\ \alpha$ to denote the set of De Bruijn indices of $\alpha$'s free variables.

The semantics is given by the functions etrm and sat (Fig. 1). Both depend on a valuation, which is a *data list* assigning a value to each variable. The satisfaction function sat for formulas additionally depends on a trace $\sigma$ and a time-point $i$, which is an index into the trace. Indexing into lists is denoted by $v\,!\,x$, the operation $z\#v$ prepends the value $z$ to the list $v$, and @ concatenates two lists. The notation $\{x\ ..< y\}$ and $\{x <.. y\}$ is shorthand for the sets $\{x, x+1, \ldots, y-1\}$ and $\{x+1, x+2, \ldots, y\}$ of natural numbers, respectively.

An aggregation formula $y \leftarrow \Omega(t;b)\ \varphi$ binds $b$ variables in the subformula $\varphi$; the remaining free variables of $\varphi$ are used for grouping. Each group is assigned an aggregate value $y$, which is computed by first evaluating the term $t$ on each valuation that matches the group and that satisfies $\varphi$, then aggregating the results using the operator $\Omega$ (e.g., MIN for minimum). To this end, eval_agg_op $\Omega\ M$ (not shown) applies $\Omega$ to a set $M$ of value–multiplicity pairs [3]; $\mathsf{card}^{\infty}\ Z$ is the cardinality of $Z$, or $\infty$ if $Z$ is infinite. The conjunct $M = \{\} \longrightarrow \mathsf{fv}\ \varphi \subseteq \{0\ ..< b\}$ ensures that the formula is satisfied by the aggregate value of an empty $M$ only if there are no grouping variables. Otherwise, infinitely many groups would be labeled with that value, rendering such aggregations non-monitorable.

The decidable predicate mon :: *frm* ⇒ *bool* specifies the monitorable fragment. We omit its formal definition and refer to the earlier descriptions of VeriMon [3,21] for details. Intuitively, mon places restrictions on the formula's structure to ensure that all subformulas have finitely many satisfying valuations. Also, the interval $I$ of every $\mathsf{U}_I$ operator must be bounded. A monitor for a monitorable formula can thus compute a finite set of satisfying valuations for every time-point after observing a sufficiently long trace prefix.

## 3 Non-Recursive Let Operator

We first introduce a non-recursive let operator Let *string* := *frm* in *frm* to the *frm* datatype. The formula Let $p := \alpha$ in $\beta$ associates the formula $\alpha$ with the predicate named $p$, which may be used in the formula $\beta$. We call such a predicate *let-bound*. The operator is

non-recursive: $p$ has the same meaning within $\alpha$ as in the surrounding context (unless it is bound by a nested let in $\alpha$). Although the non-recursive let operator does not enhance MFOTL's expressiveness, it improves readability (by using descriptive let-bound predicate names), as well as modularity and evaluation efficiency (by sharing subformulas).

Intuitively, the meaning of $\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta$ is the same as that of $\beta$ after replacing all its predicates of the form $p(as)$ with the formula $\alpha$, whose free variables have been replaced with the terms $as$ in a capture-avoiding way. The formal syntax does not specify explicitly how $\alpha$'s free variables map to $p$'s arguments. The mapping is induced by the De Bruijn indices: the variable with index 0 becomes the first argument, and so forth. We list the arguments explicitly in examples that use named variables. For instance, the formula $\mathsf{Let}\ p(x) \coloneqq p(x) \wedge \exists y.\ q(x,y)\ \mathsf{in}\ \bullet_{[0,2]}\, p(y)$ should be equivalent to $\bullet_{[0,2]}(p(y) \wedge \exists z.\ q(y,z))$. We achieve this by defining Let's semantics as follows.

$$\mathsf{sat}\ \sigma\ v\ i\ (\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta) = \mathsf{sat}\ (\sigma[p \Rrightarrow \lambda j.\ \mathsf{satrel}\ \sigma\ j\ \alpha])\ v\ i\ \beta$$

We write $\mathsf{satrel}\ \sigma\ j\ \alpha$ as an abbreviation for $\{v.\ \mathsf{sat}\ \sigma\ v\ j\ \alpha \wedge \mathsf{length}\ v = \mathsf{nfv}\ \alpha\}$, i.e., the relation containing the valuations that satisfy $\alpha$. The function $\mathsf{nfv}\ \alpha$ returns the minimum length of $v$ needed to cover all of $\alpha$'s free variables, i.e., 0 if $\alpha$ is closed and $\mathsf{Max}\ (\mathsf{fv}\ \alpha) + 1$ otherwise. The trace $\sigma[p \Rrightarrow R]$ is the same as the trace $\sigma$ except that for every time-point $i$, the database at $i$ maps the predicate name $p$ to $R\ i$, where $R$ has type $nat \Rightarrow data\ list\ set$ and is called a *temporal relation*. Note that the subformula $\alpha$ is not necessarily evaluated at time-point $i$. Instead, the choice of the time-point is deferred until the predicate $p$ is used within $\beta$, which we achieve by updating the entire trace. This supports the intuition behind *unfolding* the let operator $\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta$ described above, especially as subformulas $p(as)$ may occur under temporal operators in $\beta$.

*Implementation.* To evaluate an MFOTL formula on a trace, VeriMon computes a finite set of satisfying valuations (represented by the type *table*) recursively for each subformula. It applies standard table operations such as the natural join ($\bowtie$) and union. Tables are sets of tuples, which are lists of optional *data* values (with missing values denoted by $\bot$) and thus refine valuations. This representation allows us to use lists of the same length for subformulas with different free variables. As with valuations, the variables' De Bruijn indices are used to look up their value in a tuple.

VeriMon processes an unbounded trace incrementally. Its interface consists of two functions $\mathsf{init} :: frm \Rightarrow state$ and $\mathsf{step} :: dbs \times ts\ list \Rightarrow state \Rightarrow (nat \times table)\ list \times state$. The function init initializes the monitor's state (type *state*), and step updates it with a batch of new time-stamped databases to produce a list of new satisfactions. Instead of *db list*, step uses the type $dbs = (string \rightharpoonup table\ list)$ (a partial mapping from *string* to *table list*) to efficiently retrieve all relations (encoded as tables) associated with a predicate name at once. Besides some auxiliary data, *state* stores an *inductive state* of type *sfrm* that mirrors the inductive representation of formulas, augmented with data structures for evaluating temporal operators and buffering intermediate results. Internally $\mathsf{step}\ (dbs, tss)\ st$ calls $\mathsf{eval}\ j\ n\ tss\ dbs\ s_\varphi$, where $j$ is the combined length of the trace prefix including the new batch, $n = \mathsf{nfv}\ \varphi$ for the monitored formula $\varphi$, and $s_\varphi$ is the inductive state, all stored in $st$. The function eval returns a list of tables with new satisfactions, as well as the updated inductive state. Satisfactions are reported for every time-point in order. They may be delayed if the formula contains future operators.

To evaluate $\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta$, we use the tables with $\alpha$'s satisfactions to evaluate $p$ within $\beta$, which requires that the tuples in these tables do not have missing values. Therefore, we require that let operators satisfy $\mathsf{mon}\ (\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta) = (\{0\ ..{<}\ \mathsf{nfv}\ \alpha\} \subseteq \mathsf{fv}\ \alpha \wedge \mathsf{mon}\ \alpha \wedge \mathsf{mon}\ \beta)$. Specifically, the (indices of) $\alpha$'s free variables must not have gaps. We add the constructor $\mathsf{SLet}\ p\ m\ s_\alpha\ s_\beta$ to the inductive state, which stores $p$, the number $m = \mathsf{nfv}\ \alpha$ of free variables in $\alpha$, and the states for subformulas $\alpha$ and $\beta$. It is initialized by initializing $s_\alpha$ and $s_\beta$ recursively. The function eval evaluates it as follows.

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

We write $dbs[p \mapsto xs]$ for the partial mapping $dbs$ updated at $p$ with $xs$. The recursive call of eval on $s_\alpha$ may return multiple tables in the list $xs$. Note that step generalizes the original VeriMon interface [3] as it consumes multiple time-stamped databases at once. The generalized interface of eval allows us to pass all tables at once to the recursive call for $s_\beta$.

*Correctness.* We relate the outputs of step and sat to prove our monitor correct. As mentioned earlier, the monitor may delay its output. We precisely characterize its *progress* for a given formula and trace prefix. Intuitively, the progress is the number of time-points that the monitor is able to evaluate given a trace prefix. Progress is a useful tool in the correctness proof as it helps us describe the output *at every time-point*. Moreover, we show below that progress can be made arbitrarily large, which is important for completeness.

Formally, $\mathsf{prog}\ \sigma\ P\ \varphi\ j$ is $\varphi$'s progress $i_\varphi$ after reading the first $j$ databases of trace $\sigma$. We added the partial mapping $P$ that assigns to every let-bound predicate its own progress, i.e., the progress of the formula defining the predicate. For example, the progress of a predicate $p$ that is not let-bound is $j$. Otherwise, it is equal to the progress of the formula it is bound to (stored in $P\ p$). The progress of $\alpha\,\mathsf{U}_{[a,b]}\,\beta$ is the smallest $i$ such that $\tau\ \sigma\ i \geq \tau\ \sigma\ (\mathsf{Min}\ \{i_\alpha, i_\beta, j-1\}) - b$. The progress of both $\alpha \wedge \beta$ and $\alpha \vee \beta$ is $\mathsf{Min}\ \{i_\alpha, i_\beta\}$.

The invariant $\mathsf{invar}\ \sigma\ j\ P\ n\ s_\varphi\ \varphi$ relates an inductive state $s_\varphi$ to the formula $\varphi$. The inductive state must reflect the monitor's state after processing the first $j$ databases in the trace $\sigma$, assuming that $P$ specifies the let-bound predicates' progress. The parameter $n$ is the length of the tuples stored within $s_\varphi$. The invariant is defined inductively over $s_\varphi$; we reuse VeriMon's definition for the MFOTL operators and add a case for Let:

$$\frac{\mathsf{invar}\ \sigma\ j\ P\ m\ s_\alpha\ \alpha \quad \mathsf{invar}\ (\sigma[p \Rrightarrow \lambda i.\ \mathsf{satrel}\ \sigma\ i\ \alpha])\ j\ (P[p \mapsto \mathsf{prog}\ \sigma\ P\ \alpha\ j])\ n\ s_\beta\ \beta \quad m = \mathsf{nfv}\ \alpha \quad \{0\ ..{<}\ m\} \subseteq \mathsf{fv}\ \alpha}{\mathsf{invar}\ \sigma\ j\ P\ n\ (\mathsf{SLet}\ p\ m\ s_\alpha\ s_\beta)\ (\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta)}$$

The first two premises restrict the subformula states $s_\alpha$ and $s_\beta$, where $s_\beta$ reflects the evaluation of $\beta$ on the modified trace, and $p$'s progress is that of $\alpha$. The premise $m = \mathsf{nfv}\ \alpha$ enforces that $m$ is equal to $p$'s arity, and $\{0\ ..{<}\ m\} \subseteq \mathsf{fv}\ \alpha$ is the constraint from mon.

Our extensions preserve the monitor's correctness: we formally proved the theorem below, which characterizes the monitor's eval function. The theorem is stated here for the empty progress mapping $\varnothing$, which must be generalized in the proof (as $P$ changes in the above rule). Let $\delta$ be a natural number and $\varphi$ be a monitorable formula with $n = \mathsf{nfv}\ \varphi$. The function the maps the optional value $\langle x \rangle$ to $x$ and $\bot$ to some unspecified value.

**Theorem 1.** *(a)* $\mathsf{invar}\ \sigma\ 0\ \varnothing\ n\ s_\varphi^0\ \varphi$ *for the initial state* $s_\varphi^0$. *(b) Suppose that* $s_\varphi$ *satisfies* $\mathsf{invar}\ \sigma\ j\ \varnothing\ n\ s_\varphi\ \varphi$ *and that dbs contains all relations from* $\sigma$ *for the indices in the list* $js = [j..<j+\delta]$. *Then* $(xs, s'_\varphi) = \mathsf{eval}\ (j+\delta)\ n\ (\mathsf{map}\ (\tau\ \sigma)\ js)\ dbs\ s_\varphi$ *satisfies* $\mathsf{invar}\ \sigma\ (j+\delta)\ \varnothing\ n\ s'_\varphi\ \varphi$, *and the i-th table in the list xs, for* $\mathsf{prog}\ \sigma\ \varnothing\ \varphi\ j \leq i < \mathsf{prog}\ \sigma\ \varnothing\ \varphi\ (j+\delta)$, *contains (only) all tuples v of length n satisfying* $\mathsf{sat}\ \sigma\ (\mathsf{map}\ \mathsf{the}\ v)\ \sigma\ i\ \varphi$.

Soundness follows immediately from Thm. 1, whereas completeness additionally requires the aforementioned property that any progress can be reached by making the trace prefix long enough, which we also proved for our modified progress function:

**Theorem 2.** *If* $\mathsf{mon}\ \varphi$, *then for all i there exists a j such that* $\mathsf{prog}\ \sigma\ \varnothing\ \varphi\ j \geq i$.

## 4 Past-Recursive Let Operator

It is well-known that first-order logic (FOL) cannot express certain queries, notably the transitive closure of a binary relation. This remains true when restricted to finite structures [18]. Although MFOTL is rather different from ordinary FOL, we conjecture that it cannot express transitive closure either. This hampers its ability to model hierarchies of unbounded depth. Moreover, recursive patterns are sometimes the most natural way to express certain specifications. We describe an extension of MFOTL that can encode a "temporally directed" form of transitive closure and other recursive patterns.

Specifically, we introduce another let operator in which the predicate may refer to itself recursively. The intended semantics is that of a fixpoint, i.e., the predicate $p$ defined by a formula $\alpha$ should be interpreted by a temporal relation that is equal to the evaluation of $\alpha$ under that interpretation of $p$. The fixpoint might not always exist or it might not be unique. Therefore, different fixpoint operators have been studied in the context of nontemporal logics and query languages [1]. For instance, it is common to require that all recursive occurrences of $p$ in its defining formula are positive, i.e., under an even number of negations. This ensures monotonicity and hence the existence of a least fixpoint.

MFOTL's future operators are interpreted over infinite traces. This poses a new challenge for monitoring recursively defined predicates, even if we restrict our attention to positive formulas. Consider the recursive definition of $p$ by $q \vee \bigcirc_{[0,\infty]} p$, where $q$ is a predicate from the trace. Although $q \vee \bigcirc_{[0,\infty]} p$ is monitorable (at most one additional time-point must be known to evaluate it), the recursive definition of $p$ is equivalent to $\Diamond_{[0,\infty]} q$ under the least fixpoint semantics. However, $\Diamond_{[0,\infty]} q$ is not monitorable, as one might need the entire, infinite trace to evaluate it. Therefore, we focus on a fragment where every recursive occurence of $p$ must be *strictly in the past*. This guarantees a unique fixpoint even if the defining formula is not monotone, so the predicate may occur negatively as well.

The syntax of our past-recursive let operator is similar to the one of Let: we add the constructor LetPast *string* $:=$ *frm* in *frm* to the *frm* datatype. However, the semantics is different (Section 4.1). The restriction to strictly past recursion is enforced by a syntactic monitorability condition that is checked by mon. Consider the formula LetPast $p := \alpha$ in $\beta$. Intuitively, every recursive occurrence of $p$ in $\alpha$ must be *guarded* by at least one strictly past operator, and there must be no future operator on the path from the occurrence to $\alpha$'s root. We *do* allow future operators in the other parts of $\alpha$, though.

We give examples of LetPast in Section 4.2. The evaluation of LetPast requires an extension of VeriMon's algorithm (Section 4.3), which we also formally prove correct.

**datatype** *recSafety* =
  U | P | NF | A

**fun** $(*)$ :: *recSafety* ⇒ *recSafety* ⇒ *recSafety*
**where**
  $\mathsf{U} * \_ = \mathsf{U}$
| $\_ * \mathsf{U} = \mathsf{U}$
| $\mathsf{A} * \_ = \mathsf{A}$
| $\_ * \mathsf{A} = \mathsf{A}$
| $\mathsf{P} * \_ = \mathsf{P}$
| $\_ * \mathsf{P} = \mathsf{P}$
| $\mathsf{NF} * \mathsf{NF} = \mathsf{NF}$

**fun** slp :: *string* ⇒ *frm* ⇒ *recSafety* **where**
  $\mathsf{slp}\ p\ (q(as)) = (\underline{\text{if}}\ p = q\ \underline{\text{then}}\ \mathsf{NF}\ \underline{\text{else}}\ \mathsf{U})$
| $\mathsf{slp}\ p\ (\mathsf{Let}\ q := \alpha\ \mathsf{in}\ \beta) =$
    $(\mathsf{slp}\ q\ \beta * \mathsf{slp}\ p\ \alpha) \sqcup (\underline{\text{if}}\ p = q\ \underline{\text{then}}\ \mathsf{U}\ \underline{\text{else}}\ \mathsf{slp}\ p\ \beta)$
| $\mathsf{slp}\ p\ (\mathsf{LetPast}\ q := \alpha\ \mathsf{in}\ \beta) =$
    $(\underline{\text{if}}\ p = q\ \underline{\text{then}}\ \mathsf{U}\ \underline{\text{else}}\ (\mathsf{slp}\ q\ \beta * \mathsf{slp}\ p\ \alpha) \sqcup \mathsf{slp}\ p\ \beta)$
| $\mathsf{slp}\ p\ (t_1 \circ t_2) = \mathsf{U}$ | $\mathsf{slp}\ p\ (y \leftarrow \Omega(t;b)\ \varphi) = \mathsf{slp}\ p\ \varphi$
| $\mathsf{slp}\ p\ (\neg\varphi) = \mathsf{slp}\ p\ \varphi$ | $\mathsf{slp}\ p\ (\exists\varphi) = \mathsf{slp}\ p\ \varphi$
| $\mathsf{slp}\ p\ (\alpha \vee \beta) = \mathsf{slp}\ p\ \alpha \sqcup \mathsf{slp}\ p\ \beta$
| $\mathsf{slp}\ p\ (\alpha \wedge \beta) = \mathsf{slp}\ p\ \alpha \sqcup \mathsf{slp}\ p\ \beta$
| $\mathsf{slp}\ p\ (\bullet_I \varphi) = \mathsf{P} * \mathsf{slp}\ p\ \varphi$ | $\mathsf{slp}\ p\ (\bigcirc_I \varphi) = \mathsf{A} * \mathsf{slp}\ p\ \varphi$
| $\mathsf{slp}\ p\ (\alpha\ \mathsf{S}_I\ \beta) = \mathsf{slp}\ p\ \alpha \sqcup ((\underline{\text{if}}\ 0 \in_{\mathcal{I}} I\ \underline{\text{then}}\ \mathsf{NF}\ \underline{\text{else}}\ \mathsf{P}) * \mathsf{slp}\ p\ \beta)$
| $\mathsf{slp}\ p\ (\alpha\ \mathsf{U}_I\ \beta) = \mathsf{A} * (\mathsf{slp}\ p\ \alpha \sqcup \mathsf{slp}\ p\ \beta)$

**Fig. 2.** Auxiliary definitions for the syntactic restriction on LetPast

### 4.1 Semantics

The semantics of the past-recursive let operator is defined by the equation

$$\mathsf{sat}\ \sigma\ v\ i\ (\mathsf{LetPast}\ p := \alpha\ \mathsf{in}\ \beta) = \mathsf{sat}\ (\sigma[p \Rrightarrow \mathsf{recp}\ (\lambda R\ j.\ \mathsf{satrel}\ (\sigma[p \Rrightarrow R])\ j\ \alpha)])\ v\ i\ \beta$$

We evaluate $\beta$ at the same time-point $i$ as the recursive let operator using an appropriately updated trace. The temporal relation assigned to $p$ is computed by the combinator recp:

**fun** recp :: $((nat \Rightarrow data\ list\ set) \Rightarrow nat \Rightarrow data\ list\ set) \Rightarrow nat \Rightarrow data\ list\ set$ **where**
  $\mathsf{recp}\ f\ v\ i = f\ (\lambda j.\ \underline{\text{if}}\ j < i\ \underline{\text{then}}\ \mathsf{recp}\ f\ j\ \underline{\text{else}}\ \{\})\ i$

The argument $f$ is a function that transforms temporal relations, and recp $f$ returns again a temporal relation. Intuitively, recp $f$ evaluates to the fixpoint $f\ (\mathsf{recp}\ f)$, except that $f\ R\ i$ can only access time-points of $R$ before $i$. For all other time-points $j \geq i$, the relation $R\ j$ is empty. The combinator recp is well-defined because $i$ is a natural number; the recursive call recp $f\ j$ affects the result only if $j < i$ and hence we can prove termination using $i$ as a variant. For the semantics of LetPast, we choose $f\ R\ i = \mathsf{satrel}\ (\sigma[p \Rrightarrow R])\ i\ \alpha$, i.e., the satisfactions of $\alpha$ with $p$ mapped to $f$'s argument $R$, to which recp supplies the result of the recursive evaluation (up to but excluding $i$).

Our definition of sat is total: it gives meaning to every formula. This includes formulas LetPast $p := \alpha$ in $\beta$ where $p$ occurs in $\alpha$ without a past guard or under a future operator. However, the semantics behaves unexpectedly in such cases. For example, LetPast $p := (q \vee \bigcirc_{[0,\infty]}\ p)$ in $p$ is equivalent to $q$. Our monitor therefore requires properly guarded formulas. Not only does this avoid confusion about the semantics, it also simplifies the implementation because the monitor need not eliminate unguarded occurrences.

Next, we describe the formalization of the syntactic restriction. The idea is to determine for every predicate whether it is used strictly in the past by analyzing the formula recursively. The datatype *recSafety* (Fig. 2) represents the possible outcomes. U(nused) means that a predicate does not occur in the formula. P(ast) means that it is evaluated at strictly earlier time-points, whereas NF (Non-Future) additionally allows the current time-point. A(ny) covers all remaining cases. The linear order $<$ on *recSafety* is induced by $\mathsf{U} < \mathsf{P} < \mathsf{NF} < \mathsf{A}$. Its reflexive closure $\leq$ corresponds to implication. For example, if the predicate $p$ is unused (U), it is clearly evaluated at earlier time-points only (P). The least upper bound $x \sqcup y$ with respect to $\leq$ corresponds to logical disjunction.

The function $\mathsf{slp}\ p\ \varphi$ (Fig. 2) analyzes the past-guardedness of a predicate $p$ in a formula $\varphi$. It uses a composition operator $y * x$ on *recSafety*. The patterns in the definition of $*$ should be matched sequentially from top to bottom; e.g., $\mathsf{A} * \mathsf{U}$ is equal to $\mathsf{U}$. Intuitively, $y * x$ describes the guardedness of a predicate that is $x$-used in some subformula, which is then $y$-used. For example, $\mathsf{slp}\ p\ (\bullet_I \varphi) = \mathsf{P} * \mathsf{slp}\ p\ \varphi$ because $\varphi$ and all occurences of $p$ therein are evaluated at time-points that are strictly in the past relative to $\bullet_I \varphi$. Note that we make a case distinction for $\alpha\, \mathsf{S}_I \beta$: if the interval $I$ excludes zero, $\beta$ is always evaluated strictly in the past. Future operators always result in $\mathsf{A}$ if $p$ is used in an operand.

Finally, we define the mon predicate for the recursive let operator:

$$\mathsf{mon}\ (\mathsf{LetPast}\ p \coloneqq \alpha\ \mathsf{in}\ \beta) = (\mathsf{slp}\ p\ \alpha \leq \mathsf{P} \wedge \{0\ ..{<}\ \mathsf{nfv}\ \alpha\} \subseteq \mathsf{fv}\ \alpha \wedge \mathsf{mon}\ \alpha \wedge \mathsf{mon}\ \beta)$$

The only difference to Let is the restriction of $p$'s occurrences in $\alpha$ via slp, which is generally an over-approximation. For example, $\mathsf{slp}\ p\ (\bullet_I \bullet_I \bigcirc_I p) = \mathsf{A}$ even though $p$ is evaluated at strictly earlier time-points. Therefore, some instances of LetPast that our algorithm could evaluate correctly are not considered to satisfy mon. We plan to replace *recSafety* with a more precise lattice in future work.

### 4.2 Examples

*Temporal Operators.* We first show that the non-metric S operator can be reduced to LetPast and $\bullet$. (We omit the interval subscripts if the interval is $[0,\infty]$.) Using the special $\mathsf{ts}(t)$ predicate, which is true iff $t$ is the current time-stamp, we can also express the metric version. This example serves to gently illustrate the semantics of LetPast. In general, formulas are more readable if they are directly expressed in terms of S, and monitoring can be more efficient. Below we give further examples in which LetPast adds expressiveness.

Let $\alpha$ and $\beta$ be two monitorable MFOTL formulas with free variables $\mathsf{fv}\ \alpha$ and $\mathsf{fv}\ \beta$, respectively. The formula $\alpha\, \mathsf{S}\, \beta$ is monitorable only if $\mathsf{fv}\ \alpha \subseteq \mathsf{fv}\ \beta$, so let us assume that, too. The following unfolding of S's semantics is well-known:

$$\mathsf{sat}\ \sigma\ v\ i\ (\alpha\, \mathsf{S}\, \beta) \iff \mathsf{sat}\ \sigma\ v\ i\ \beta \vee \big(\mathsf{sat}\ \sigma\ v\ i\ \alpha \wedge i > 0 \wedge \mathsf{sat}\ \sigma\ v\ (i-1)\ (\alpha\, \mathsf{S}\, \beta)\big) \quad (1)$$

As the unfolding recursively evaluates the formula at the previous time-point, we can directly translate it into a recursive let operator: $\varphi_{\mathsf{S}} \equiv \mathsf{LetPast}\ s(\overline{x}) \coloneqq \psi\ \mathsf{in}\ s(\overline{x})$, where $\psi \equiv \beta \vee (\alpha \wedge \bullet s(\overline{x}))$. The predicate name $s$ must be fresh, i.e., it must not occur in $\alpha$ nor $\beta$. The variable list $\overline{x}$ enumerates $\mathsf{fv}\ \beta$. The formula $\varphi_{\mathsf{S}}$ is monitorable because $\bullet s(\overline{x})$ is clearly past-guarded, and hence $\mathsf{slp}\ s\ \psi = \mathsf{P}$. (We also need $\mathsf{fv}\ \beta = \{0\ ..{<}\ \mathsf{nfv}\ \beta\}$, which can be achieved by renaming variables in $\alpha$ and $\beta$.) Let us analyze the semantics of $\varphi_{\mathsf{S}}$:

$$\begin{aligned}
\mathsf{sat}\ \sigma\ v\ i\ \varphi_{\mathsf{S}} &\iff \mathsf{sat}\ (\sigma[s \Rrightarrow \mathsf{recp}\ \underbrace{(\lambda R\ j.\ \mathsf{satrel}\ (\sigma[s \Rrightarrow R])\ j\ \psi)}_{=f_\psi}])\ v\ i\ (s(\overline{x})) \\
&\iff v \in \mathsf{recp}\ f_\psi\ i \\
&\iff \mathsf{sat}\ (\sigma[s \Rrightarrow \lambda j.\ \underline{\mathsf{if}}\ j < i\ \underline{\mathsf{then}}\ \mathsf{recp}\ f_\psi\ j\ \underline{\mathsf{else}}\ \{\}])\ v\ i\ \psi \\
&\overset{(*)}{\iff} \mathsf{sat}\ \sigma\ v\ i\ \beta \vee \big(\mathsf{sat}\ \sigma\ v\ i\ \alpha \wedge i > 0 \\
&\qquad \wedge v \in (\underline{\mathsf{if}}\ i-1 < i\ \underline{\mathsf{then}}\ \mathsf{recp}\ f_\psi\ (i-1)\ \underline{\mathsf{else}}\ \{\})\big) \\
&\iff \mathsf{sat}\ \sigma\ v\ i\ \beta \vee \big(\mathsf{sat}\ \sigma\ v\ i\ \alpha \wedge i > 0 \wedge \mathsf{sat}\ \sigma\ v\ (i-1)\ \varphi_{\mathsf{S}}\big)
\end{aligned}$$

These equations hold for all valuations $v$ of length $\mathsf{nfv}\ \beta$ and if the variables $\overline{x}$ are ordered by their De Bruijn indices. Step $(*)$ exploits the freshness of $s$ with respect to $\alpha$ and $\beta$,

which allows us to replace $\sigma[s \Rrightarrow \ldots]$ by $\sigma$. The equations result in the same unfolding as (1). Hence, we can prove the semantic equivalence of $\varphi_{\mathsf{S}}$ and $\alpha\,\mathsf{S}\beta$ by induction on $i$.

The following *SinceLet* formula encodes $\alpha\,\mathsf{S}_{[a,b]}\beta$. Other encodings exist, however.

$$\mathsf{LetPast}\ s(\overline{x},t) \coloneqq (\beta \wedge \mathsf{ts}(t)) \vee (\alpha \wedge \bullet\, s(\overline{x},t))\ \mathsf{in}\ \exists t,u.\ s(\overline{x},t) \wedge \mathsf{ts}(u) \wedge a \leq u-t \wedge u-t \leq b$$

Here, $t$ and $u$ are fresh variables, where $t$ records the time-stamp of the past satisfaction of $\beta$, whereas $u$ is the time-stamp at which we evaluate *SinceLet*. The subformula $a \leq u-t \wedge u-t \leq b$ corresponds to $\mathsf{T}\ \sigma\ j - \mathsf{T}\ \sigma\ i \in_{\mathcal{I}} [a,b]$, which is part of $\mathsf{S}_{[a,b]}$'s semantics (Fig. 1).

*Temporally-Directed Transitive Closure.* We proceed by showing that LetPast can compute a temporally-directed transitive closure over events observed at a sequence of distinct time-points. Hence, we assume that the trace contains a single event at every time-point. The closure is directed in the sense that the transitive chains can only be extended by *newer* events. We consider the following two types of events from [14]: $r(y,x,d)$ denotes that process $y$ reports some data $d$ to another process $x$, and $s(x,y)$ denotes that process $x$ spawns process $y$. The *Spawn* formula

$$\mathsf{LetPast}\ p(u,v) \coloneqq s(u,v) \vee (\bullet\, p(u,v)) \vee (\exists t.\ (\bullet\, p(u,t)) \wedge s(t,v))\ \mathsf{in}\ r(y,x,d) \wedge \neg p(x,y)$$

encodes violations of the property that whenever process $y$ sends some data $d$ to a process $x$, denoted as $r(y,x,d)$, then there was a chain of process spawns: $s(x,x_1), s(x_1,x_2), \ldots, s(x_k,y)$, occurring in this order in the trace. In other words, a process may only send data to its "ancestors". To check this property, a monitor needs to compute the (temporally-directed) transitive closure $p(u,v)$ of the relation $s$. The definition of the closure has two recursive predicate instances with different arguments. The *Spawn* formula is inspired by a similar one used to evaluate the DejaVu monitor [14]. Unlike DejaVu, we do not require the formula to be closed and thus leave the variables $x$, $y$, and $d$ free.

The *Trans* formula

$$\begin{aligned}\mathsf{LetPast}\ p(u,v) \coloneqq\ & s(u,v) \vee (\bullet\, p(u,v)) \vee \\ & (\exists t.\ (\bullet\, p(u,t)) \wedge s(t,v)) \vee (\exists t.\ s(u,t) \wedge (\bullet\, p(t,v))) \vee \\ & (\exists t,t'.\ (\bullet\, p(u,t)) \wedge s(t,t') \wedge (\bullet\, p(t',v)))\quad \mathsf{in}\quad r(y,x,d) \wedge \neg p(x,y)\end{aligned}$$

encodes violations of the same property as *Spawn* even if $s(x,x_1), s(x_1,x_2), \ldots, s(x_k,y)$ are received by the monitor out-of-order, i.e., they do not occur in this order in the trace.

We can interpret the events $s(x,y)$ as edges in a directed graph and the predicate $p(x,y)$ in *Trans* as computing the reachability of vertices in the directed graph. We also extend the directed edges $s(x,y)$ with a weight $w$ to $s^+(x,y,w)$. Then the *Trans*$^+$ formula

$$\begin{aligned}\mathsf{LetPast}\ p(u,v,w) \coloneqq\ & s^+(u,v,w) \vee (\bullet\, p(u,v,w)) \vee \\ & (\exists t,w_1,w_2.\ (\bullet\, p(u,t,w_1)) \wedge s^+(t,v,w_2) \wedge w = w_1 + w_2) \vee \\ & (\exists t,w_1,w_2.\ s^+(u,t,w_1) \wedge (\bullet\, p(t,v,w_2)) \wedge w = w_1 + w_2) \vee \\ & (\exists t,t',w_1,w_2,w_3.\ (\bullet\, p(u,t,w_1)) \wedge s^+(t,t',w_2) \wedge (\bullet\, p(t',v,w_3)) \wedge \\ & \qquad w = w_1 + w_2 + w_3)\ \mathsf{in} \\ \mathsf{Let}\ m(u,v,w) \quad \coloneqq\ & w \leftarrow \mathsf{MIN}(w;u,v).\ p(u,v,w)\quad \mathsf{in}\quad m(x,y,w) \wedge \neg(\bullet\, m(x,y,w))\end{aligned}$$

yields all pairs of vertices $x$, $y$ and the length $w$ of the shortest path from $x$ to $y$ whenever $y$ becomes reachable from $x$ or the length of the shortest path changes. The relation

$s^+(x,y,w)$ can itself be obtained by evaluating a more complex temporal formula, e.g., $s^+(x,y,w) \equiv e(x,y,w) \wedge \neg \Diamond_{[0,10]}\, d(x,y)$ with the following two types of events: $e(x,y,w)$ denotes an edge from $x$ to $y$ with weight $w$; $d(x,y)$ denotes deletion of the edge from $x$ to $y$. The *eventually* operator $\Diamond_I \varphi$ abbreviates $(\exists x.\ x = x)\, \mathsf{U}_I \varphi$. Such a relation $s^+(x,y,w)$ contains all edges that are not revoked within 10 time units after receiving $e(x,y,w)$. We could use the non-recursive let operator $\mathsf{Let}\ s^+(x,y,w) \coloneqq e(x,y,w) \wedge \neg \Diamond_{[0,10]}\, d(x,y)$ to precompute the relation and use it when evaluating the recursive let operator in $\mathit{Trans}^+$.

As another application of future operators under LetPast, recall our introductory example. Suppose that hosts in a network communicate with each other and with the outside world: $\mathit{comm}(\mathit{src},\mathit{dest})$ indicates that host *src* sends a message to host *dest*; $\mathit{in}(r,h)$ and $\mathit{out}(h,r)$ indicate that the host $h$ receives or sends traffic from or to an IP address in the range $r$, respectively. The hosts are equipped with an intrusion detection system (IDS), whose alerts are denoted by $\mathit{ids}(h)$. We say that a host $h$ is *tainted* by an address range $r$ iff there is a chain of communication from $r$ to $h$ and all hosts on the chain (including $h$) trigger an IDS alert within one hour after communicating with the previous host. The formula

$$\begin{aligned}\mathsf{LetPast}\ \mathit{taint}(r,h) \coloneqq\ & \big(\big(\mathit{in}(r,h) \vee \exists h'.\ (\bullet\, \mathit{taint}(r,h')) \wedge \mathit{comm}(h',h)\big) \wedge \Diamond_{[0,1\mathrm{h}]}\, \mathit{ids}(h)\big) \vee \\ & (\bullet\, \mathit{taint}(r,h))\quad \mathsf{in}\quad \mathit{taint}(r,h) \wedge \mathit{out}(h,r)\end{aligned}$$

is true whenever a host communicates back to the IP range by which it was tainted.

*Periodic Behavior.* Suppose that we monitor a boolean signal $b(x)$, parametrized by an integer parameter $x$, between the user's $\mathit{start}(x)$ and $\mathit{stop}(x)$ commands. An arbitrary amount of time may pass between these two commands. Our task is to detect periodic activations of $b(x)$, with a fixed period $t > 0$ and error tolerance $0 \le \varepsilon < t$. We shall ignore positive noise in $b(x)$, i.e., additional activations besides the periodic ones.

Let us make the task more precise. An alarm must be raised at time-point $i_n$ iff there exist time-points $i_0 < i_1 < \cdots < i_n$ such that $\mathit{start}(x)$ holds at $i_0$, $\mathit{stop}(x)$ holds at $i_n$, and $b(x)$ holds at all $i_k$ for $1 \le k \le n-1$. Moreover, the difference of time-stamps for adjacent time-points $i_k$ and $i_{k+1}$, where $1 \le k \le n-2$, must be in the interval $[t-\varepsilon, t+\varepsilon]$; the differences for the pairs $i_0$, $i_1$ and $i_{n-1}$, $i_n$ must each be at most $t+\varepsilon$.

Our first attempt *PB* to formalize the alarm condition without recursion is

$$\mathit{stop}(x) \wedge \big(\blacklozenge_I(\mathit{start}(x) \vee b(x))\big) \wedge \big((b(x) \longrightarrow (\blacklozenge_I\, \mathit{start}(x)) \vee (\blacklozenge_J\, b(x)))\, \mathsf{S}\, \mathit{start}(x)\big)$$

where $I = [0, t+\varepsilon]$, $J = [t-\varepsilon, t+\varepsilon]$, and $\blacklozenge_K \varphi$ abbreviates $(\exists x.\ x = x)\, \mathsf{S}_K \varphi$. This formula follows an inductive approach: every $b(x)$ between $\mathit{start}(x)$ and $\mathit{stop}(x)$ must be preceded by $b(x)$ or $\mathit{start}(x)$, with the appropriate time difference. However, *PB* does not ignore noise, as adding $b(x)$ events to the trace may silence an alarm. For example, let $t = 10$, $\varepsilon = 0$, and $\sigma$ be a trace starting with $(\{\mathit{start}(1)\}, 0)$, $(\{b(1)\}, 10)$, $(\{\mathit{stop}(1)\}, 20)$. We write $\{p(1), p(2)\}$ for the database where the predicate $p$ holds for 1 and 2. On $\sigma$, *PB* is true at the third time-point. Inserting a database $\{b(1)\}$ with time-stamp 15 falsifies *PB* at the now fourth time-point, although the trace still satisfies the natural language description.

The following *PBLet* formula expresses the intended condition using LetPast:

$$\begin{aligned}&\mathsf{LetPast}\ \mathit{periodic}(x) \coloneqq \mathit{start}(x) \vee \big(b(x) \wedge \big((\blacklozenge_I\, \mathit{start}(x)) \vee (\blacklozenge_J\, \mathit{periodic}(x))\big)\big)\ \mathsf{in} \\ &\mathit{stop}(x) \wedge \blacklozenge_I\, \mathit{periodic}(x)\end{aligned}$$

This example depends crucially on the flexible past guards we support: here, the recursion goes through $\blacklozenge$ with an interval constraint. Note that $0 \notin J$ because we assumed $\varepsilon < t$.

As another example of periodic behavior, we analyze an integer-valued $\mathit{signal}(y)$ between the (now non-parametric) commands *start* and *stop*. We aim to discover whether $\mathit{signal}(y)$ is piecewise constant, with the constant segments being exactly $t$ time units long. Moreover, the signal's values for subsequent segments must differ by at most $\delta$. The next formula uses the general $\mathsf{S}$ operator as the recursion guard to capture this property.

$$\begin{aligned}&\mathsf{LetPast}\ \mathit{segment}(y) \coloneqq \exists z.\ \mathit{signal}(y) \wedge \Big(\big((\bullet\, \mathit{signal}(z))\ \mathsf{S}_{[0,t]}\ (\mathit{signal}(z) \wedge \bullet\, \mathit{start})\big) \vee \\ &\qquad \big((\bullet\, \mathit{signal}(z))\ \mathsf{S}_{[t,t]}\ \mathit{segment}(z)\big)\Big) \wedge -\delta \le y - z \wedge y - z \le \delta\ \mathsf{in} \\ &\mathit{stop} \wedge \exists y.\ ((\bullet\, \mathit{signal}(y))\ \mathsf{S}_{[0,t]}\ \mathit{segment}(y))\end{aligned}$$

*Turing Machines.* Every MFOTL formula can be viewed as a function on traces, where the function's output is the set of satisfying valuations, either at a fixed or at all time-points. VeriMon's monitorable fragment guarantees that one can compute the valuation at every time-point. Thus, monitorable formulas correspond to computable functions. If we give up on the requirement that the function's output must be available at a fixed time-point, the past-recursive let operator is expressive enough to simulate arbitrary Turing machines (TM). This is not a contradiction: we simulate a single TM step at every time-point, and there is an infinite supply of time-points. Running the monitor on a configuration that does not halt will never produce an output, i.e., a nonempty set of satisfying valuations.

Let $M = \langle \Sigma, b, Q, q_0, q_f, \delta \rangle$ be a deterministic TM with tape alphabet $\Sigma$, blank symbol $b \in \Sigma$, control states $Q$, initial state $q_0 \in Q$, final state $q_f \in Q$, and transition function $\delta \in (Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\})$. Whenever the machine is in state $q_1$ and reads the symbol $s_1$, it enters state $q_2$, writes the symbol $s_2$, and moves the head by $m$ tape cells to the right, where $\delta(q_1, s_1) = \langle q_2, s_2, m \rangle$. Without loss of generality, we assume that $\Sigma$ and $Q$ are finite subsets of the integers. We simulate $M$ using the formula $\varphi_M$ shown below.

$$\begin{aligned}&\mathsf{LetPast}\ \mathit{cfg}(q,i,s) \coloneqq \\ &\quad \mathsf{Let}\ \mathit{cfg}(q,i,s) \coloneqq \bullet\, \mathit{cfg}(q,i,s)\ \mathsf{in} \\ &\quad \mathsf{Let}\ \mathit{head}(q,s) \coloneqq \mathit{cfg}(q,0,s) \vee \big(\neg(\exists x, z.\ \mathit{cfg}(x,0,z)) \wedge (\exists y, z.\ \mathit{cfg}(q,y,z)) \wedge s = b\big)\ \mathsf{in} \\ &\quad \big(\mathit{input}(i,s) \wedge q = q_0\big) \vee \\ &\qquad \bigvee_{\substack{q_1, s_1 \\ \delta(q_1,s_2) = \langle q_2, s_2, m \rangle}} \Big(\mathit{head}(q_1,s_1) \wedge q = q_2 \wedge \big((i = -m \wedge s = s_2) \vee \\ &\qquad\qquad (\exists j.\ \mathit{cfg}(q_1,j,s) \wedge j \neq 0 \wedge i = j - m)\big)\Big)\ \mathsf{in}\ \mathit{cfg}(q_f,i,s)\end{aligned}$$

The idea is that *cfg* represents the current configuration of the TM. Specifically, $\mathit{cfg}(q,i,s)$ holds if the machine is in control state $q$ and the tape contains the symbol $s$ in the $i$th cell to the right of the head ($i$ may be negative). Note that we use nested, non-recursive let operators to abbreviate repeated subformulas. In the body of $\mathsf{Let}\ \mathit{cfg}(q,i,s) \coloneqq \bullet\, \mathit{cfg}(q,i,s)\ \mathsf{in}\ \ldots$, the predicate *cfg* refers to the previous configuration. The predicate *head* provides the current state and the symbol under the head. Its definition extends the tape by a blank symbol if necessary. The simulation is started at time-point 0 by providing the tape's initial content in the predicate *input*, which must include the cell $\mathit{input}(0, s_0)$ with the symbol $s_0$ under the head's initial position. If and only if $M$ halts on this input, there exists a time-point $i$ at which $\varphi_M$ is satisfied by at least one valuation $(i, s)$. Moreover, the satisfying valuations at $i$ represent the final state of the tape.

### 4.3 Algorithm

The restriction to past-guarded recursion allows for an efficient evaluation algorithm for LetPast formulas. It is efficient because no fixpoint iteration is required at individual time-points. To evaluate LetPast $p \coloneqq \alpha$ in $\beta$, we first try to evaluate $\alpha$ for as many time-points as possible and then use the results to interpret $p$ in $\beta$. This part is the same as for the non-recursive Let, but the evaluation of $\alpha$ itself differs. The syntactic monitorability condition guarantees that $\alpha$ at time-point $i$ depends on the predicate $p$ only for time-points strictly less than $i$. Specifically, we have defined mon (LetPast $p \coloneqq \alpha$ in $\beta$) such that the progress of $\alpha$'s evaluation does not depend on $p$'s progress beyond time-point $i-1$. Therefore, we can evaluate $\alpha$ at time-point 0 without providing any table for $p$, then use the result to evaluate $\alpha$ at time-point 1, and so forth.

There are two cases that require care. First, if $\alpha$ contains future operators, multiple time-points may be evaluated at once. The above process must then be repeated within a single monitor step. Second, if $\alpha$ contains no future operators, $\alpha$ is evaluated at all time-points $i < j$, where $j$ is the current trace prefix length. We could then attempt to evaluate $\alpha$ once more at time-point $j$ using the table computed at $j-1$ for $p$. However, this would not yield any further tables because all occurrences of $p$ are below at least one past operator that tries to access the time-stamp at time-point $j$, which is not yet known. Therefore, this last evaluation attempt would needlessly traverse the formula state. We optimize this case and buffer $\alpha$'s result at time-point $j-1$ until the next input database arrives.

It is crucial that the evaluation of a recursive let does not get stuck waiting for tables that it needs to produce itself. Therefore, all operators that are strictly past-guarding as defined by slp (Fig. 2) must be well-behaved: the evaluation algorithm must compute a result at time-point $i < j$ even if the operands' results are available only for time-points $i' < i$. In particular, $\mathsf{S}_I$ without 0 in the interval is considered strictly past-guarding. We have modified VeriMon's evaluation algorithm for $\alpha\, \mathsf{S}_I\, \beta$ to achieve this behavior.

The inductive state SLetPast $p\ m\ s_\alpha\ s_\beta\ i\ buf$ for a recursive let operator extends SLet with a counter $i :: nat$, which tracks the progress of $p$ as observed by $s_\alpha$, and an optional buffer $buf :: table\ option$. The meaning of the other arguments is the same as for SLet. In the initial state, $i$ is zero and $buf$ is $\bot$. Let the function list_opt map $\bot$ to $[]$ and $\langle x \rangle$ to $[x]$, where $\langle x \rangle$ is the embedding of $x$ into the *option* type. A single monitor step updates the state as follows (see Section 3 for a description of eval's interface):

$$\begin{aligned}
&\mathsf{eval}\ j\ n\ tss\ dbs\ (\mathsf{SLetPast}\ p\ m\ s_\alpha\ s_\beta\ i\ buf) = \\
&\quad \big(\underline{\mathsf{let}}\ (xs, s'_\alpha, i', buf') = \mathsf{eval}_{\mathsf{LP}}\ j\ m\ tss\ dbs\ p\ []\ s_\alpha\ i\ (\mathsf{list_opt}\ buf); \\
&\qquad (ys, s'_\beta) = \mathsf{eval}\ j\ n\ tss\ (dbs[p \mapsto xs])\ s_\beta \\
&\quad \underline{\mathsf{in}}\ (ys, \mathsf{SLetPast}\ p\ m\ s'_\alpha\ s'_\beta\ i'\ buf')\big)
\end{aligned}$$

The heavy lifting is performed by $\mathsf{eval}_{\mathsf{LP}}$, which is mutually recursive with eval. We forward relevant variables from eval. The accumulator $xs :: table\ list$ collects $s_\alpha$'s results.

$$\begin{aligned}
&\mathsf{eval}_{\mathsf{LP}}\ j\ m\ tss\ dbs\ p\ xs\ s_\alpha\ i\ buf = \\
&\quad \big(\underline{\mathsf{let}}\ (xs', s'_\alpha) = \mathsf{eval}\ j\ m\ tss\ (dbs[p \mapsto buf])\ s_\alpha;\ \ i' = i + \mathsf{length}\ buf \\
&\quad \underline{\mathsf{in}}\ \big(\underline{\mathsf{case}}\ xs'\ \underline{\mathsf{of}}\ [] \Rightarrow (xs, s'_\alpha, i', \bot) \\
&\qquad\qquad \mid x \# _ \Rightarrow (\underline{\mathsf{if}}\ i' + 1 \geq j\ \underline{\mathsf{then}}\ (xs\ @\ xs', s'_\alpha, i', \langle x \rangle) \\
&\qquad\qquad\qquad\qquad \underline{\mathsf{else}}\ \mathsf{eval}_{\mathsf{LP}}\ m\ j\ []\ (\mathsf{clear_dbs}\ dbs)\ p\ (xs\ @\ xs')\ s'_\alpha\ i'\ xs')\big)\big)
\end{aligned}$$

First, $\mathsf{eval}_{\mathsf{LP}}$ evaluates $s_\alpha$ with *dbs* updated at $p$ using the current buffer, which may be empty. Since $i$ tracks $p$'s progress, we then increase its new value $i'$ by the length of *buf*. The evaluation results in a list $xs'$ of tables and a new state $s'_\alpha$. We continue to iterate $\mathsf{eval}_{\mathsf{LP}}$ only if two conditions are met: $xs'$ must be nonempty, as otherwise there is no new data to evaluate $s'_\alpha$ on, and $i'+1$ must be less than the current input prefix length. The latter condition serves as an obvious termination criterion, although it is stricter than necessary. We could perform an additional iteration in the case that $i'+1=j$. However, such an iteration would never produce new results because the past operators guarding $p$ can only be evaluated further if there are new time-stamps. Therefore, we optimize this case by choosing the stricter condition. If we continue the iteration, we append $xs'$ to the accumulator *xs*. Moreover, we clear *tss* and *dbs* because all tables from the new input database have already been processed by the first call to eval. Specifically, the function clear_dbs *dbs* updates *dbs* at all points at which it is defined to an empty list.

We illustrate our algorithm with an example, tracing the computations of eval and $\mathsf{eval}_{\mathsf{LP}}$. We evaluate $\mathsf{LetPast}\, p(x) \coloneqq q(x) \vee \bullet p(x)\ \mathsf{in}\ p(x)$, which has the same semantics as $\blacklozenge_{[0,\infty]} q(x)$, on a prefix with two time-points at time-stamps 0 and 3. We omit details about the subformulas' states, as well as brackets around singleton lists, i.e., $[1]$ is displayed as 1. Let $dbs_0 = \{q \mapsto [\{1\},\{2\}]\}$ be the content of the trace prefix.

$$
\begin{aligned}
&\mathsf{eval}\ j{:}2\ n{:}1\ tss{:}[0,3]\ dbs{:}dbs_0\ s_\varphi{:}(\mathsf{SLetPast}\ p\ 1\ \alpha_0\ \beta_0\ 0\ \bot)\\
&\mid \mathsf{eval}_{\mathsf{LP}}\ j{:}2\ m{:}1\ tss{:}[0,3]\ dbs{:}dbs_0\ p{:}p\ xs{:}[]\ s_\alpha{:}\alpha_0\ i{:}0\ buf{:}[]\\
&\mid\ \mid \mathsf{eval}\ j{:}2\ n{:}1\ tss{:}[0,3]\ dbs{:}(dbs_0[p \mapsto []])\ s_\varphi{:}\alpha_0 && = ([\{1\}],\alpha_1)\\
&\mid\ \mid \mathsf{eval}_{\mathsf{LP}}\ j{:}2\ m{:}1\ tss{:}[]\ dbs{:}\{q \mapsto []\}\ p{:}p\ xs{:}[\{1\}]\ s_\alpha{:}\alpha_1\ i{:}0\ buf{:}[\{1\}]\\
&\mid\ \mid\ \mid \mathsf{eval}\ j{:}2\ n{:}1\ tss{:}[]\ dbs{:}\{p \mapsto [\{1\}], q \mapsto []\}\ s_\varphi{:}\alpha_1 && = ([\{1,2\}],\alpha_2)\\
&\mid\ \mid\ \mid \text{iteration stops because } i'=1 \text{ and hence } i'+1=2 \geq j=2\\
&\mid\ \mid = ([\{1\},\{1,2\}],\alpha_2,1,\langle\{1,2\}\rangle)\\
&\mid = ([\{1\},\{1,2\}],\alpha_2,1,\langle\{1,2\}\rangle)\\
&\mid \mathsf{eval}\ j{:}2\ n{:}1\ tss{:}[0,3]\ dbs{:}(dbs_0[p \mapsto [\{1\},\{1,2\}]])\ s_\varphi{:}\beta_0 && = ([\{1\},\{1,2\}],\beta_2)\\
&= ([\{1\},\{1,2\}],\mathsf{SLetPast}\ p\ 1\ \alpha_2\ \beta_2\ 1\ \langle\{1,2\}\rangle)
\end{aligned}
$$

*Correctness.* We extended the correctness proof of eval (Thm. 1) to cover the new state constructor SLetPast. The added case differs from the one for the non-recursive let in that $\mathsf{eval}_{\mathsf{LP}}$ is used to evaluate the first subformula. The proof also required additional invariants for the $i$ and *buf* arguments of SLetPast, as well as a characterization of LetPast's progress. Recall that progress describes the number of time-points that the monitor is able to evaluate given a trace prefix of length $j$. We express the progress of the let-bound predicate $p$, which is defined in terms of $\alpha$, as a least fixpoint:

$$
\begin{aligned}
&\mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j = \bigsqcap \{i.\ i = \mathsf{prog}\ \sigma\ (P[p \mapsto i])\ \alpha\ j\}\\
&\mathsf{prog}\ \sigma\ P\ (\mathsf{LetPast}\ p \coloneqq \alpha\ \mathsf{in}\ \beta)\ j = \mathsf{prog}\ \sigma\ (P[p \mapsto \mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j])\ \beta\ j
\end{aligned}
$$

(We do not update $\sigma$ in these definitions as progress depends only on the time-stamp sequence but not on the databases in $\sigma$.) The above characterization follows the iteration in $\mathsf{eval}_{\mathsf{LP}}$: Since prog is pointwise monotone in $P$ and at most $j$ (both facts we prove in the formalization), the fixpoint can be reached by iteratively computing $\mathsf{prog}\ \sigma\ (P[p \mapsto i])\ \alpha\ j$ starting with $i=0$. Similarly, $\mathsf{eval}_{\mathsf{LP}}$ starts by evaluating $\alpha$ with no data for $p$ and it feeds the results back into the evaluation until no further results can be obtained. Theorem 2 remains true after adding the above equation to prog.

The state invariant for SLetPast is given by the rule

$$\frac{\begin{array}{c}\mathsf{invar}\ \big(\sigma[p \Rrightarrow \mathsf{recp}\ (\lambda R\ k.\ \mathsf{satrel}\ (\sigma[p \Rrightarrow R])\ k\ \alpha)]\big)\ j\ (P[p \mapsto i])\ m\ s_\alpha\ \alpha \\ \mathsf{invar}\ \big(\sigma[p \Rrightarrow \mathsf{recp}\ (\lambda R\ k.\ \mathsf{satrel}\ (\sigma[p \Rrightarrow R])\ k\ \alpha)]\big)\ j\ (P[p \mapsto \mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j])\ n\ s_\beta\ \beta \\ \mathit{buf} = \bot \longrightarrow i = \mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j \\ \big(\forall Z.\ \mathit{buf} = \langle Z \rangle \longrightarrow i + 1 = \mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j \\ \qquad \wedge\, \mathsf{table}\ m\ (\mathsf{fv}\ \alpha)\ (\mathsf{recp}\ (\lambda R\ k.\ \mathsf{satrel}\ (\sigma[p \Rrightarrow R])\ k\ \alpha))\ Z\big) \\ m = \mathsf{nfv}\ \alpha \quad \mathsf{slp}\ p\ \alpha \leq \mathsf{P} \quad \{0\ ..{<}\ m\} \subseteq \mathsf{fv}\ \alpha\end{array}}{\mathsf{invar}\ \sigma\ j\ P\ n\ (\mathsf{SLetPast}\ p\ m\ s_\alpha\ s_\beta\ i\ \mathit{buf})\ (\mathsf{LetPast}\ p := \alpha\ \mathsf{in}\ \beta)}$$

The first two premises use the same updated trace as in the semantics of LetPast (Section 4.1). The updated progress for $p$ differs slightly between the premise for $s_\alpha$ and that for $s_\beta$. For the latter it is given by $\mathsf{prog}_{\mathsf{LP}}$, as expected. The predicate $p$'s progress within $s_\alpha$ is equal to the state variable $i$, which is one less than $\mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j$ if the buffer *buf* is nonempty. This reflects to the optimization discussed in Section 4.3. The predicate $\mathsf{table}\ A\ n\ R\ Z$ is true iff the table $Z$ contains tuples of length $n$ that assign values to variables $A$ and they are exactly the tuples of this kind satisfying $\mathsf{map}\ \mathsf{the}\ v \in R$.

## 5 Evaluation

We have used Isabelle/HOL's code generator [12] to export a certified implementation of VeriMon's core init and step functions and every function those depend on (e.g., operations on red-black trees), which amounts to about 10 000 lines of OCaml code. VeriMon augments this generated code with unverified parsers and pretty-printers. We evaluate this implementation to answer the following research questions: (1) How does VeriMon perform when monitoring formulas with the recursive let operator?; and (2) How does it compare to existing monitors for temporal first-order specifications with recursive rules?

To answer these questions, we run VeriMon and DejaVu and benchmark some of the example formulas introduced in Section 4.2. Instead of *SinceLet*, we opt for the simpler *OnceLet* = LetPast $o(u,v) := s(u,v) \vee \bullet\, o(u,v)$ in $\mathit{filter}(x,y) \wedge o(x,y)$ encoding the non-metric $\blacklozenge$ operator. We also include *Once* = $\mathit{filter}(x,y) \wedge \blacklozenge\, s(x,y)$ for comparison. The predicate $\mathit{filter}(x,y)$ keeps the output size small. The *OnceLet* formula uses only one recursive predicate instance, whose variable order matches the one in the predicate's definition. Other formulas have more than one instance with different variable orders.

For the *PBLet* formula, we use an existing random trace generator [17] configured to pick parameters from a small integer domain, which increases the probability of producing satisfactions. For the other formulas, we generate traces using a similar strategy to the one used in DejaVu's benchmarks on the *Spawn* formula [14]. Namely, edges of a tree of spawned processes with a configurable branching factor are linearized into a trace, level by level. In the final level all edges converge to a single node for the formulas *Trans* and $\mathit{Trans}^+$. We define the edges by $\mathsf{Let}\ s^+(x,y,w) := e(x,y,w) \wedge \neg \Diamond_{[0,10]}\, d(x,y)$ in the $\mathit{Trans}^+$ formula and revoke one half of the edges on the second level of the branching.

We have executed our experiments on an Intel Core i5-4200U CPU using 8 GB RAM. Initially, DejaVu crashed on the *OnceLet* and *Spawn* formulas. We investigated the issue and found that its formula's abstract syntax tree was disconnected in these cases. We assume that this is caused by naming variables in the recursive rules' definitions

| Trace length | *Once* | | *OnceLet* | | *Spawn* | | *Trans* | | *Trans*$^+$ | *PBLet* |
|---|---|---|---|---|---|---|---|---|---|---|
| | VeriMon | DejaVu | VeriMon | DejaVu | VeriMon | DejaVu | VeriMon | DejaVu | VeriMon | VeriMon |
| 100 | 0.0 | 1.1 | 0.0 | 1.1 | 0.6 | 1.5 | 1.3 | 3.7 | 5.6 | 0.0 |
| 200 | 0.0 | 1.2 | 0.0 | 1.2 | 3.1 | 2.1 | 6.1 | 8.1 | 25.9 | 0.0 |
| 400 | 0.0 | 1.3 | 0.0 | 1.3 | 14.0 | 3.4 | 28.3 | 23.6 | 117.4 | 0.0 |
| 800 | 0.0 | 1.5 | 0.0 | 1.4 | 64.8 | 8.2 | TO | 83.4 | TO | 0.0 |
| 4000 | 0.2 | 41.3 | 0.1 | 40.5 | TO | TO | TO | TO | TO | 0.1 |
| 8000 | 0.4 | TO | 0.2 | TO | TO | TO | TO | TO | TO | 0.1 |
| 10000 | 0.5 | TO | 0.3 | TO | TO | TO | TO | TO | TO | 0.2 |

**Fig. 3.** Execution times of the monitors in seconds (TO = timeout of 120 seconds)

differently from those in the rules' usages. After renaming the variables in the let-bound predicates of these two formulas, the issue was fixed and we restarted the experiments.

The evaluation results (Figure 3) show that DejaVu's performance is incomparable to VeriMon's. VeriMon outperforms DejaVu on the formulas *Once* and *OnceLet* and scales well on *PBLet*, which, together with the *Trans*$^+$ formula, we could not express in PFLTL with recursion. DejaVu outperforms VeriMon on the *Spawn* and *Trans* formulas for which VeriMon's time complexity of processing one event is linear in the trace length because the number $N$ of valuations satisfying the recursive predicates grows linearly in the trace length and the time complexity of updating the recursive predicate is linear in $N$. We conjecture based on some preliminary experiments that VeriMon's performance can be significantly improved by optimizing the representation of sets of tuples in two ways: (a) using tuples of a fixed length with a fixed assignment of variables to positions in a tuple (i.e., no De Bruijn indices); (b) using a collection of indices to optimize the computation of joins on various sets of shared columns. Nevertheless, processing one event can unlikely be made trace-length independent: *Trans* encodes the *incremental dynamic transitive closure* graph problem, with the best known algorithm processing every new edge in the input in amortized linear time (in the graph's maximum out-degree) [23].

## 6 Conclusion

We have presented the extension of a monitor for MFOTL with non-recursive and past-recursive let operators. The presence of bounded future temporal operators complicates both the semantics and the evaluation algorithms for the new constructs, compared to earlier unverified extensions of past-only monitors [14]. Yet, the formal correctness proofs that we have carried out ensure the trustworthiness of our development.

As future work we plan to improve the performance of evaluating expensive joins by introducing indices, as used in database management systems. Expressiveness-wise we will consider further relaxing the requirements on the recursive let. We can omit the past guard if we define a Datalog-style fragment for which the fixpoint is well-defined. Beyond relaxing guards, we may want to allow recursion through future operators in certain situations. The main challenge is that this would make the progress notion data-dependent (unlike currently, where it only depends on the time-stamps).

*Acknowledgments* We thank David Basin for supporting this work and the anonymous TACAS reviewers for their helpful comments. Dmitriy Traytel is supported by a Novo Nordisk Fonden Start Package Grant (NNF20OC0063462).

## References


1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer (2004). https://doi.org/10.1007/978-3-540-24622-0_5
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., 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
5. 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
6. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: Temporal stream-based specification language. In: Massoni, T., Mousavi, M.R. (eds.) SBMF 2018. LNCS, vol. 11254, pp. 144–162. Springer (2018). https://doi.org/10.1007/978-3-030-03044-5_10
7. Cucala, D.J.T., Walega, P.A., Grau, B.C., Kostylev, E.V.: Stratified negation in Datalog with metric temporal operators. In: AAAI 2021. pp. 6488–6495. AAAI Press (2021)
8. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: TIME 2005. pp. 166–174. IEEE Computer Society (2005). https://doi.org/10.1109/TIME.2005.26
9. 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)
10. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. **23**(2), 255–284 (2021). https://doi.org/10.1007/s10009-021-00609-z
11. Gorostiaga, F., Sánchez, C.: Stream runtime verification of real-time event streams with the striver language. Int. J. Softw. Tools Technol. Transf. **23**(2), 157–183 (2021). https://doi.org/10.1007/s10009-021-00605-3
12. Haftmann, F.: Code generation from specifications in higher-order logic. Ph.D. thesis, Technical University Munich (2009)
13. Havelund, K.: Rule-based runtime verification revisited. Int. J. Softw. Tools Technol. Transf. **17**(2), 143–170 (2015). https://doi.org/10.1007/s10009-014-0309-2
14. Havelund, K., Peled, D.: An extension of LTL with rules and its application to runtime verification. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 239–255. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_14
15. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. Formal Methods Syst. Des. **56**(1), 1–21 (2020). https://doi.org/10.1007/s10703-018-00327-4
16. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification – Introductory and Advanced Topics, LNCS, vol. 10457, pp. 61–102. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_3
17. Krstić, S., Schneider, J.: A benchmark generator for online first-order monitoring. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 482–494. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_27
18. Libkin, L.: Elements of Finite Model Theory. Springer (2004)

19. Ronca, A., Kaminski, M., Grau, B.C., Motik, B., Horrocks, I.: Stream reasoning in temporal Datalog. In: McIlraith, S.A., Weinberger, K.Q. (eds.) AAAI 2018. pp. 1941–1948. AAAI Press (2018)
20. Sánchez, C.: Online and offline stream runtime verification of synchronous systems. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 138–163. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_9
21. 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
22. Walega, P.A., Kaminski, M., Grau, B.C.: Reasoning over streaming data in metric temporal Datalog. In: AAAI 2019. pp. 3092–3099. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33013092
23. Yellin, D.M.: Speeding up dynamic transitive closure for bounded degree graphs. Acta Informatica **30**(4), 369–384 (1993). https://doi.org/10.1007/BF01209711
24. Zingg, S., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: VeriMon's development repository. https://bitbucket.org/jshs/monpoly/src/887b996966/thys/ (2021)