A Formally Verified Monitor for Metric First-Order Temporal Logic

Joshua Schneider, David Basin, Srđan Krstić, and Dmitriy Traytel

Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland

Abstract. Runtime verification tools must correctly establish a specification's validity or detect violations. This task is difficult, especially when the specification is given in an expressive declarative language that demands a non-trivial monitoring algorithm. We use a proof assistant to not only solve this task, but also to gain confidence in our solution. We formally verify the correctness of a monitor for metric first-order temporal logic specifications using the Isabelle/HOL proof assistant. From our formalization, we extract an executable algorithm with correctness guarantees and use differential testing to find discrepancies in the outputs of two unverified monitors for first-order specification languages.

Keywords: First-Order Monitoring · Temporal Logic · Proof Assistant

1 Introduction

Runtime verification (RV) tools are used today in safety, mission, and security-critical applications, where mistakes are too costly to be tolerated. These tools rely on complex monitoring algorithms for expressive specification languages. The correctness of these algorithms and their implementations is important and rarely obvious.

The RV community has considered different ways of improving monitors' trustworthiness by model checking monitoring algorithms [15, 21, 22] and using proof assistants to formally verify monitor instances for fixed specifications [6, 30] or entire monitors for linear temporal logic (LTL) on finite words [24] and differential dynamic logic (d$\mathcal{L}$) [7, 18]. We add to these lines of work and use the Isabelle/HOL proof assistant (Sect. 2) to develop and prove correct a monitor that supports a large fragment of metric first-order temporal logic with past and future operators (MFOTL) (Sect. 3).

Basin et al. [2] describe an efficient monitoring algorithm for MFOTL, which is implemented in the state-of-the-art monitoring tool MonPoly [3]. Our implementation deviates from the algorithm's informal description [2] in several fine points, in particular regarding the concrete representation of the monitor's state. Our formally verified algorithm closely follows MonPoly's implementation, while incorporating several simplifications regarding the evaluation order of subformulas and using simpler, less optimized data structures.

Like MonPoly, we consider a fragment of MFOTL that is monitorable using finite relations, which we represent as tables (Sect. 4). (Another version of MonPoly also supports full MFOTL using automata to represent regular relations, but is orders of magnitude less efficient.) Our monitoring algorithm processes a parametric event stream online, incrementally updates its state, and outputs verdicts specifying for every position in the event stream whether a violation has occurred and which parameters caused it (Sect. 5). We have proved the algorithm correct by establishing a complex invariant on its state and verifying that the outputted violations faithfully reflect MFOTL's semantics (Sect. 6).

Using Isabelle's code generator [9], we extract an executable OCaml implementation from our formalization. The resulting certified algorithm is integrated into MonPoly by replacing its core algorithm, while reusing its (unverified) formula and log parsers. The certified algorithm is slower than MonPoly's original algorithm. Yet it is efficient enough to process roughly 4 000 events per second on a formula with non-trivial past and future operators, whereas the original algorithm can process 23 000 events per second.

To demonstrate the verified monitor's usefulness, we perform a case study in differential testing: We compare our algorithm's output to MonPoly's on randomly generated inputs (Sect. 7). We also compare with DejaVu [11–13], a monitor for past-only first-order temporal logic. We find some discrepancies in the outputs of both tools, exhibiting corner cases where the unverified tools deviate from MFOTL's standard semantics.

In summary, we contribute a highly trustworthy monitor implementation by verifying its correctness in Isabelle/HOL. The monitor features an expressive parametric specification language with past and future metric temporal operators. Our case study confirms the usefulness of having a trusted testing oracle. We describe the formalized algorithm using concrete Isabelle syntax, demonstrating that programming in Isabelle is not different from programming in any other functional programming language. Moreover, the described algorithm can be seen as a more faithful and precise description of the MFOTL monitor than the original paper by Basin et al. [2]. With only 3 000 lines of Isabelle definitions and proofs, the verification effort was modest. The formalization is publicly available [29].

Related Work. Monitoring parametric traces and first-order specifications is bread-and-butter business in runtime verification [2, 3, 10–13, 23, 25, 26]. We refer to Havelund et al. [14] for a recent overview. Here, we discuss verification efforts targeting monitors.

Pike et. al [15,21,22] use SMT-based model-checking to increase the trustworthiness of monitors within the Copilot framework. Blech et al. [6] extract executable monitors for regular expressions from a formalization in the Coq proof assistant. However, the monitors must be proved correct manually for every property because their construction is not verified. Völlinger [30] develops a framework for certifying the output of distributed algorithms in Coq. The certification procedures that are part of this framework can be seen as concrete monitors for specific properties. Their correctness, too, must be proved manually for every distributed algorithm considered. Rizaldi et al. [24] verify a dynamic programming monitor for LTL on finite traces in Isabelle/HOL as part of their work on monitoring traffic rules. The finite trace semantics significantly simplifies their algorithm. ModelPlex is a framework for synthesizing correct-by-construction monitors for cyber-physical systems [18]. Bohrer et al. [7] further extend this work to an entire verified pipeline that culminates in the usage of a verified compiler. Both works use differential dynamic logic, which targets cyber-physical systems, but cannot easily express metric temporal properties.

More distantly related verification efforts in proof assistants include regular expression matchers [1, 20], a model checker for LTL [8], a library of timed automata [31] including a model checker [32], and relational database management systems [4, 5, 16].

In a separate line of work [27], we have extended our formalization with a framework for adaptive parallel monitoring. There parallel instances of the verified monitor must exchange parts of their states. Having the formalization of the monitor in the first place was crucial to gain trust in the correctness of this nontrivial extension.

2 Isabelle/HOL

Proof assistants are tools that mechanically check human-written proofs. They provide the highest level of trustworthiness by being built around a small, well-understood inference kernel. All proofs must pass through the kernel, which rules out invalid arguments.

Isabelle/HOL [19] is a proof assistant based on classical higher-order logic (HOL) with Hilbert choice, the axiom of infinity, and rank-1 polymorphism. HOL's syntax resembles that of functional programming languages, but with quantifiers. Isabelle features a code generator [9], which exports executable specifications to Haskell, OCaml, and Scala.

HOL's basic types include type variables $'a, 'b, \ldots$, Booleans *bool*, natural numbers *nat*, sets $'a$ *set*, pairs $'a \times 'b$, and functions $'a \Rightarrow 'b$. Functions are usually curried, and $\Rightarrow$ is right-associative. Type constructors such as $'a$ *set* are written postfix, e.g., *nat set* denotes the type of sets of natural numbers. The command **type_synonym** $t = u$ introduces an abbreviation t for an existing type u. The command **typedef** $t = S$ defines a genuinely new type from a nonempty set S over an existing type. Recursive datatypes are defined by the **datatype** command, similar to Haskell's `data`. For example, **datatype** $'a$ *list* $=$ $[] \mid \mathsf{Cons}\ 'a\ ('a\ list)$ defines finite lists. The Cons constructor is usually written infix as #.

Terms are built from variables $x, y, \ldots$, constants, function applications $f\ x$, and abstractions $\lambda x.\ z$. Function application is left-associative. We use additional notation, e.g., conditionals $\underline{\mathsf{if}}\ b\ \underline{\mathsf{then}}\ z_1\ \underline{\mathsf{else}}\ z_2$, case distinctions for datatypes $\underline{\mathsf{case}}\ d\ \underline{\mathsf{of}}\ x\#xs \Rightarrow z \mid \ldots$, and infix operators. The expression $z :: t$ denotes that the term z has type t. The command **definition** $\mathsf{c} :: t$ **where** $\mathsf{c} = z$ defines a new constant c from the term $z :: t$, which may not contain c. Recursive functions are defined by pattern-matching using **fun**. For example,

fun $\mathsf{map} :: ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
$\mathsf{map}\ f\ [] = [] \mid \mathsf{map}\ f\ (x\#xs) = f\ x\ \#\ \mathsf{map}\ f\ xs$

defines the standard list map function. Inductive predicates can be introduced differently:

inductive $\mathsf{list\_all2} :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow bool$ **where**
$\mathsf{list\_all2}\ P\ []\ [] \mid P\ x\ y \wedge \mathsf{list\_all2}\ P\ xs\ ys \longrightarrow \mathsf{list\_all2}\ P\ (x\#xs)\ (y\#ys)$

The **inductive** command defines list_all2 as the least (inductive) predicate closed under the two given rules (implications). In other words, $\mathsf{list\_all2}\ P\ xs\ ys$ is true iff the lists xs and ys have the same lengths, and their elements satisfy the binary predicate P pairwise.

We use many constructs from Isabelle's library: e.g., projections fst and snd on pairs and the minimum operator min. The type *enat* extends *nat* with infinity ∞. The set $\{x.\ P\ x\}$ contains all x satisfying P. Other set operations are $A \times B$ (Cartesian product), $A - B$ (set difference), $\bigcup x \in A.\ F\ x$ (indexed union, i.e., $\{y.\ \exists x \in A.\ y \in F\ x\}$), $\mathsf{Inf}\ A$ (infimum), and $f\ `A$ (image of A under f, i.e., $\bigcup x \in A.\ \{f\ x\}$). The sets $\{a\ ..<\ b\}$ and $\{a\ <..\ b\}$ contain all natural numbers n with $a \leq n < b$ and $a < n \leq b$, respectively. The list $[a\ ..<\ b]$ contains all of $\{a\ ..<\ b\}$ in ascending order. The datatype $'a$ *option* has two constructors $\bot$ and $\langle x :: 'a \rangle$. The term $\mathsf{map\_option}\ f$ maps $\bot$ to $\bot$ and $\langle x \rangle$ to $\langle f\ x \rangle$. Options can be converted to sets via $\lceil \bot \rceil = \{\}$ and $\lceil \langle x \rangle \rceil = \{x\}$. The function $\mathsf{these} :: 'a\ option\ set \Rightarrow 'a\ set$ maps A to $\bigcup x \in A.\ \lceil x \rceil$. The term $\mathsf{foldr}\ f\ xs\ z$ combines the elements of the list xs with the binary function f, using z as the initial value, e.g., $\mathsf{foldr}\ (-)\ [1,2]\ 3 = 1 - (2 - 3)$. The set of all elements in the list xs is $\mathsf{set}\ xs$, the length of xs is $\mathsf{length}\ xs$, the i-th element of xs is $xs\,!\,i$ (zero-based, requires $i < \mathsf{length}\ xs$), and list concatenation is $xs\ @\ ys$. Strings *string* are character lists. Streams $'a$ *stream* are infinite sequences of values of type $'a$.

type_synonym *name* = *string* **type_synonym** *domain* = *string*
type_synonym *db* = (*name* × *domain list*) *set* **type_synonym** *ts* = *nat*

typedef *trace* = {*s* :: (*db* × *ts*) *stream*. wf_trace *s*}

datatype *trm* = V *nat* | C *domain* **typedef** $\mathcal{I}$ = {(*a* :: *nat*, *b* :: *enat*). *a* ≤ *b*}

datatype *frm* = Pred *name* (*trm list*) | Eq *trm trm* | Neg *frm* | Or *frm frm* | Exists *frm*
| Since *frm* $\mathcal{I}$ *frm* | Until *frm* $\mathcal{I}$ *frm*

fun eval_trm :: *domain list* ⇒ *trm* ⇒ *domain* **where** eval_trm *v* (V *x*) = *v* ! *x* | eval_trm *v* (C *x*) = *x*

fun sat :: *trace* ⇒ *domain list* ⇒ *nat* ⇒ *frm* ⇒ *bool* **where**
sat σ *v i* (Pred *r ts*) = ((*r*, map (eval_trm *v*) *ts*) ∈ Γ σ *i*)
| sat σ *v i* (Eq t_1 t_2) = (eval_trm *v* t_1 = eval_trm *v* t_2) | sat σ *v i* (Neg ψ) = (¬ sat σ *v i* ψ)
| sat σ *v i* (Exists ψ) = (∃*z*. sat σ (*z*#*v*) *i* ψ) | sat σ *v i* (Or α β) = (sat σ *v i* α ∨ sat σ *v i* β)
| sat σ *v i* (Since α *I* β) = (∃*j* ≤ *i*. τ σ *i* − τ σ *j* ∈$_{\mathcal{I}}$ *I* ∧ sat σ *v j* β ∧ (∀*k* ∈ {*j* <.. *i*}. sat σ *v k* α))
| sat σ *v i* (Until α *I* β) = (∃*j* ≥ *i*. τ σ *j* − τ σ *i* ∈$_{\mathcal{I}}$ *I* ∧ sat σ *v j* β ∧ (∀*k* ∈ {*i* ..< *j*}. sat σ *v k* α))

Fig. 1. Syntax and semantics of MFOTL

3 Metric First-Order Temporal Logic

We interpret MFOTL over infinite streams of time-stamped events. Figure 1 shows the types of event streams and formulas along with the relation sat defining the semantics. Events consist of a name and a list of parameters from some domain (*name* × *domain list*). The *name* and *domain* types are arbitrary; we choose strings for convenience. We group concurrent events into databases (*db*). Time-stamps are discrete and modeled as natural numbers (*ts*). An event stream (*trace*) is an infinite stream of time-stamped databases. We write Γ σ *i* for the *i*-th database in event stream σ and τ σ *i* for the corresponding time-stamp, where *i* is a zero-based index. The predicate wf_trace in *trace*'s definition ensures that the time-stamp sequence is monotonic (∀*i*. τ σ *i* ≤ τ σ (*i* + 1)) and unbounded (∀*t*. ∃*i*. *t* < τ σ *i*). A (stream) prefix π is a finite list of time-stamped databases. It satisfies wf_prefix iff the prefix π has monotonic time-stamps. We write prefix_of π σ if the event stream σ extends the prefix π, i.e., the sequence of σ's first length π elements equals π.

The datatypes for terms (*trm*) and formulas (*frm*) are mostly standard. We use De Bruijn indices to represent free and bound variables, e.g., ∃*y*. *A*(*x*, *y*) is encoded as Exists (Pred A [V 1, V 0]). In examples, we will show both the standard notation and the concrete encoding. The term fv φ denotes the set of φ's free variables. The degree nfv φ is the least number *n* such that for than any *x* ∈ fv φ we have *x* < *n*. The predicate is_Const tests whether its argument is C *d* for some *d*. The type $\mathcal{I}$ models nonempty intervals over the natural numbers. The term interval *a b* represents the interval from *a* to *b* (both inclusive), and point *c* = interval *c c*. We write left *I* and right *I* for the endpoints of *I* : $\mathcal{I}$, and *n* ∈$_{\mathcal{I}}$ *I* for the membership of *n* in *I*. We use abbreviations for some derived operators: And α β = Neg (Or (Neg α) (Neg β)), AndNot α β = Neg (Or (Neg α) β), TT = Eq (C *d*) (C *d*), where *d* is an arbitrary domain value, and Eventually *I* ψ = Until TT *I* ψ. We omit the operators *previous* and *next* from our presentation. These operators are implemented in the formalization [29].

We have sat σ *v i* φ iff the formula φ is satisfied by the valuation *v* at index *i*, given the event stream σ. Valuations are modeled as lists of domain values, the first element being the assignment to the variable with index 0, the second to the variable 1, and so forth.

type_synonym *tuple* = *domain option list* **type_synonym** *table* = *tuple set*

definition wf_tuple :: *nat* ⇒ *nat set* ⇒ *tuple* ⇒ *bool* **where**
  wf_tuple *n V v* = (length *v* = *n* ∧ (∀*i* < *n*. *v* ! *i* = ⊥ ⟷ *i* ∉ *V*))

definition wf_table :: *nat* ⇒ *nat set* ⇒ (*tuple* ⇒ *bool*) ⇒ *table* ⇒ *bool* **where**
  wf_table *n V Q A* = (∀*v*. *v* ∈ *A* ⟷ (*Q v* ∧ wf_tuple *n V v*))

fun join1 :: *tuple* × *tuple* ⇒ *tuple option* **where**
  join1 ([], []) = ⟨[]⟩
| join1 (⊥ # *xs*, ⊥ # *ys*) = map_option (λ*zs*. ⊥ # *zs*) (join1 (*xs*, *ys*))
| join1 (⟨*x*⟩ # *xs*, ⊥ # *ys*) = map_option (λ*zs*. ⟨*x*⟩ # *zs*) (join1 (*xs*, *ys*))
| join1 (⊥ # *xs*, ⟨*y*⟩ # *ys*) = map_option (λ*zs*. ⟨*y*⟩ # *zs*) (join1 (*xs*, *ys*))
| join1 (⟨*x*⟩ # *xs*, ⟨*y*⟩ # *ys*) = (if *x* = *y* then map_option (λ*zs*. ⟨*x*⟩ # *zs*) (join1 (*xs*, *ys*)) else ⊥)
| join1 (_, _) = ⊥

definition join :: *bool* ⇒ *table* ⇒ *table* ⇒ *table* **where**
  join *p A B* = (if *p* then these (join1 ‘ (*A* × *B*)) else *A* − these (join1 ‘ (*A* × *B*)))

Fig. 2. Finite tables

4 Finite Tables

Our monitor computes and outputs all satisfying valuations of a formula at all indices. To do so efficiently, it operates on finite sets of valuations, which can be viewed as finite tables, and manipulates them using standard relational operations like natural join.

A way to represent finite sets of valuations for a given formula φ is to use sets of n-ary tuples (i.e., lists of length n), where n is the number of free variables in φ. The representation must map free variables to positions in the tuple and the natural join operation changes the arity of tuples. We chose a slightly different representation to simplify the implementation of union and join: Our tuples xs are lists of *optional* domain values, which assign values only to those variables i whose corresponding entries $xs\,!\,i$ are not $\bot$. This allows us to use tuples of a fixed length n, regardless of the formula's free variables, while ensuring that for any subformula all its free variables given by a set V are assigned, as specified by the well-formedness predicate wf_tuple (Fig. 2). We use the statement wf_tuple n (fv φ) v with nfv $\varphi \leq n$ to express that v is a well-formed tuple for φ. We obtain the corresponding valuation $\overline{v}$ = map the v, where the function the maps $\langle x \rangle$ to x and $\bot$ to some unspecified domain element. The actual value assigned to $\bot$ is irrelevant for the valuation's satisfaction since it is only assigned to variables that are not free in φ.

A well-formed table A, written wf_table n V Q A, is a set of well-formed tuples. The parameter Q is a predicate on tuples that further restricts our attention to those tuples satisfying Q. Typically, Q will be instantiated by sat expressing that a table A consists of precisely the well-formed tuples that satisfy a given formula φ on stream σ at index i, i.e., wf_table n (fv φ) (λv. sat σ $\overline{v}$ i φ) A, where nfv $\varphi \leq n$. The abbreviation table n V A = wf_table n V (λv. $v \in A$) A expresses that A only contains well-formed tuples.

Using this representation, the union of tables A and B both satisfying table n V is just the set union $A \cup B$, satisfying table n V $(A \cup B)$. The natural join operation is more involved. We first show how to join two individual tuples. We define this function join1 recursively (Fig. 2) assuming that the two input tuples have the same length (but not necessarily the same variables being set to $\bot$). The function join1 returns an optional tuple, where $\bot$ indicates either that the inputs do not have the same length (last equation) or that they are not joinable, i.e., have conflicting assigned domain values (the else branch in

type_synonym *buf* = *table list* × *table list* **type_synonym** *saux* = (*ts* × *table*) *list*
type_synonym *uaux* = (*ts* × *table* × *table*) *list*

datatype *state* = $\mathsf{Eq_S}$ *table* | $\mathsf{Pred_S}$ *name* (*trm list*)
| $\mathsf{And_S}$ *state bool state buf* | $\mathsf{Or_S}$ *state state buf* | $\mathsf{Exists_S}$ *state*
| $\mathsf{Since_S}$ *bool state* $\mathcal{I}$ *state buf* (*ts list*) *saux* | $\mathsf{Until_S}$ *bool state* $\mathcal{I}$ *state buf* (*ts list*) *uaux*

datatype *mstate* = MState *nat nat state*

definition init :: *frm* ⇒ *mstate* **where** …
fun step :: *db* × *ts* ⇒ *mstate* ⇒ (*nat* × *tuple*) *set* × *mstate* **where** …
fun steps :: (*db* × *ts*) *list* ⇒ *mstate* ⇒ (*nat* × *tuple*) *set* **where**
steps [] _ = {} | steps (*tdb*#π) *s* = (let (*A*,*s*′) = step *tdb s* in *A* ∪ steps π *s*′)

Fig. 3. The monitor's state and its high-level interface

the if-expression). The key property of join1 is its correspondence to logical conjunction:

$$\mathsf{wf\_tuple}\ n\ V\ v \wedge \mathsf{wf\_tuple}\ n\ W\ w \longrightarrow \mathsf{join1}\ (v,w) = \langle z\rangle \longleftrightarrow (\mathsf{wf\_tuple}\ n\ (V \cup W)\ z \wedge v = z \downarrow V \wedge w = z \downarrow W),$$

where $v \downarrow V$ maps all domain elements assigned to variables outside of the set V to $\bot$; formally, $v \downarrow V = \mathsf{map}\ (\lambda i.\ \mathsf{if}\ i \in V\ \mathsf{then}\ v\,!\,i\ \mathsf{else}\ \bot)\ [0\ ..{<}\ \mathsf{length}\ v]$.

The function join (Fig. 2) lifts join1 to tables, where the Boolean p indicates whether a join ($p = \mathsf{True}$) or an anti-join ($p = \mathsf{False}$) is computed. Naturally, join's key property is similar to join1's, but now expressed on tables. For the anti-join, the negated part's variables (W) must be contained in those of the non-negated part (V) to ensure finiteness.

$$\mathsf{table}\ n\ V\ A \wedge \mathsf{table}\ n\ W\ B \wedge (\neg p \longrightarrow W \subseteq V) \longrightarrow z \in \mathsf{join}\ p\ A\ B \longleftrightarrow (\mathsf{wf\_tuple}\ n\ (V \cup W)\ z \wedge z \downarrow V \in A \wedge (p \longleftrightarrow z \downarrow W \in B))$$

Above, join computes $A \times B$ before applying join1. We prove and use for code generation the more space-efficient definition $\mathsf{join}\ \mathsf{True}\ A\ B = \bigcup v \in A.\ \bigcup w \in B.\ \lceil \mathsf{join1}\ (v,w) \rceil$.

5 Monitor

A monitor takes an MFOTL formula φ and an event stream σ as inputs. It computes *satisfactions*: pairs (i,v) of indices i and valuations v that satisfy the formula, i.e., $\mathsf{sat}\ \sigma\ \overline{v}\ i\ \varphi$. One is often interested in finding the *violations* of a formula φ, which are the pairs (i,v) such that $\neg\,\mathsf{sat}\ \sigma\ \overline{v}\ i\ \varphi$. Violations can be obtained by monitoring the negated formula.

A monitor cannot directly process an infinite event stream. Instead, in the *offline* setting, the monitor computes satisfactions for a single stream prefix. In the *online* setting, the monitor processes an unbounded stream incrementally and produces intermediate outputs. Our monitor always receives a whole time-stamped database at once, since MFOTL formulas cannot distinguish the order and arrival time of events within a database.

Figure 3 shows our monitor's state type (*mstate*) and its online and offline interface. The online interface is a transition system given by two functions: init, which computes the initial state, and step, which updates the state with a new input (a time-stamped database) and outputs satisfactions. The offline interface is the function $\mathsf{monitor}\ \varphi\ \pi = \mathsf{steps}\ \pi\ (\mathsf{init}\ \varphi)$, where steps iterates step on a prefix and collects all satisfactions in a set.

Example 1. Consider the formula $A(x) \longrightarrow \Diamond_{[1,2]}(\exists y.\, B(x,y))$, i.e., all A events must be followed by a matching B event after one or two time units. To obtain violations, we monitor the negation $A(x) \wedge \neg \Diamond_{[1,2]}(\exists y.\, B(x,y))$, which we encode as

$\varphi_{ex} = \mathsf{AndNot}\ (\mathsf{Pred}\ \mathtt{A}\ [\mathsf{V}\ 0])\ \big(\mathsf{Eventually}\ (\mathsf{interval}\ 1\ 2)\ (\mathsf{Exists}\ (\mathsf{Pred}\ \mathtt{B}\ [\mathsf{V}\ 1, \mathsf{V}\ 0]))\big)$.

Given the prefix $\pi_{ex} = \big[(\{(\mathtt{A},[\mathtt{d}]),(\mathtt{A},[\mathtt{e}])\},1),(\{(\mathtt{B},[\mathtt{d},\mathtt{f}])\},2),(\{(\mathtt{B},[\mathtt{e},\mathtt{f}])\},5)\big]$, which consists of three databases with indices 0, 1, 2 and time-stamps 1, 2, and 5, with four events in total, there is one satisfaction: $\mathsf{monitor}\ \varphi_{ex}\ \pi_{ex} = \{(0,[\langle \mathtt{e} \rangle])\}$. The satisfaction originates from the event $(\mathtt{A},[\mathtt{e}])$, which is part of the database with index 0 in π_{ex}. The satisfaction's valuation is $[\langle \mathtt{e} \rangle]$ because the parameter of $(\mathtt{A},[\mathtt{e}])$ is bound to φ_{ex}'s first (and only) free variable. The satisfaction is output after processing the third database:

$$\begin{aligned}
&\mathsf{step}\ (\{(\mathtt{A},[\mathtt{d}]),(\mathtt{A},[\mathtt{e}])\},1)\ (\mathsf{init}\ \varphi_{ex}) &&= (\{\}, s_1)\\
&\mathsf{step}\ (\{(\mathtt{B},[\mathtt{d},\mathtt{f}])\},2)\quad s_1 &&= (\{\}, s_2)\\
&\mathsf{step}\ (\{(\mathtt{B},[\mathtt{e},\mathtt{f}])\},5)\quad s_2 &&= (\{(0,[\langle \mathtt{e} \rangle])\}, s_3),
\end{aligned}$$

where s_1, s_2, and s_3 are the monitor's states after processing each input.

Overview of the Algorithm. We require satisfactions to be output in the order they occur. Namely, (i_1, v_1) cannot be output after (i_2, v_2) if $i_1 < i_2$. Therefore, the monitor's state is characterized by its *progress*, which we represent by a stream index i. The progress is the smallest index for which new satisfactions cannot be computed without receiving more databases. It is initially zero and always at most the number of databases received. It is generally not possible to compute all satisfactions for an index j after processing the j-th input when monitoring a formula with future operators. For example, if the j-th input contains the event $(\mathtt{A},[d])$, we do not know whether $(j,[\langle d \rangle])$ satisfies φ_{ex} from Example 1 until we either observe a matching B event or a time-stamp that is at least three units ahead.

For every input database, step advances i by recursively evaluating the monitored formula. The evaluation of a subformula ψ at index i yields a table containing all valuations v with $\mathsf{sat}\ \sigma\ \overline{v}\ i\ \psi$. For any binary operator in the formula, it may be possible to evaluate its subformulas up to different indices, e.g., if one subformula contains a future operator and the other does not. Our monitor evaluates subformulas as far as possible. Therefore, every subformula ψ has its own progress i_ψ describing how far it has been evaluated. (We omit the subscript when it is clear from the context.) Since several indices might be resolved at once by a new input, the evaluation result is a list $xs :: \mathit{table\ list}$. Its elements correspond to indices $[i\ ..{<}\ i + \mathsf{length}\ xs]$ according to their position in the list.

Recall that tables are finite sets. Evaluation of a subformula must therefore not result in infinitely many satisfying valuations. This is not guaranteed for all MFOTL formulas. For example, the formula $\neg A(x)$ (i.e., $\mathsf{Neg}\ (\mathsf{Pred}\ \mathtt{A}\ [\mathsf{V}\ 0])$) has infinitely many satisfying valuations at each index, regardless of the event stream. We, therefore, adopt the restriction to a syntactic fragment of MFOTL that is used in the table-based variant of MonPoly [2]. A formula is *monitorable* if and only if its satisfies the recursive predicate mf (Fig. 4). Note that the negation of φ must be monitorable if we search for violations of φ, which is generally different from the monitorability of φ itself. Basin et al. [2] describe a heuristic that attempts to rewrite formulas into equivalent, monitorable ones.

An equality $\mathsf{Eq}\ t_1\ t_2$ is only monitorable if at least one of the terms is a constant (otherwise, an infinite number of valuations satisfy $x = x$, i.e., $\mathsf{Eq}\ (\mathsf{V}\ 0)\ (\mathsf{V}\ 0)$). In general, monitorable formulas may contain negations only in specific places. The pattern $\mathsf{Neg}\ (\mathsf{Or}\ (\mathsf{Neg}\ \alpha)\ (\mathsf{Neg}\ \beta))$ corresponds to a conjunction $\mathsf{And}\ \alpha\ \beta$, which is always

fun mf :: *frm* ⇒ *bool* **where**
  mf (Eq t_1 t_2) = (is_Const t_1 ∨ is_Const t_2)
| mf (Pred *e trms*) = True
| mf (Neg (Or (Neg α) β)) = (mf α ∧ (mf β ∧ fv β ⊆ fv α ∨ mf $?_{\mathsf{Neg}}$ β)
| mf (Or α β) = (fv α = fv β ∧ mf α ∧ mf β)
| mf (Exists ψ) = mf ψ
| mf (Since α I β) = (fv α ⊆ fv β ∧ (mf α ∨ mf $?_{\mathsf{Neg}}$ α) ∧ mf β)
| mf (Until α I β) = (fv α ⊆ fv β ∧ (mf α ∨ mf $?_{\mathsf{Neg}}$ α) ∧ mf β ∧ right $I \neq \infty$)
| mf _ = False

Fig. 4. Monitorable formulas (f $?_{\mathsf{Neg}}$ φ abbreviates case φ of Neg $\varphi' \Rightarrow f\ \varphi'$ | _ ⇒ False)

fun init0 :: *nat* ⇒ *frm* ⇒ *state* **where** …
fun eval :: *nat* ⇒ *ts* ⇒ *db* ⇒ *state* ⇒ *table list* × *state* **where** …

definition init :: *frm* ⇒ *mstate* **where** init φ = (let n = nfv φ in MState n 0 (init0 n φ))

fun step :: *db* × *ts* ⇒ *mstate* ⇒ (*nat* × *tuple*) *set* × *mstate* **where**
  step (*db*,*t*) (MState n i s) = (let (xs, s') = eval n *t db* s
    in ($\bigcup(k,V) \in$ set (enumerate i xs). $\bigcup v \in V.\ \{(k,v)\}$, MState n (i + length xs) s'))

Fig. 5. Initialization and step functions of the monitor

monitorable if α and β are monitorable. For AndNot α β, we additionally require that all variables free in β are free in α. This rules out formulas like $A(x) \wedge \neg B(y)$ (i.e., AndNot (Pred A [V 0]) (Pred B [V 1])), which has infinitely many satisfactions if the stream contains at least one A event. The subformulas α and β of Or α β must have exactly the same free variables for similar reasons. The temporal operators Since and Until allow a negated left subformula α = Neg α' even if α itself is not monitorable. However, there is always a restriction on the free variables. For example, Since α I β is already satisfied at index i if β is satisfied at i. Any free variable in α that is not free in β could thus be assigned any value, and the resulting table would be infinite. Moreover, the future reach of Until α I β must be bounded to ensure that the monitor can make progress.

The monitor's state MState n i s consists of the formula's degree n = nfv φ (to avoid recomputation), the progress i, and a formula state s. The formula state datatype *state* (Fig. 3) extends the abstract syntax tree of formulas with the state that is associated with the formula's operators. It restricts the syntax to a superset of the monitorable formulas, such that the evaluation can be implemented directly as a recursive function on *state*.

The monitor's entry points are defined in Fig. 5. The function init uses init0 to convert the formula recursively into a formula state. We omit init0's definition, which follows mf's definition. Some of *state*'s constructors carry a Boolean flag p that indicates whether one of the subformulas is positive (p = True) or negated (p = False). In those cases where a negated subformula is not monitorable, we remove the negation before the recursive conversion and set p to False. For example, $\alpha \wedge \neg\beta$ (i.e., AndNot α β) is converted to $\mathsf{And}_{\mathsf{S}}$ (init0 n α) False (init0 n β) ([], []). All lists in the *state* are initially empty.

The step function step is a wrapper for eval that evaluates a formula state given the formula's degree and the new time-stamp and database. It returns a list of tables for all indices that could be evaluated, and the updated *state*. We cover all cases of eval's definition in the following subsections. The standard function enumerate i xs maps the elements V of xs to pairs (k,V), where the numbers k increase sequentially starting at i.

```
fun match :: trm list ⇒ domain list ⇒ (nat ⇒ domain option) option where
  match [] [] = ⟨λx. ⊥⟩
| match (C x#trms) (y#ys) = (if x = y then match trms ys else ⊥)
| match (V x#trms) (y#ys) = (case match trms ys of ⊥ ⇒ ⊥
    | ⟨f⟩ ⇒ (case f x of ⊥ ⇒ ⟨f(x ↦ y)⟩ | ⟨z⟩ ⇒ if y = z then ⟨f⟩ else ⊥))
| match _ _ = ⊥
```

Fig. 6. The match function

```
fun buf_add :: table list ⇒ table list ⇒ buf ⇒ buf where
  buf_add xs' ys' (xs, ys) = (xs @ xs', ys @ ys')
fun buf_take :: (table ⇒ table ⇒ 'b) ⇒ buf ⇒ 'b list × buf
  buf_take f (x#xs, y#ys) = (let (zs, b) = buf_take f (xs, ys) in (f x y#zs, b))
| buf_take f (xs, ys) = ([], (xs, ys))
```

Fig. 7. Buffer operations

Atomic Formulas. The constructor $\mathsf{Eq_S}$ of *state* represents constant tables corresponding to (monitorable) equalities. The associated *state* is always the same table, which is returned upon evaluation. For a predicate's state $\mathsf{Pred_S}\ e\ trms$, we first select all events in the database *db* with the name *e*. The auxiliary function match, defined in Fig. 6, is applied to each selected event. This function attempts to compute the unique valuation for the variables in *trms* that makes the terms match the event's parameters. It returns $\langle f \rangle$ if such a valuation f exists, and $\bot$ otherwise. To simplify match's definition, f is encoded as a partial function $nat \Rightarrow domain\ option$. We convert it into a tuple using $\mathsf{map}\ f\ [0\ ..{<}\ n]$.

```
  eval n t db (Eq_S r) = ([r], Eq_S r)
| eval n t db (Pred_S e trms) = ([(λf. map f [0 ..< n]) ‘ these
      (match trms ‘ (⋃(e', x) ∈ db. if e = e' then {x} else {}))], Pred_S e trms)
```

Non-Temporal Operators. It may be possible to evaluate the two subformulas α and β of a binary operator up to different indices $i_\alpha \neq i_\beta$. Then, the operator itself can only be evaluated up to the minimum of i_α and i_β. The remaining tables obtained from the subformula that is further ahead must be stored until more results are available from the other subformula. We store the tables in a *buffer* of type *buf*, which consists of one list for each subformula. The lists act as queues: new results are appended, and whenever both lists are nonempty, the subformula can be evaluated by removing pairs of tables from the front. The function $\mathsf{buf\_add}\ xs'\ ys'\ b$ (Fig. 7) adds the result lists xs' and ys' from the two subformulas to the buffer *b*. The function $\mathsf{buf\_take}\ f\ b$ removes pairs of tables from the front of the buffer and applies the operator-specific function f to them, collecting a list of results.

For $\mathsf{And_S}$ and $\mathsf{Or_S}$, we evaluate both subformulas and obtain two result lists *xs* and *ys*, as well as the updated subformula states s_1' and s_2'. The results are added to the buffer *b*. Then, buf_take combines the results that are available for both subformulas into the results of the operator, using an (anti-)join for conjunctions and a union for disjunctions.

```
| eval n t db (And_S s1 p s2 b) = (let (xs, s1') = eval n t db s1; (ys, s2') = eval n t db s2
    (zs, b) = buf_take (join p) (buf_add xs ys b) in (zs, And_S s1' p s2' b)
| eval n t db (Or_S s1 s2 b) = (let (xs, s1') = eval n t db s1; (ys, s2') = eval n t db s2;
    (zs, b) = buf_take (∪) (buf_add xs ys b) in (zs, Or_S s1' s2' b)
```

```
definition update_since :: 𝓘 ⇒ bool ⇒ table ⇒ table ⇒ ts ⇒ saux ⇒ table × saux where
  update_since I p r1 r2 nt aux = (let aux = (case [(t, join r p r1). (t, r) ← aux, nt − t ≤ right I] of
      [] ⇒ [(nt, r2)]
    | x#aux' ⇒ (if fst x = nt then (fst x, snd x ∪ r2)#aux' else (nt, r2)#x#aux'))
    in (foldr (∪) [r. (t, r) ← aux, left I ≤ nt − t] {}, aux)
fun update_until :: 𝓘 ⇒ bool ⇒ table ⇒ table ⇒ ts ⇒ uaux ⇒ uaux where
  update_until I p r1 r2 nt aux = (map (λx. case x of (t, a1, a2) ⇒
    (t, if p then join a1 True r1 else a1 ∪ r1, if nt − t ∈𝓘 I then a2 ∪ join r2 p a1 else a2)) aux)@
    [(nt, r1, if left I = 0 then r2 else {})]
fun eval_until :: 𝓘 ⇒ ts ⇒ uaux ⇒ table list × uaux
  eval_until I nnt [] = ([], [])
| eval_until I nnt ((t, a1, a2)#aux) = (if t + right I < nnt
    then (let (xs, aux) = eval_until I nnt aux in (a2#xs, aux)) else ([], (t, a1, a2)#aux))
fun tbuf_take :: (table ⇒ table ⇒ ts ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ buf ⇒ ts list ⇒ 'b × buf × ts list where
  tbuf_take f z (x#xs, y#ys) (t#ts) = tbuf_take f (f x y t z) (xs, ys) ts
| tbuf_take f z (xs, ys) ts = (z, (xs, ys), ts)
```

Fig. 8. Auxiliary operations for evaluating $\mathsf{Since}_\mathsf{S}$ and $\mathsf{Until}_\mathsf{S}$

We increment the degree in the recursive computation of an existential quantifier $\mathsf{Exists}_\mathsf{S}$ to account for the variables' De Bruijn encoding. Each computed tuple v (which is a list) is then replaced by its tail $\mathsf{tl}\ v$ to remove the assignment to the bound variable.

```
| eval n t db (Exists_S s) = (let (xs, s') = eval (n + 1) t db s
    in (map (λr. tl ‘ r) xs, Exists_S s')
```

Since and Until. The monitoring algorithm implements $\mathsf{Since}\ \alpha\ I\ \beta$ by decomposing the interval I. Note that $\mathsf{Since}\ \alpha\ I\ \beta$ is equivalent to a disjunction of $\mathsf{Since}\ \alpha\ (\mathsf{point}\ c)\ \beta$, where c ranges over I. We can additionally bound c from above by the time-stamp $\tau\ \sigma\ (i-1)$, where i is the operator's progress. This ensures that the disjunction always consists of finitely many terms (even if $\mathsf{right}\ I = \infty$). We store the satisfactions for $\mathsf{Since}\ \alpha\ (\mathsf{point}\ c)\ \beta$ in a list in the monitor's state, together with time-stamps $\tau\ \sigma\ (i-1) - c$. The list, called the *auxiliary state*, is sorted on the time-stamps c. It also contains satisfactions for $c < \mathsf{left}\ I$ (if $\mathsf{left}\ I > 0$) because these may move into the interval I as time progresses.

For every new input database, the function update_since (Fig. 8) updates the list to maintain the correspondence with $\mathsf{Since}\ \alpha\ (\mathsf{point}\ c)\ \beta$. It also computes the satisfactions by taking the union over all tables in the list that satisfy $c \in_\mathcal{I} I$. The arguments of update_since are the interval I of the Since operator, a flag p indicating whether the left subformula is positive (not negated), the subformulas' results r_1 and r_2 at index i, the time-stamp $nt = \tau\ \sigma\ i$, and the old auxiliary state *aux*. We assume that the left subformula is evaluated without the negation. If the new time-stamp $\tau\ \sigma\ i$ differs from the previous time-stamp $\tau\ \sigma\ (i-1)$, i.e., $\Delta = \tau\ \sigma\ i - \tau\ \sigma\ (i-1) \neq 0$, the tables in the old auxiliary state now represent $\mathsf{Since}\ \alpha\ (\mathsf{point}\ (c+\Delta))\ \beta$, but without taking the satisfactions of α and β at i into account. First, update_since removes all tables for which $c+\Delta$ exceeds the right bound of the interval. It then joins each remaining table with the result r_1 for α, and adds the satisfactions r_2 for β either to the first table (if $c+\Delta = 0$) or as a new list element to the list.

Decomposing a formula $\psi = \mathsf{Until}\ \alpha\ I\ \beta$ into point intervals is not as useful because there is no obvious way to compute the satisfactions of $\mathsf{Until}\ \alpha\ (\mathsf{point}\ c)\ \beta$ at index $i+1$

from those at index i, which would allow us to reuse previous computations. Another difference to Since is that we cannot immediately output the satisfactions once we have the subformulas' results. A new input may still change the satisfactions for previous indices.

Let i^* be the minimum of i_α and i_β. The auxiliary state for Until, which has type *uaux*, stores for all k in $\{i_\psi \mathinner{..<} i^*\}$ the time-stamp $\tau\ \sigma\ k$ and two tables a_1 and a_2, sorted by k. The meaning of the tables depends on the flag p, which indicates whether the left subformula α is positive. If $p = \mathsf{True}$, the table a_1 contains the valuations satisfying α at all indices in $\{k \mathinner{..<} i^*\}$. If $p = \mathsf{False}$, it contains the valuations satisfying $\mathsf{Neg}\ \alpha$ at some index in $\{k \mathinner{..<} i^*\}$. The table a_2 contains the valuation satisfying β at some index k' in $\{k \mathinner{..<} i^*\}$ with $\tau\ \sigma\ k' - \tau\ \sigma\ k \in_{\mathcal{I}} I$, and satisfying α for all indices in $\{k \mathinner{..<} k'\}$. Note that this is not the same as the satisfactions for $\mathsf{Until}\ \alpha\ I\ \beta$ at k because the interval may be incomplete between k and i^*. The function update_until (Fig. 8) maintains this invariant for every advance of i^*. Its arguments have the same meaning as for update_since. However, it does not compute the results, which is instead done by eval_until. Its argument *nnt* denotes the time-stamp $\tau\ \sigma\ (i^* + 1)$, or $\tau\ \sigma\ j$ if j is the most recent input database and $i^* = j$. The function retrieves those tables a_2 for which the interval is complete, i.e., *nnt* is more than right I units ahead of the associated time-stamp t.

The implementation of eval for $\mathsf{Since}_\mathsf{S}$ and $\mathsf{Until}_\mathsf{S}$ follows the other binary operators, but with an additional update step for the auxiliary state.

$$
\begin{aligned}
&\mid \mathsf{eval}\ n\ t\ db\ (\mathsf{Since}_\mathsf{S}\ p\ s_1\ I\ s_2\ b\ ts\ aux) = (\underline{\mathsf{let}}\ (xs, s_1') = \mathsf{eval}\ n\ t\ db\ s_1;\\
&\quad (ys, s_2') = \mathsf{eval}\ n\ t\ db\ s_2;\ ((zs, aux), b, ts) = \mathsf{tbuf\_take}\ (\lambda r_1\ r_2\ t\ (zs, aux).\\
&\qquad\quad \underline{\mathsf{let}}\ (z, aux) = \mathsf{update\_since}\ I\ p\ r_1\ r_2\ t\ aux\ \underline{\mathsf{in}}\ (zs\ @\ [z], aux))\\
&\qquad ([], aux)\ (\mathsf{buf\_add}\ xs\ ys\ b)\ (ts\ @\ [t])\\
&\quad \underline{\mathsf{in}}\ (zs, \mathsf{Since}_\mathsf{S}\ p\ s_1'\ I\ s_2'\ b\ ts\ aux)\\
&\mid \mathsf{eval}\ n\ t\ db\ (\mathsf{Until}_\mathsf{S}\ p\ s_1\ I\ s_2\ b\ ts\ aux) = (\underline{\mathsf{let}}\ (xs, s_1') = \mathsf{eval}\ n\ t\ db\ s_1;\\
&\quad (ys, s_2') = \mathsf{eval}\ n\ t\ db\ s_2;\ ((zs, aux), b, ts) =\\
&\qquad \mathsf{tbuf\_take}\ (\mathsf{update\_until}\ I\ p)\ aux\ (\mathsf{buf\_add}\ xs\ ys\ b)\ (ts\ @\ [t]);\\
&\quad (zs, aux) = \mathsf{eval\_until}\ I\ (\underline{\mathsf{case}}\ ts\ \underline{\mathsf{of}}\ [] \Rightarrow t \mid t'\#\_ \Rightarrow t')\ aux\\
&\quad \underline{\mathsf{in}}\ (zs, \mathsf{Until}_\mathsf{S}\ p\ s_1'\ I\ s_2'\ b\ ts\ aux)
\end{aligned}
$$

Here, tbuf_take (Fig. 8) is used instead of buf_take as we must consider the time-stamps $ts\ @\ [t]$ corresponding to the subformulas' (future) results. Unlike buf_take, this function does not apply f individually, but it folds all results from left to right.

6 Correctness

We define a formal invariant for the monitor's state, which connects its structure with MFOTL's semantics and the stream prefix observed so far. We then prove that init establishes and that step preserves the invariant. Moreover, we show that the satisfactions output by steps are sound and eventually complete for monitorable formulas.

The invariant relates the tables stored in a formula state to the semantics of the corresponding subformulas. In Sect. 5, we introduced the notion of progress i_ψ, which states how far the subformula ψ has been evaluated. The function prog in Fig. 9 defines i_ψ concretely. Its arguments are the trace σ, an arbitrary MFOTL formula ψ, and the index j of the next time-stamped database to be received by the monitor. (Initially, j is zero, and every application of step increases it by one.) Predicates and equalities can always be evalu-

fun prog :: *trace* ⇒ *frm* ⇒ *nat* ⇒ *nat* **where**
  prog σ (Pred e ts) $j = j$ | prog σ (Eq t_1 t_2) $j = j$
| prog σ (Neg ψ) j = prog σ ψ j | prog σ (Or α β) j = min (prog σ α j) (prog σ β j)
| prog σ (Exists ψ) j = prog σ ψ j | prog σ (Since α I β) j = min (prog σ α j) (prog σ β j)
| prog σ (Until α I β) j =
    Inf $\{i.\ \forall k.\ k < j \wedge k \leq$ min (prog σ α j) (prog σ β j) $\longrightarrow \tau\ \sigma\ i +$ right $I \geq \tau\ \sigma\ k\}$
definition prog2 σ α β j = min (prog σ α j) (prog σ β j)

Fig. 9. Progress of the monitor

definition wf_mstate :: *frm* ⇒ (*db* × *ts*) *list* ⇒ *mstate* ⇒ *bool* **where**
  wf_mstate φ π (MState n i s) $\longleftrightarrow$ wf_prefix $\pi \wedge n =$ nfv $\varphi \wedge (\forall \sigma.$ prefix_of $\pi\ \sigma \longrightarrow$
    $i =$ prog σ φ (length π) $\wedge$ wf_state σ (length π) n s φ)
inductive wf_state :: *trace* ⇒ *nat* ⇒ *nat* ⇒ *state* ⇒ *frm* ⇒ *bool* **where**
  ...
| wf_state σ j n s_1 $\alpha \wedge$ wf_state σ j n s_2 $\beta \wedge$
    $(p \longrightarrow \alpha' = \alpha) \wedge (\neg p \longrightarrow \alpha' =$ Neg $\alpha) \wedge$ mf $\alpha = p \wedge$ fv $\alpha \subseteq$ fv $\beta \wedge$
    wf_buf σ j n α β $b \wedge$ wf_ts σ j α β $ts \wedge$ wf_uaux σ j n p α I β $aux \longrightarrow$
  wf_state σ j n (Until$_S$ p s_1 I s_2 b ts aux) (Until α' I β)

Fig. 10. Main invariant predicates (excerpt)

ated up to j. For Until α I β, we take the least index i at which we cannot evaluate the operator yet. Recall that these are the indices for which we do not have complete information up to and including the time $\tau\ \sigma\ i +$ right I. The index k in the definition ranges over all indices for which the time-stamp (condition $k < j$) and the results from both subformulas (condition min (prog σ α j) (prog σ β j)) are available. All other operators are only constrained by the progress of their subformula(s). We note some basic properties of prog.

Lemma 1. *(a) Monotonicity:* $j \leq j'$ *implies* prog σ φ $j \leq$ prog σ φ j'. *(b) Upper bound:* prog σ φ $j \leq j$. *(c) Completeness:* mf φ *implies* $\exists j.\ i \leq$ prog σ φ j, *for all* i.

The predicate wf_mstate φ π *mst* (Fig. 10) is the invariant for a monitor state *mst* after monitoring φ on the stream prefix π. We require that π is a well-formed prefix and that the cached degree n agrees with the formula. All auxiliary invariants that are used to define wf_mstate are expressed in terms of infinite streams instead of prefixes. This includes wf_state σ j n s φ, which holds iff s :: *state* corresponds to the monitorable formula φ :: *frm* after monitoring the first j databases of σ. Therefore, we consider all event streams σ that extend the prefix π, i.e., prefix_of π σ.

We only show the case for Until$_S$ in wf_state here. This case states the conditions under which Until$_S$ p s_1 I s_2 b ts aux is a well-formed state corresponding to Until α' I β. Depending on the flag p, α' is either a negated subformula $\alpha' =$ Neg α for some α, or its outermost operator is not a negation and $\alpha' = \alpha$. The invariant inherits the condition fv $\alpha \subseteq$ fv β from the mf predicate (Sect. 5). The two subformula states m_1 and m_2 must be recursively well-formed and correspond to α and β, respectively.

The predicate wf_buf (Fig. 11) encodes the invariant for buffers of type *buf*. These store all results of the formulas α and β from index prog2 σ α β j to indices prog σ α j and prog σ β j, respectively. The results must be tables assigning values to the free variables of the corresponding formula. The invariant wf_ts ensures that the additional time-stamp list *ts*, which is used by binary temporal operators, contains all time-stamps

definition wf_buf :: *trace* ⇒ *nat* ⇒ *nat* ⇒ *frm* ⇒ *frm* ⇒ *buf* ⇒ *bool* **where**
wf_buf $\sigma\ j\ n\ \alpha\ \beta\ b \longleftrightarrow$ (case b of (xs, ys) ⇒
list_all2 $(\lambda k.$ wf_table n (fv α) $(\lambda v.$ sat $\sigma\ \overline{v}\ k\ \alpha))$ [prog2 $\sigma\ \alpha\ \beta\ j$ ..< prog $\sigma\ \alpha\ j$] $xs \wedge$
list_all2 $(\lambda k.$ wf_table n (fv β) $(\lambda v.$ sat $\sigma\ \overline{v}\ k\ \beta))$ [prog2 $\sigma\ \alpha\ \beta\ j$ ..< prog $\sigma\ \beta\ j$] ys)

definition wf_ts :: *trace* ⇒ *nat* ⇒ *frm* ⇒ *frm* ⇒ *ts list* ⇒ *bool* **where**
wf_nts $\sigma\ j\ \alpha\ \beta\ ts \longleftrightarrow$ list_all2 $(\lambda k\ t.\ t = \tau\ \sigma\ k)$ [prog2 $\sigma\ \alpha\ \beta\ j$ ..< j] ts

definition wf_uaux :: *trace* ⇒ *nat* ⇒ *nat* ⇒ *bool* ⇒ *frm* ⇒ $\mathcal{I}$ ⇒ *frm* ⇒ *uaux* ⇒ *bool* **where**
wf_uaux $\sigma\ j\ n\ p\ \alpha\ I\ \beta\ aux \longleftrightarrow$ prog σ (Until $\alpha\ I\ \beta$) j + length aux = prog2 $\sigma\ \alpha\ \beta\ j \wedge$
list_all2 $(\lambda x\ k.$ case x of $(t, r_1, r_2) \Rightarrow t = \tau\ \sigma\ k \wedge$
wf_table n (fv α) $(\lambda v.$ if p then $(\forall k' \in \{k$ ..< prog2 $\sigma\ \alpha\ \beta\ j\}.$ sat $\sigma\ \overline{v}\ k'\ \alpha)$
else $(\exists k' \in \{k$ ..< prog2 $\sigma\ \alpha\ \beta\ j\}.$ sat $\sigma\ \overline{v}\ k'\ \alpha)) \wedge$
wf_table n (fv β) $(\lambda v.\ \exists k'.\ k \le k' \wedge k' <$ prog2 $\sigma\ \alpha\ \beta\ j \wedge \tau\ \sigma\ k' - \tau\ \sigma\ k \in_{\mathcal{I}} I \wedge$
sat $\sigma\ \overline{v}\ k'\ \beta \wedge (\forall k'' \in \{k$ ..< $k'\}.$ if p then sat $\sigma\ \overline{v}\ k''\ \alpha$ else ¬sat $\sigma\ \overline{v}\ k''\ \alpha)))$
aux [prog σ (Until $\alpha\ I\ \beta$) j ..< prog2 $\sigma\ \alpha\ \beta\ j$])

Fig. 11. Invariants of the state's components

from the start of the result buffer to the most recent input, which has index j.

The invariant for the auxiliary states *aux* of type *uaux* for the $\psi =$ Until $\alpha'\ I\ \beta$ operator is shown in Fig. 11. The elements of *aux* are in a one-to-one correspondence with the indices in $[i_\psi$ ..< $i^*]$, where $i_\psi =$ prog $\sigma\ \psi\ j$ and $i^* =$ prog2 $\sigma\ \alpha'\ \beta\ j =$ prog2 $\sigma\ \alpha\ \beta\ j$ (the possible negation in α' does not affect the progress). Each element for such an index k with time-stamp t is a triple (t, r_1, r_2). The content of the tables r_1 and r_2 is described in Sect. 5.

We state the correctness of the satisfactions output by step (and hence monitor) in terms of a function verdicts, which characterizes the monitor's output semantically. For a formula φ and stream prefix π, it returns exactly the pairs (k, v) where the monitor has made progress beyond k, and for which sat $\sigma\ \overline{v}\ i\ \varphi$ is true for all traces σ that extend π.

definition verdicts :: *frm* ⇒ $(db \times ts)$ *list* ⇒ $(nat \times tuple)$ *set* **where**
verdicts $\varphi\ \pi = \{(k, v).$ wf_tuple (nfv φ) (fv φ) $v \wedge (\forall \sigma.$ prefix_of $\pi\ \sigma \longrightarrow$
$k <$ prog $\sigma\ \varphi$ (length π) $\wedge$ sat $\sigma\ \overline{v}\ k\ \varphi)\}$

Using the completeness of prog, we show that verdicts behaves according to the informal description of a monitor, which we gave in the beginning of Sect. 5.

Lemma 2. *For all monitorable formulas* φ, verdicts φ *is sound and eventually complete, i.e., for all prefixes* π *extending the stream* σ*, indices* k*, and tuples* v*,*

(a) $(k, v) \in$ verdicts $\varphi\ \pi \longrightarrow$ sat $\sigma\ \overline{v}\ k\ \varphi$*, and*
(b) $k <$ length $\pi \wedge$ wf_tuple (nfv φ) (fv φ) $v \wedge (\forall \sigma'.$ prefix_of $\pi\ \sigma' \longrightarrow$ sat $\sigma'\ \overline{v}\ k\ \varphi) \longrightarrow$
$(\exists \pi'.$ prefix_of $\pi'\ \sigma \wedge (k, v) \in$ verdicts $\varphi\ \pi')$.

We can now state the main correctness result for the more general online interface consisting of init and step. The correctness of monitor follows easily. Let last_ts π denote the last time-stamp of π, and 0 if π is empty.

Theorem 1. *(a)* init *establishes the invariant:* mf φ *implies* wf_mstate φ [] (init φ).
(b) step *preserves the invariant and its output can be described in terms of* verdicts*: Let* step (db, t) $mst = (A, mst')$*. If* wf_mstate $\varphi\ \pi$ mst *and* last_ts $\pi \le t$*, then we have* $A =$ verdicts φ $(\pi$ @ $[(db, t)]) -$ verdicts $\varphi\ \pi$ *and* wf_mstate φ $(\pi$ @ $[(db, t)])$ mst'.

Corollary 1. *If* mf φ *and* wf_prefix π, monitor $\varphi\ \pi =$ verdicts $\varphi\ \pi$.

7 Case Study in Differential Testing

To demonstrate the benefit of our verified monitor we perform differential testing [17] to compare our monitor to two existing unverified state-of the-art monitors, MonPoly [3] and DejaVu [13], which support first-order temporal logic specifications.

We used Isabelle/HOL's code generator [9] to export a certified implementation of our monitoring algorithm (called *VeriMon*) for the monitorable fragment of MFOTL. The generated file consists of about 2 800 lines of OCaml code and includes code generated from an Isabelle library of red-black trees, which are used to efficiently implement sets. To be used as a standalone monitor, the verified monitor must be augmented with a formula and log parser. We reused MonPoly's parsing components, as they were implemented in OCaml and extensively used and tested. About 130 lines of straightforward, unverified OCaml code integrates these unverified components with the verified algorithm, translating between the analogous types for formulas and traces.

We focus on randomized differential testing. We generate random stream prefixes and formulas, invoke the monitors, and validate the results using VeriMon. For this purpose, we have developed a random MFOTL formula generator that takes as parameters the formula size (in terms of number of operators) and the number of free variables that occur in the formula, and outputs a random formula and a signature describing the name, arity, and parameter types for each predicate used in the formula. The generator creates a random formula of size n by randomly selecting an operator op and then recursively creating its subformula of size $n-1$ (if op is unary) or its two subformulas of size m and $n-m-1$ (if op is binary) for some random non-negative $m < n$. The generator creates predicate or equality formulas for size $n = 0$. Since each monitor can be tested on the logical fragment it mutually supports with VeriMon, our formula generator only generates monitorable MFOTL formulas for testing MonPoly and monitorable, past-only, non-metric formulas for testing DejaVu. Monitorable formulas are generated by sampling only the operators that correspond to the cases in the definition of the recursive predicate mf (Fig. 4). Whenever an operator op is sampled, free variables for its subformulas are sampled to satisfy mf's conditions for op. DejaVu requires the generator to sample only past temporal operators, use only $\mathsf{interval}\ 0\ \infty$ in temporal formulas, and since it does not support free variables, all generated formulas are closed. DejaVu can only monitor traces with databases containing a single event, which results in formulas like $P \wedge Q$ (i.e., $\mathsf{And}\ (\mathsf{Pred}\ \mathtt{P}\ [])\ (\mathsf{Pred}\ \mathtt{Q}\ [])$) evaluating to false. The generator avoids this by ensuring that binary Boolean formulas have at least one temporal subformula referring to the past.

The generated signature file is used by a random stream prefix generator to sample random event names defined in the file. For each event, the generator uniformly samples its parameter values from the domain $D = \{0, 1, \ldots, 10^9 - 1\}$. With a given probability r, the last q unique values that were previously sampled are sampled again to ensure that events have common parameter values. This makes the subsequent monitoring less trivial.

DejaVu's output differs from MonPoly's and VeriMon's. DejaVu does not output variable valuations that violate the formula, but only the prefix indices where the formula is violated. We use these indices as the basis for comparing its output with VeriMon.

We ran our testing suite for formula sizes ranging from 2 to 5, having up to 6 free variables. For each combination of these parameters, we generated 1 000 random formulas and for each formula 4 random prefixes with lengths of 20, 40, 60, and 100 databases.

Our results reveal two classes of inconsistencies in MonPoly's output and three in DejaVu's output. The inconsistencies in MonPoly's output correspond to two implementation errors. The first error manifests in MonPoly's handling of finite trace semantics. Specifically, after reading the entire stream prefix MonPoly outputs an additional violation for a non-existing index (beyond the last index present in the prefix). MonPoly's second implementation error was exhibited by its failure to correctly monitor a formula of the form $\alpha \wedge \neg(\beta \,\mathsf{S}\, \alpha)$ (i.e., $\mathsf{AndNot}\ \alpha\ (\mathsf{Since}\ \beta\ (\mathsf{interval}\ 0\ \infty)\ \alpha)$) where $\mathsf{nfv}\ \beta > 0$, $\mathsf{fv}\ \beta \subset \mathsf{fv}\ \alpha$, and the order of occurrences of free variables in the two instances of α is different. These conditions trigger a heavily optimized part of MonPoly's code, confirming our intuition that complex performance optimizations can lead to implementation errors.

The problems exhibited by DejaVu's implementation are arguably less severe and all related to monitoring formulas with equalities. The most benign issue is that formulas containing only arithmetic relations (and no predicates) fail to parse. Next, we discovered that DejaVu does not produce any violation on a prefix satisfying a propositional formula α, when monitoring a formula of the form $\neg\exists x.\ \alpha \wedge x = 24$ (i.e., $\mathsf{Neg}\ (\mathsf{Exists}\ (\mathsf{And}\ \alpha\ (\mathsf{Eq}\ (\mathsf{V}\ 0)\ (\mathsf{C}\ 24)))))$). DejaVu's authors documented that the formula semantics changes if a variable occurs in arithmetic relations [11, §5]. Specifically, the variable's quantifier becomes bounded: it quantifies only over the active domain defined as values seen in the prefix so far. The change has an (unintuitive) effect on the subformulas where the variable does not occur as shown in the example above. Finally, DejaVu does not output any violation for the formula $\neg\exists x.\ x = 24 \wedge \neg P(x)$ (i.e., $\mathsf{Neg}\ (\mathsf{Exists}\ (\mathsf{AndNot}\ (\mathsf{Eq}\ (\mathsf{V}\ 0)\ (\mathsf{C}\ 24))(\mathsf{Pred}\ \mathtt{P}\ [\mathsf{V}\ 0]))))$ when monitored on a prefix without the event $(\mathtt{P}, [24])$. This formula's violations coincide under both standard and active domain quantifier semantics. However, DejaVu's definition of the active domain does not include the constants occurring in the formula, which causes the discrepancy.

In addition to using random formulas, we included the tool's benchmarks in our testing. All the experiments are available in an easy to reproduce Docker image [28].

8 Conclusion

We demonstrated an approach to increase the trustworthiness of runtime verification by formally verifying a monitor for MFOTL in the Isabelle/HOL proof assistant. Our formalization of the non-trivial monitoring algorithm is essentially a high-level implementation as one would write it in a functional programming language. To prove its correctness, we had to characterize the algorithm's output, which precisely documents its behavior. Being able to execute a verified monitor with acceptable performance enables the systematic testing of more performant implementations. Our results from differential testing, which uncovered two genuine errors in MonPoly, show that this is beneficial.

One possible use case of our verified monitor is as a referee in tool competitions, where it can provide the ground truth. We also believe that it is a good starting point for extensions of the monitoring algorithm, whose correctness may not be obvious, as in our unpublished draft on adaptive monitoring [27]. Other future extensions may include the use of more optimized and verified data structures, which would make the generated code even more efficient. Finally, we hope that our compact formalization encourages machine-checked proofs for other algorithms and tools.

Acknowledgment. Joshua Schneider is supported by the US Air Force grant "Monitoring at Any Cost" (FA9550-17-1-0306). Srđan Krstić is supported by the Swiss National Science Foundation grant "Big Data Monitoring" (167162). Martin Raszyk pointed us to DejaVu's non-standard semantics for formulas with equality. Anonymous reviewers gave numerous helpful suggestions on how to improve the presentation.

References

1. Ausaf, F., Dyckhoff, R., Urban, C.: POSIX lexing with derivatives of regular expressions (proof pearl). In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 69–86. Springer (2016)
2. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
3. Basin, D., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017)
4. Benzaken, V., Contejean, E.: A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In: Mahboubi, A., Myreen, M.O. (eds.) CPP 2019. pp. 249–261. ACM (2019)
5. Benzaken, V., Contejean, E., Keller, C., Martins, E.: A Coq formalisation of SQL's execution engines. In: Avigad, J., Mahboubi, A. (eds.) ITP 2018. LNCS, vol. 10895, pp. 88–107. Springer (2018)
6. Blech, J.O., Falcone, Y., Becker, K.: Towards certified runtime verification. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 494–509. Springer (2012)
7. Bohrer, B., Tan, Y.K., Mitsch, S., Myreen, M.O., Platzer, A.: Veriphy: verified controller executables from verified cyber-physical system models. In: Foster, J.S., Grossman, D. (eds.) PLDI 2018. pp. 617–630. ACM (2018)
8. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer (2013)
9. Haftmann, F.: Code generation from specifications in higher-order logic. Ph.D. thesis, Technical University Munich (2009)
10. Havelund, K.: Rule-based runtime verification revisited. STTT **17**(2), 143–170 (2015)
11. Havelund, K., Peled, D.: Efficient runtime verification of first-order temporal properties. In: Gallardo, M., Merino, P. (eds.) Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10869, pp. 26–47. Springer (2018)
12. Havelund, K., Peled, D., Ulus, D.: First order temporal logic monitoring with BDDs. In: FMCAD 2017. pp. 116–123. IEEE (2017)
13. Havelund, K., Peled, D., Ulus, D.: Dejavu: A monitoring tool for first-order temporal logic. In: MT@CPSWeek 2018. pp. 12–13 (2018)
14. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Lectures on Runtime Verification, LNCS, vol. 10457, pp. 61–102. Springer (2018)
15. Laurent, J., Goodloe, A., Pike, L.: Assuring the guardians. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 87–101. Springer (2015)
16. Malecha, J.G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL 2010. pp. 237–248. ACM (2010)
17. McKeeman, W.M.: Differential testing for software. Digital Technical Journal **10**(1), 100–107 (1998)

18. Mitsch, S., Platzer, A.: Modelplex: verified runtime validation of verified cyber-physical system models. Formal Methods in System Design **49**(1-2), 33–74 (2016)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. Springer (2002)
20. Nipkow, T., Traytel, D.: Unified decision procedures for regular expression equivalence. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 450–466. Springer (2014)
21. Pike, L., Niller, S., Wegmann, N.: Runtime verification for ultra-critical systems. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 310–324. Springer (2011)
22. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Experience report: a do-it-yourself high-assurance compiler. In: Thiemann, P., Findler, R.B. (eds.) ICFP 2012. pp. 335–340. ACM (2012)
23. Reger, G., Rydeheard, D.E.: From first-order temporal logic to parametric trace slicing. In: RV 2015. LNCS, vol. 9333, pp. 216–232. Springer (2015)
24. Rizaldi, A., Keinholz, J., Huber, M., Feldle, J., Immler, F., Althoff, M., Hilgendorf, E., Nipkow, T.: Formalising and monitoring traffic rules for autonomous vehicles in Isabelle/HOL. In: Polikarpova, N., Schneider, S. (eds.) iFM 2017. LNCS, vol. 10510, pp. 50–66. Springer (2017)
25. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Log. Methods Comput. Sci. **8**(1) (2012)
26. 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)
27. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Adaptive online first-order monitoring. In: Chen, Y.F., Cheng, C.H., Esparza, J. (eds.) ATVA 2019. Springer (2019), `http://people.inf.ethz.ch/trayteld/papers/atva19-adaptive/aom.pdf`, to appear.
28. Schneider, J., Basin, D., Krstić, S., Traytel, D.: Case study associated with this paper. `https://hub.docker.com/r/infsec/verified-monpoly-exps` (2019), Docker image (tag 1.3.0)
29. Schneider, J., Traytel, D.: Formalization of a monitoring algorithm for metric first-order temporal logic. Archive of Formal Proofs (2019), `http://isa-afp.org/entries/MFOTL_Monitor.html`
30. Völlinger, K.: Verifying the output of a distributed algorithm using certification. In: Lahiri, S.K., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 424–430. Springer (2017)
31. Wimmer, S.: Formalized timed automata. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 425–440. Springer (2016)
32. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 61–78. Springer (2018)