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 (dL) [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. Vollinger [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, …, Booleans bool, natural numbers nat, sets ‘a set, pairs ‘a × ‘b, and functions ‘a ⇒ ‘b. Functions are usually curried, and ⇒ 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 = [ ] | Cons ‘a (‘a list) defines finite lists. The Cons constructor is usually written infix as #.

Terms are built from variables x, y, …, constants, function applications f x, and abstractions λx. z. Function application is left-associative. We use additional notation, e.g., conditionals if b then z₁ else z₂, case distinctions for datatypes case d of x # xs ⇒ z | …, and infix operators. The expression z :: t denotes that the term z has type t. The command definition c :: t where 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 map :: (‘a ⇒ ‘b) ⇒ ‘a list ⇒ ‘b list where
map f [ ] = [ ] | map f (x # xs) = f x # map f xs

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

inductive list_all2 :: (‘a ⇒ ‘b ⇒ bool) ⇒ ‘a list ⇒ ‘b list ⇒ bool where
list_all2 P [] [] | P x y ∧ list_all2 P xs ys → 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, 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:: ‘a. P x} contains all x satisfying P. Other set operations are A × B (Cartesian product), A − B (set difference), ∪ x∈A. F x (indexed union, i.e., {y. ∃x∈A. y∈F x}), Inf A (infimum), and f ‘A (image of A under f, i.e., ∪ x∈A. {f x}). The sets {a..<b} and {a..<b} contain all natural numbers n with a ≤ n < b and a < n ≤ b, respectively. The list [a..<b] contains all of {a..<b} in ascending order. The datatype ‘a option has two constructors ⊥ and ⟨x:: ‘a⟩. The term map_option f maps ⊥ to ⊥ and ⟨x⟩ to ⟨f x⟩. Options can be converted to sets via ⊥ = {} and ⟨[x]⟩ = {x}. The function these :: ‘a option set ⇒ ‘a set maps A to ∪ x∈A. [x]. The term foldr f xs z combines the elements of the list xs with the binary function f, using z as the initial value, e.g., foldr (−) [1, 2] 3 = 1 − (2 − 3). The set of all elements in the list xs is set xs, the length of xs is length xs, the i-th element of xs is xs ! i (zero-based, requires i < length xs), and list concatenation is xs @ ys. Strings string are character lists. Streams ‘a stream are infinite sequences of values of type ‘a.
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 \times 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 \( \Gamma \sigma i \) for the \( i \)-th database in event stream \( \sigma \) and \( \tau \sigma i \) for the corresponding time-stamp, where \( i \) is a zero-based index. The predicate \( \text{wf\_trace} \) in trace's definition ensures that the time-stamp sequence is monotonic (\( \forall i. \tau \sigma i \leq \tau \sigma (i + 1) \)) and unbounded (\( \forall i. \exists j. \tau \sigma i < \tau \sigma j \)). A (stream) prefix \( \pi \) is a finite list of time-stamped databases. It satisfies \( \text{wf\_prefix} \) iff the prefix \( \pi \) has monotonic time-stamps. We write prefix_of \( \pi \sigma i \) if the event stream \( \sigma \) extends the prefix \( \pi \), i.e., the sequence of \( \sigma \)'s first length \( \pi \) elements equals \( \pi \).

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

We have sat \( \sigma v i \varphi \) iff the formula \( \varphi \) is satisfied by the valuation \( v \) at index \( i \), given the event stream \( \sigma \). 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.

### 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 \times 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 \( \Gamma \sigma i \) for the \( i \)-th database in event stream \( \sigma \) and \( \tau \sigma i \) for the corresponding time-stamp, where \( i \) is a zero-based index. The predicate \( \text{wf\_trace} \) in trace's definition ensures that the time-stamp sequence is monotonic (\( \forall i. \tau \sigma i \leq \tau \sigma (i + 1) \)) and unbounded (\( \forall i. \exists j. \tau \sigma i < \tau \sigma j \)). A (stream) prefix \( \pi \) is a finite list of time-stamped databases. It satisfies \( \text{wf\_prefix} \) iff the prefix \( \pi \) has monotonic time-stamps. We write prefix_of \( \pi \sigma i \) if the event stream \( \sigma \) extends the prefix \( \pi \), i.e., the sequence of \( \sigma \)'s first length \( \pi \) elements equals \( \pi \).

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

We have sat \( \sigma v i \varphi \) iff the formula \( \varphi \) is satisfied by the valuation \( v \) at index \( i \), given the event stream \( \sigma \). 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 (x, y) = (if x = y then map_option (λz. x) (join1 (x, y)) else ⊥)

fun 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 are lists of optional domain values, which assign values only to those variables i whose corresponding entries xs!i are not ⊥. 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 ϕ ≤ n to express that v is a well-formed tuple for ϕ.

We obtain the corresponding valuation τ = map the v, where the function the maps (x) to x and ⊥ to some unspecified domain element. The actual value assigned to ⊥ 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) σ (i, sat σ τ i ϕ) A, where nfv ϕ ≤ n. The abbreviation table n V A = wf_table n V (λv. v ∈ 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 ∪ B, satisfying table n V (A ∪ 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 ⊥). The function join1 returns an optional tuple, where ⊥ 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
A monitor takes an MFOTL formula $\varphi$ and an event stream $\sigma$ as inputs. It computes satisfactions: pairs $(i, v)$ of indices $i$ and valuations $v$ that satisfy the formula, i.e., $\text{sat} \sigma \models i \varphi$. One is often interested in finding the violations of a formula $\varphi$, which are the pairs $(i, v)$ such that $\neg \text{sat} \sigma \models i \varphi$. 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 monitor $\varphi \pi = \text{steps} \pi (\text{init} \varphi)$, where steps iterates step on a prefix and collects all satisfactions in a set.

Example 1. Consider the formula $A(x) \rightarrow \diamond_{[1,2]} (\exists y. B(x, y))$, 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 $A(x) \land \neg \diamond_{[1,2]} (\exists y. B(x, y))$, which we encode as...
\( \varphi_{ex} = \text{AndNot} (\text{Pred A} [V 0]) \) (Eventually (interval 1 2) (Exists (\text{Pred B} [V 1, V 0])))

Given the prefix \( \pi_{ex} = \{ \{(A, [d]), (A, [e])\}, 1\}, \{(B, [d, f])\}, 2\}, \{(B, [e, f])\}, 5\} \), 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: monitor \( \varphi_{ex} \). The satisfaction originates from the event \( (A, [e]) \), which is part of the database with index 0 in \( \pi_{ex} \). The satisfaction’s valuation is \( (\{e\}) \) because the parameter of \( (A, [e]) \) is bound to \( \varphi_{ex} \)’s first (and only) free variable. The satisfaction is output after processing the third database:

\[
\begin{align*}
\text{step} (\{(A, [d]), (A, [e])\}, 1) & (\text{init } \varphi_{ex}) = (\{\}, s_1) \\
\text{step} (\{(B, [d, f])\}, 2) & s_1 = (\{\}, s_2) \\
\text{step} (\{(B, [e, f])\}, 5) & s_2 = (\{(0, (\{e\}))\}, s_3),
\end{align*}
\]

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 \( (A, [d]) \), we do not know whether \( (j, [d]) \) satisfies \( \varphi_{ex} \) 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 \( \psi \) at index \( i \) yields a table containing all valuations \( v \) with \( \text{sat } \sigma \vdash_1 \psi \). 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 \( \psi \) has its own progress \( i_\psi \), 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 \( x_3 :: \text{table list} \). Its elements correspond to indices \( [i .. i + \text{length } x_3] \) 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 \( \neg \text{A} (x) \) (i.e., Neg (Pred A [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 \( \varphi \) must be monitorable if we search for violations of \( \varphi \), which is generally different from the monitorability of \( \varphi \) itself. Basin et al. [2] describe a heuristic that attempts to rewrite formulas into equivalent, monitorable ones.

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

The monitor’s state $\text{MState } n \ i \ s$ consists of the formula’s degree $n = \text{nfv } \varphi$ (to avoid re computation), the progress $i$, and a formula state $s$. The formula state datatype $\text{state}$ (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 $\text{state}$.

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 $\text{mf}$’s definition. Some of $\text{state}$’s constructors carry a Boolean flag $p$ that indicates whether one of the subformulas is positive ($p = \text{True}$) or negated ($p = \text{False}$). 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, $\alpha \land \neg \beta$ (i.e., $\text{AndNot } \alpha \beta$) is converted to $\text{And}_5 (\text{init0 } n \alpha) \text{False } (\text{init0 } n \beta) ([], []).$ All lists in the $\text{state}$ 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 $\text{state}$. 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 $i$.
\textbf{fun} \textit{match :: trm list \rightarrow domain list \rightarrow \langle\\langle nat \Rightarrow domain option\rangle\\rangle option where}
\textit{match \[\{x. \perp\] = \langle\{x \mapsto \perp\\}\rangle}
| \textit{match \langle C \times \mathcal{T}\rangle \{y \mapsto \mathcal{Y}\} \Rightarrow (if x = y \text{ then } \text{match} \mathcal{T} \mathcal{Y} \text{ else } \perp)}
| \textit{match \langle V \times \mathcal{T}\rangle \{y \mapsto \mathcal{Y}\} \Rightarrow (\text{case match} \mathcal{T} \mathcal{Y} \text{ of } \perp \Rightarrow \perp)}
| (f) \Rightarrow (\text{case } f \times \mathcal{X} \Rightarrow (f(x \mapsto y)) \mid (z) \Rightarrow (if y = z \text{ then } (f) \text{ else } \perp))
| \textit{match \_\_\_\_ = \perp}

\textbf{fun} \textit{buf_add :: table list \rightarrow table list \Rightarrow buf \Rightarrow buf where}
\textit{buf_add x' \_ \_y' (x, y) \Rightarrow (x @ x', y @ y')}

\textbf{fun} \textit{buf_take :: (table \Rightarrow table \Rightarrow 'b) \Rightarrow buf \Rightarrow 'b list \times buf}
\textit{buf_take f \langle x \# xx, y \# yx\rangle = (\text{let } (z, b) = \text{buf_take} f \langle xx, yx\rangle \text{ in } (f x \# z, b)\rangle}
| \textit{buf_take f \langle xx, yx\rangle = (\langle\langle\rangle, (xx, yx)\rangle)

\textbf{Fig. 6.} The match function

\textbf{Fig. 7.} Buffer operations

\textit{Atomic Formulas.} The constructor $\mathsf{Eq}_S$ of state represents constant tables corresponding to (monitorable) equalities. The associated $\mathit{state}$ is always the same table, which is returned upon evaluation. For a predicate’s state $\mathsf{Pred}_S \ e \ \mathcal{T}$, we first select all events in the database $\mathit{db}$ with the name $e$. The auxiliary function $\text{match}$, defined in Fig. 6, is applied to each selected event. This function attempts to compute the unique valuation for the variables in $\mathcal{T}$ that makes the terms match the event’s parameters. It returns $(f)$ if such a valuation $f$ exists, and $\perp$ otherwise. To simplify $\text{match}$’s definition, $f$ is encoded as a partial function $\text{nat} \Rightarrow \text{domain} \ \mathit{option}$. We convert it into a tuple using map $f \ [0..<n]$.

\begin{align*}
\text{eval } n \ t \ \mathit{db} (\mathsf{Eq}_S \ r) &= ([r], \mathsf{Eq}_S \ r) \\
\text{eval } n \ t \ \mathit{db} (\mathsf{Pred}_S \ e \ \mathcal{T}) &= ([\lambda f. \ \text{map} \ f \ [0..<n]) \ \text{these} \\
& (\text{match} \ \mathcal{T} \ \mathsf{\&} \ (\bigcup \{e', x \in \mathit{db}. \ \text{if } e = e' \ \text{then } \{x\} \ \text{else } \{\}\})], \mathsf{Pred}_S \ e \ \mathcal{T})
\end{align*}

\textit{Non-Temporal Operators.} It may be possible to evaluate the two subformulas $\alpha$ and $\beta$ 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 $i_\alpha$ and $i_\beta$. 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 $\mathit{buf}$, 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 $\text{buf_add } xx' \ yx' \ b$ (Fig. 7) adds the result lists $xx'$ and $yx'$ from the two subformulas to the buffer $b$. The function $\text{buf_take } f \ b$ 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 $xx$ and $yx$, as well as the updated subformula states $s_1'$ and $s_2'$. The results are added to the buffer $b$. Then, $\text{buf_take}$ 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.

\begin{align*}
\text{eval } n \ t \ \mathit{db} (\mathsf{And}_S \ s_1 \ p \ s_2 \ b) &= (\text{let } (xx, s'_1) = \text{eval } n \ t \ \mathit{db} \ s_1; (yx, s'_2) = \text{eval } n \ t \ \mathit{db} \ s_2 \ (zs, b) = \text{buf_take} (\text{join } p) \ (\text{buf_add } xx \ yx \ b) \ (\text{in } (zs, \mathsf{And}_S \ s'_1 \ p \ s'_2 \ b)) \\
\text{eval } n \ t \ \mathit{db} (\mathsf{Or}_S \ s_1 \ s_2 \ b) &= (\text{let } (xx, s'_1) = \text{eval } n \ t \ \mathit{db} \ s_1; (ys, s'_2) = \text{eval } n \ t \ \mathit{db} \ s_2 \ (zs, b) = \text{buf_take} (\cup) \ (\text{buf_add } xx \ yx \ b) \ (\text{in } (zs, \mathsf{Or}_S \ s'_1 \ s'_2 \ b))
\end{align*}
**definition** update_since :: \( I \Rightarrow bool \Rightarrow table \Rightarrow ts \Rightarrow aux \Rightarrow table \times aux \) where

update_since \( I \, p \, r_1 \, r_2 \, nt \, aux \) = (let aux = (case \((t, \, \text{join} \, r \, p \, r_1, \, t, \, r) \leftarrow aux, \, nt - t \leq \text{right} \, I \) of 
\[ [] \rightarrow ([nt, r_2]) \]
\[ x \# aux' \rightarrow \begin{cases} \text{if} \, \text{fst} \, x = nt \rightarrow (\text{fst} \, x, \, \text{snd} \, x \cup r_2) \# aux' \, \text{else} \, (nt, r_2) \# x \# aux') \end{cases} \]
\[ \text{in} \, \text{foldr} \,(\cup) \,(r, \, \text{t}, \, \text{r}, \, nt) \leftarrow aux, \, \text{left} \, I \leq nt - t \, \{ [], \, aux \} \) )

**fun** update_until :: \( I \Rightarrow bool \Rightarrow table \Rightarrow ts \Rightarrow aux \Rightarrow aux \) where

update_until \( I \, p \, r_1 \, r_2 \, nt \, aux \) = (map (\((x, \, \text{case} \, x \, \text{of} \, (t, \, a_1, \, a_2) \rightarrow \begin{cases} (t, \, \text{if} \, p \rightarrow \text{join} \, a_1 \, \text{True} \, r_1 \, \text{else} \, a_1 \, \cup \, r_1, \, \text{if} \, nt - t \leq \text{right} \, I \rightarrow a_2 \, \cup \, \text{join} \, r_2 \, p \, a_1 \, \text{else} \, a_2) \, aux) \end{cases} \)) \) \)

**fun** eval_until :: \( I \Rightarrow ts \Rightarrow aux \Rightarrow table \, \text{list} \times aux \)

\[
\text{eval_until} \, I \, nt \, [] = ([], []) \\
\mid \text{eval_until} \, I \, nt \, ((t, \, a_1, \, a_2) \# aux) = \begin{cases} \text{if} \, t + \text{right} \, I < nt \rightarrow \text{then} \, (let \,(xx, \, aux) = \text{eval_until} \, I \, nt \, \text{aux} \, \text{in} \,(a_2 \# xx, \, aux)) \, \text{else} \, ([], (t, \, a_1, \, a_2) \# aux) \end{cases} \\
\]

**fun** tbuf_take :: \( (table \Rightarrow \text{table} \Rightarrow ts \Rightarrow \text{b} \Rightarrow \text{b} \Rightarrow \text{t} \Rightarrow \text{ts} \Rightarrow \text{b} \times \text{b} \times \text{ts} \Rightarrow \text{list} \) where

tbuf_take \( f \) \( x, \, y, \, t, \, z \) \( (x, \, y) \times (t, \, s) = \) \( \text{tbuf}_{-}\, \text{take} \) \( f \, (x, \, y, \, t, \, z) \times (xs, \, ys) \, ts \) \mid \text{tbuf}_{-}\, \text{take} \) \( f \, (x, \, y, \, t, \, z) \times (xs, \, ys) \, ts = \) \( (z, \, (xs, \, ys) \, ts) \)

**Fig. 8.** Auxiliary operations for evaluating \( \text{Since}_S \) and \( \text{Until}_S \)

We increment the degree in the recursive computation of an existential quantifier \( \text{Exists}_S \) to account for the variables’ De Bruijn encoding. Each computed tuple \( v \) (which is a list) is then replaced by its tail \( t \) \( v \) to remove the assignment to the bound variable.

\[
\mid \text{eval} \, n \, t \, db \, (\text{Exists}_S \, s) = (let \,(xs, \, s') = \text{eval} \, (n + 1) \, t \, db \, s \in (\text{map} \,(\lambda \, t \rightarrow r) \, \text{xs}, \, \text{Exists}_S \, s'))
\]

**Since and Until.** The monitoring algorithm implements \( \text{Since} \, \alpha \, I \beta \) by decomposing the interval \( I \). Note that \( \text{Since} \, \alpha \, I \beta \) is equivalent to a disjunction of \( \text{Since} \, \alpha \, (\text{point} \, c) \, \beta \), 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 \( \text{right} \, I = \infty \)). We store the satisfactions for \( \text{Since} \, \alpha \, (\text{point} \, c) \, \beta \) 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 \( c < \text{left} \, I \) (if \( \text{left} \, I > 0 \)) because these may move into the interval \( I \) as time progresses.

For every new input database, the function \( \text{update}_{-}\, \text{since} \) (Fig. 8) updates the list to maintain the correspondence with \( \text{Since} \, \alpha \, (\text{point} \, c) \, \beta \). It also computes the satisfactions by taking the union over all tables in the list that satisfy \( c \in I \). The arguments of \( \text{update}_{-}\, \text{since} \) 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 \( \text{Since} \, \alpha \, (\text{point} \, (c + A)) \, \beta \), but without taking the satisfactions of \( \alpha \) and \( \beta \) at \( i \) into account. First, \( \text{update}_{-}\, \text{since} \) removes all tables for which \( c + A \) exceeds the right bound of the interval. It then joins each remaining table with the result \( r_1 \) for \( \alpha \), and adds the satisfactions \( r_2 \) for \( \beta \) either to the first table (if \( c + A = 0 \)) or as a new list element to the list.

Decomposing a formula \( \psi = \text{Until} \, \alpha \, I \beta \) into point intervals is not as useful because there is no obvious way to compute the satisfactions of \( \text{Until} \, \alpha \, (\text{point} \, c) \, \beta \) 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 \( i_0 \) and \( i_j \). The auxiliary state for Until, which has type \( uaux \), stores for all \( k \) in \( \{ i_0..<i^* \} \) 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 \( \alpha \) is positive. If \( p = \text{True} \), the table \( a_1 \) contains the valuations satisfying \( \alpha \) at all indices in \( \{ k..<i^* \} \). If \( p = \text{False} \), it contains the valuations satisfying \( \neg \alpha \) at some index in \( \{ k..<i^* \} \). The table \( a_2 \) contains the valuation satisfying \( \beta \) at some index \( k' \) in \( \{ k..<i^* \} \) with \( \tau \sigma k' - \tau \sigma k \in \tau I \), and satisfying \( \alpha \) for all indices in \( \{ k..<k' \} \). Note that this is not the same as the satisfactions for Until \( \alpha \ \beta \) at \( k \) because the interval may be incomplete between \( k \) and \( i^* \). The function \( \text{update}_\text{until} \) (Fig. 8) maintains this invariant for every advance of \( i^* \). Its arguments have the same meaning as for \( \text{update}_\text{since} \). However, it does not compute the results, which is instead done by \( \text{eval}_\text{until} \). 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 \( f \) units ahead of the associated time-stamp \( t \).

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

\[
| \text{eval } n \ t \ db \ (\text{Since}_2 \ p \ s_1 \ I \ s_2 \ b \ ts \ aux) = (\text{let} \ (xs, s'_1) = \text{eval } n \ t \ db \ s_1; \ \\
(y, s'_2) = \text{eval } n \ t \ db \ s_2; ((zs, aux), b, ts) = \text{tbuf}_\text{take} (\text{let} r_1 \ r_2 \ τ \ aux \ \in (zs @ [2], aux)); \ \\
(\text{let} (z, aux) = \text{update}_\text{since} I \ p \ r_1 \ r_2 \ t \ aux \ \in (zs @ [2], aux)); \ \\
(ys \ @ t) = \text{buf}_\text{add} \ xs \ ys \ b \ (ts @ [t]) \ \\
in (zs, \text{Since}_2 \ p \ s'_1 \ I \ s'_2 \ b \ ts \ aux) |
\]

| \text{eval } n \ t \ db \ (\text{Until}_2 \ p \ s_1 \ I \ s_2 \ b \ ts \ aux) = (\text{let} \ (xs, s'_1) = \text{eval } n \ t \ db \ s_1; \ \\
(y, s'_2) = \text{eval } n \ t \ db \ s_2; ((zs, aux), b, ts) = \text{tbuf}_\text{take} (\text{update}_\text{until} I \ p) \ aux \ (\text{buf}_\text{add} \ xs \ ys \ b) \ (ts @ [t]); \ \\
(zs, aux) = \text{eval}_\text{until} I \ \text{case} \ ts \ of \ [t] \Rightarrow t' | t' # \Rightarrow t' \ aux \ \\
in (zs, \text{Until}_2 \ p \ s'_1 \ I \ s'_2 \ b \ ts \ aux) |
\]

Here, \( \text{tbuf}_\text{take} \) (Fig. 8) is used instead of \( \text{buf}_\text{take} \) as we must consider the time-stamps \( ts @ [t] \) corresponding to the subformulas’ (future) results. Unlike \( \text{buf}_\text{take} \), 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 \( i_0 \), which states how far the subformula \( \psi \) has been evaluated. The function \( \text{prog} \) in Fig. 9 defines \( i_0 \) concretely. Its arguments are the trace \( \sigma \), an arbitrary MFOTL formula \( \psi \), 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 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 t1 t2) j = j  
  prog σ (Neg ψ) j = prog σ ψ j  
  prog σ (Or α β) j = min (prog σ α) (prog σ β) j  
  prog σ (Exists ψ) j = prog σ ψ j  
  prog σ (Since α I β) j = min (prog σ α) (prog σ β) j  
  prog σ (Until α I β) j =
  \ Inf \{ i. \forall k. k < j \land k \leq min (prog σ α) (prog σ β) j \mapsto τ σ i + right I \geq τ σ k \}

definition prog2 σ α β j = min (prog σ α) (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) ⟷ wf_prefix π ∧ n = nfv ϕ ∧ \ prefix_of π σ ⟷
  i = prog σ ϕ (length π) ∧ wf_state σ (length π) n ≤ ϕ

inductive wf_state :: trace ⇒ nat ⇒ nat ⇒ state ⇒ frm ⇒ bool where

the cached degree \( n \) agrees with the formula. All auxiliary invariants that are used
to define \( \text{wf}_\text{mstate} \) are expressed in terms of infinite streams instead of prefixes. This
includes \( \text{wf}_\text{state} \) and \( \text{wf}_\text{aux} \), which holds iff \( s :: \text{state} \) corresponds to the monitorable
formula \( ϕ :: \text{frm} \) after monitoring the first \( j \) databases of \( σ \). Therefore, we consider all
event streams \( σ \) that extend the prefix \( π \), i.e., \text{prefix_of} π σ.

We only show the case for \text{Until} in \text{wf_state} here. This case states the conditions
under which \text{Until} \( p s1 s2 ts aux \) is a well-formed state corresponding to \text{Until} \( α I β \).
Depending on the flag \( p, α \) is either a negated subformula \( α' = \text{Neg} α \) for some \( α \), or
its outermost operator is not a negation and \( α' = α \). The invariant inherits the condition
\( \text{fv} α \subseteq \text{fv} β \) from the \( \text{mf} \) predicate (Sect. 5). The two subformula states \( m1 \) and \( m2 \) must
be recursively well-formed and correspond to \( α \) and \( β \), respectively.

The predicate \( \text{wf}_\text{buf} \) (Fig. 11) encodes the invariant for buffers of type \text{buf}. These
store all results of the formulas \( α \) and \( β \) from index \( \text{prog2} σ α β j \) to indices \( \text{prog} σ α j \) and \( \text{prog} σ β j \), respectively. The results must be tables assigning values to the free
variables of the corresponding formula. The invariant \( \text{wf}_\text{ts} \) ensures that the additional
time-stamp list \( ts \), which is used by binary temporal operators, contains all time-stamps
from the start of the result buffer to the most recent input, which has index \( j \).

The invariant for the auxiliary states \( aux \) 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 \([l_\alpha ..<i^*]\), where \( l_\alpha = prog_\alpha \psi j \) and \( i^* = prog_\alpha \alpha' \beta j = prog_\alpha \alpha \beta j \) (the possible negation in \( \alpha' \) 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 \( \varphi \) and stream prefix \( \pi \), it returns exactly the pairs \((k, \nu)\) where the monitor has made progress beyond \( k \), and for which \( sat \sigma \lor i \varphi \) is true for all traces \( \sigma \) that extend \( \pi \).

**Definition** verdicts :: \( frm \Rightarrow (db \times ts) list \Rightarrow (nat \times tuple) set \) where

\[
\text{verdicts} \; \varphi \; \pi = \{(k, \nu) \mid \text{wf}_\text{tuple} (\text{nfv} \; \varphi) \; (\text{fv} \; \nu) \lor (\forall \nu. \; \text{prefix}_\text{of} \; \pi \; \sigma \rightarrow \\
\; k < \text{prog} \; \sigma \; \varphi \; (\text{length} \; \pi) \land \text{sat} \; \sigma \lor \nu \varphi \})
\]

Using the completeness of \( \text{prog} \), 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 \( \varphi \), verdicts \( \varphi \) is sound and eventually complete, i.e., for all prefixes \( \pi \) extending the stream \( \sigma \), indices \( k \), and tuples \( \nu \),

(a) \((k, \nu) \in \text{verdicts} \; \varphi \; \pi \rightarrow \text{sat} \; \sigma \lor k \varphi \), and

(b) \( k < \text{length} \; \pi \land \text{wf}_\text{tuple} (\text{nfv} \; \varphi) \; (\text{fv} \; \nu) \lor (\forall \nu'. \; \text{prefix}_\text{of} \; \pi \; \sigma' \rightarrow \\
\; \text{sat} \; \sigma' \lor k \varphi \rightarrow \\
\; (\forall \nu'. \; \text{prefix}_\text{of} \; \pi' \; \sigma' \land (k, \nu) \in \text{verdicts} \; \varphi \; \pi')\).

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 \( \text{last}_\text{ts} \; \pi \) denote the last time-stamp of \( \pi \), and 0 if \( \pi \) is empty.

**Theorem 1.** (a) \( \text{init} \) establishes the invariant: \( \text{mf} \; \varphi \implies \text{wf}_\text{mstate} \; \varphi \; [] \) (init \( \varphi \)).

(b) \text{step} preserves the invariant and its output can be described in terms of verdicts: Let \( \text{step} \; (\text{db}, t) \; \text{mst} = (A, \text{mst}') \). If \( \text{mf}_\text{mstate} \; \varphi \; \text{mst} \) and \( \text{last}_\text{ts} \; \pi \leq t \), then we have A = verdicts \( \varphi \; (\pi \; @ \; [(\text{db}, t)]) \) = verdicts \( \varphi \) and \( \text{wf}_\text{mstate} \; \varphi \; (\pi \; @ \; [(\text{db}, t)]) \) = verdicts \( \varphi \; (\pi \; @ \; [(\text{db}, t)]) \) = verdicts \( \varphi \; (\pi \; @ \; [(\text{db}, t)]) \).

**Corollary 1.** If \( \text{mf} \; \varphi \) and \( \text{wf}_\text{prefix} \; \pi \) monitor \( \varphi \; \pi = \text{verdicts} \; \varphi \; \pi \).
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 interval $0 \rightarrow \infty$ 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 $P \land Q$ (i.e., $\text{And (Pred P \{\}} (\text{Pred Q \{\}))$ 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 $\alpha \wedge \neg(\beta S \alpha)$ (i.e., \texttt{AndNot $\alpha$ (Since $\beta$ (interval \(0.0\)) $\alpha$)}) where $\text{fv} \beta > 0$, $\text{fv} \beta \subset \text{fv} \alpha$, and the order of occurrences of free variables in the two instances of $\alpha$ 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 $\alpha$, when monitoring a formula of the form $\neg\exists x. \alpha \wedge x = 24$ (i.e., $\texttt{Neg (Exists (And $\alpha$ (Eq (V 0) (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., $\texttt{Neg (Exists (AndNot (Eq (V 0) (C 24))) (Pred P [V 0])))})$ when monitored on a prefix without the event $(P, [24])$. 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). Srdan 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