Scaling Up Proactive Enforcement: Technical Report

François Hublet[1], Leonardo Lima[2], David Basin[1], Srđan Krstić[1], and Dmitriy Traytel[2]

[1] ETH Zürich, Zurich, Switzerland
{francois.hublet, basin, srdan.krstic}@inf.ethz.ch
[2] University of Copenhagen, Denmark
{leonardo, traytel}@di.ku.dk

Abstract. Runtime enforcers receive events from a system and output commands ensuring the system's policy compliance. Proactive enforcers extend traditional (reactive) enforcers by emitting commands at any time, rather only as a response to system actions. However, proactive enforcers have so far lacked support for many useful policy features. This, along with the existing tools' poor performance, hinders their adoption. We present a performance-optimized, proactive enforcement algorithm for a rich policy language: metric first-order temporal logic with function applications, aggregations, and `let` bindings. We have implemented this algorithm in EnfGuard, the first proactive enforcer tool that supports the above constructs. We evaluated our tool using a novel set of six benchmarks containing both real-world and synthetic policies and logs, demonstrating that it enforces realistic policies out-of-the-box and achieves the necessary performance to be used in real-time systems.

1 Introduction

Statically certifying the behavior of large, complex systems is often impossible. As an alternative, runtime enforcement [42] has emerged as a family of techniques aimed at observing and correcting the behavior of systems during their execution.

In runtime enforcement, an *enforcer* is a policy enforcement mechanism that observes the real-time execution of a system under enforcement (SuE) through the sequence of *events* that occur in it and sends *commands* to the SuE to ensure policy compliance (Figure 1). These commands instruct the system to suppress, cause, modify, or delay specific events. In *reactive* enforcement, the enforcer emits commands immediately upon receiving events (Figure 1, interactions 1.1–1.2). In *proactive* enforcement [5], the enforcer can additionally give commands at any time, rather than only after SuE events (Figure 1, interactions 2.1–2.2). This is crucial whenever policies require action to be taken before a deadline, even in the absence of SuE actions, as in common, e.g., in privacy regulations [25].

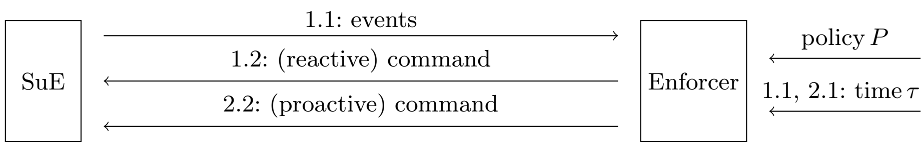


Fig. 1: Communication diagram for enforcement. R-step: 1.1, 1.2; P-step: 2.1, 2.2

To be practical, enforcers must be able to process SuE events at high rates. Moreover, they should support policies written in an expressive specification language. As an example, consider the policy stating "an alert must be raised whenever, within a 30-minute window, a data center dc has seen a pattern of unintended reboots of its servers that is classified as an outlier by Grubbs's test [19]:"

$$\mathsf{let}\ \mathsf{badReboot}(s, dc) = \mathsf{reboot}(s, dc) \wedge \neg \bullet(\neg \mathsf{reboot}(s, dc)\ \mathsf{S}\ \mathsf{intendReboot}(s, dc))\ \mathsf{in}$$
$$\mathsf{let}\ \mathsf{cntReboots}(dc, c) = c \leftarrow \mathtt{CNT}(i; dc)(\blacklozenge_{[0,1800)}(\mathsf{badReboot}(s, dc) \wedge \mathsf{tp}(i)))\ \mathsf{in}$$
$$\Box(\forall dc, l.\ dc, l \leftarrow \mathtt{GRUBBS}(dc, c;)\,(\mathsf{cntReboots}(dc, c)) \wedge l \approx 1$$
$$\longrightarrow \mathsf{alert}(\text{"Data center "}\ \hat{}\ \mathsf{int\_to\_string}\ dc\ \hat{}\ \text{" has rebooted too often"}))$$

In this policy, the user-defined aggregation function GRUBBS takes a finite sequence of pairs (k_i, v_i) with k_i an integer key and v_i a floating-point value, and returns a sequence of pairs (k_i, b_i), where $b_i = 1$ iff the Grubbs test identifies v_i as an outlier in $\{v_1, ..., v_i, ...\}$. A special event tp is used to retrieve the current timepoint. Moreover, this policy contains: *applications of a function* int_to_string and a string concatenation operator (ˆ); *aggregations* that use a user-defined aggregation function GRUBBS and an SQL-style aggregation operator CNT ('count') with grouping, e.g., cntReboots counts the number of reboots in each data center within the last 1800 seconds ($\blacklozenge_{[0,1800)}$ operator); and let *bindings* that define, e.g., an 'unintended reboot' as a reboot event that does not follow (S operator) an announce_reboot event strictly in the past ($\bullet$ operator). To the best of our knowledge, none of the existing proactive enforcement algorithms [5,24,25] supports any of these features. Thus, they cannot enforce policies like the above.

In this paper, we present the first proactive enforcement algorithm that supports metric first-order temporal logic (MFOTL) with function applications, aggregations, and let bindings. We implement this algorithm in EnfGuard, a new tool building on an existing proactive enforcement algorithm for simple MFOTL policies [25]. The original algorithm works as follows: (1) it maintains a queue of temporal obligations with deadlines (e.g., "fulfill $P(5)$ within three hours"); (2) it checks if newly observed events fulfill pending obligations (e.g., if $\mathsf{P}(5)$ occurred), proactively causing events when any deadline risks being missed; and (3) it suppresses and causes events reactively. In addition to supporting a more expressive policy language, EnfGuard achieves up to 30× speedup over prior work.

We evaluate EnfGuard on six benchmarks involving a combination of both real-world and synthetic policies and system logs. Our evaluation shows that our tool, unlike previous work [24,25], directly supports all policies from these benchmarks and can enforce them at high event rates (up to 1,000–10,000 events/s).

After reviewing prior work (Section 2), we make the following contributions:

– We extend prior work to support function applications, aggregations, and let bindings (Section 3). This extension fundamentally changes the underlying data structures, the enforcement algorithm, and the enforceable formulae.
– We describe our enforcement algorithm's optimizations (Section 4). These involve the lazy evaluation of Boolean operators, skipping unnecessary subformulae evaluation, and memoization of subformula evaluation results.
– We implement our algorithm in the EnfGuard enforcer. We validate our

tool's expressiveness and performance on six benchmarks (Section 5), showing that it can be used in real-time and surpasses existing tools' capabilities. The proofs of all propositions can be found in the Appendix. EnfGuard is open source and is publicly available on GitHub [26].

Related Work. Reactive enforcement was introduced by Schneider et al. using security automata [42,14] that terminate the SuE to prevent violations. Subsequent research supported the suppression [10,18] and causation [31] of individual events by buffering SuE events before making decisions. This (unrealistic) buffering capability was later dropped [35], and other capabilities, such as delaying events [38,15] and SuE code inspection [39], were considered.

Many enforcers use (timed) automata either as a policy language [16,17] or as the translation target for logics such as MITL [37,41]. Controller synthesis tools for LTL [27,13,44], Timed CTL [11,36], and MTL [30,23] also generate enforcers.

Very few works enforce *first-order temporal* policies: Hallé and Villemaire [20] give an enforcer for LTL-FO$^+$, a first-order variant of future-only LTL. Hublet et al. [24] reactively enforce a restricted set of MFOTL policies that cannot refer to the future. Aceto et al. [1,2] consider safety policies in Hennessy-Milner Logic with recursion; their approach is non-metric and does not support causation.

To the best of our knowledge, only two works study *proactive* enforcement. Basin et al. [5] describe a proactive enforcer for finite automata and dynamic condition response graphs [22], which is a propositional formalism. Hublet et al. [25] provide the only existing proactive first-order enforcement algorithm, which we substantially extend in this paper.

2 Preliminaries

We now review proactive enforcement (Section 2.1) and metric first-order temporal logic (Section 2.2). We then summarize the relevant data structures (Section 2.3) and the enforcement algorithm (Section 2.4) by Hublet et al [25].

2.1 Proactive runtime enforcement

Let Σ be a signature $(\mathbb{D}, \mathbb{E}, a)$ with an infinite domain $\mathbb{D}$ of values, a finite set of *event names* $\mathbb{E}$, each with arity $a(e) \in \mathbb{N}, e \in \mathbb{E}$. An *event* $e(d_1, \ldots, d_{a(e)}) \in \mathbb{E} \times \mathbb{D}^{a(e)}$ is a pair of an event name e and its $a(e)$ parameters $d_1, \ldots, d_{a(e)}$.

Events encode system actions that can be observed and controlled by the enforcer, or only observed. The enforcer can control an event by suppressing or causing it. We partition $\mathbb{E}$ into *suppressable* event names ($\mathbb{S} \subseteq \mathbb{E}$), *causable* event names ($\mathbb{C} \subseteq \mathbb{E}$), and *observable* event names ($\mathbb{O} = \mathbb{E} \setminus (\mathbb{S} \cup \mathbb{C})$). The enforcer can cause all events with names in $\mathbb{C}$ and suppress all events with names in $\mathbb{S}$. The set $\mathbb{DB}$ of *databases* over Σ is $\mathcal{P}(\{e(\overline{d}) \mid e \in \mathbb{E},\ \overline{d} \in \mathbb{D}^{a(e)}\})$ and a *trace* σ is a sequence $\langle(\tau_i, D_i)\rangle_{0 \leq i \leq k}, k \in \mathbb{N} \cup \{\infty\}$ of timestamps $\tau_i \in \mathbb{N}$ and finite databases $D_i \in \mathbb{DB}$, where timestamps grow monotonically ($\forall i < |\sigma|.\ \tau_i \leq \tau_{i+1}$) and progress (if $|\sigma| = \infty$, then $\lim_i \tau_i = \infty$). An index $0 \leq i < |\sigma|$ in a trace σ is called a *time-point*. The empty trace is denoted by ε, the set of all traces by $\mathbb{T}$,

1 $\mathsf{run}(s, \sigma, \sigma', \tau) = \textbf{case } \sigma' \textbf{ of } \varepsilon \Rightarrow \varepsilon$
2 $\mid (\tau', D) \cdot \sigma'' \textbf{ when } \tau' > \tau \Rightarrow \textbf{let } (o, s') = \mu(\sigma, s, \tau, \mathsf{tick}) \textbf{ in}$
3 $\quad \textbf{case } o \textbf{ of } \mathsf{PCom}(D_{\mathbb{C}}) \Rightarrow (\tau, D_{\mathbb{C}}) \cdot \mathsf{run}(s', \sigma \cdot (\tau, D_{\mathbb{C}}), \sigma', \tau + 1)$
4 $\qquad\qquad \mid \mathsf{NoCom} \Rightarrow \mathsf{run}(s', \sigma, \sigma', \tau + 1)$
5 $\mid (\tau', D) \cdot \sigma'' \textbf{ when } \tau' = \tau \Rightarrow \textbf{let } (o, s') = \mu(\sigma, s, \tau, D); D' = (D \setminus D_{\mathbb{S}}) \cup D_{\mathbb{C}} \textbf{ in}$
6 $\quad \textbf{case } o \textbf{ of } \mathsf{RCom}(D_{\mathbb{C}}, D_{\mathbb{S}}) \Rightarrow (\tau, D') \cdot \mathsf{run}(s', \sigma \cdot (\tau, D'), \sigma'', \tau + 1)$
7 $\mathcal{E}(\sigma) = \mathsf{run}(s_0, \varepsilon, \sigma, \textbf{case } \sigma \textbf{ of } \varepsilon \Rightarrow 0 \mid (\tau, D) \cdot \sigma' \Rightarrow \tau)$

Fig. 2: Enforced trace

and the set of finite (resp. infinite) traces by $\mathbb{T}_f$ (resp. $\mathbb{T}_\omega$). For traces $\sigma \in \mathbb{T}_f$ and $\sigma' \in \mathbb{T}$, $\sigma \cdot \sigma'$ denotes their concatenation. A *property* is a subset $P \subseteq \mathbb{T}_\omega$.

Given a prefix of a SuE trace, a *proactive enforcer* can either perform a (reactive) R-step (Figure 1, interactions 1.1 and 1.2), where it reads a new timestamp τ and database D, or a (proactive) P-step (interactions 2.1 and 2.2) where it reads a τ only. In both cases, it returns an appropriate *command*. In R-steps, a command is of the form $\mathsf{RCom}(D_{\mathbb{C}}, D_{\mathbb{S}})$ where $D_{\mathbb{C}}$ and $D_{\mathbb{S}} \subseteq D$ are databases over the signatures $(\mathbb{D}, \mathbb{C}, a)$ and $(\mathbb{D}, \mathbb{S}, a)$, respectively. Such a command instructs the SuE to cause $D_{\mathbb{C}}$ and suppress a subset $D_{\mathbb{S}}$ of D. In P-steps, a command is of the form $\mathsf{PCom}(D_{\mathbb{C}})$ or NoCom. In the former case, $D_{\mathbb{C}}$ is caused; in the latter, no event is caused or suppressed. Cmd denotes the set of all commands.

Definition 1. *A* (proactive) enforcer $\mathcal{E}$ *is a triple* $(\mathcal{S}, s_0, \mu)$*, where* $\mathcal{S}$ *is a set of states,* $s_0 \in \mathcal{S}$ *is an initial state, and* $\mu : \mathbb{T}_f \times \mathcal{S} \times \mathbb{N} \times (\mathbb{DB} \cup \{\mathsf{tick}\}) \to \mathsf{Cmd} \times \mathcal{S}$ *is a computable* update *function, such that the following two conditions hold:*

$$\forall \sigma, \tau, D \neq \mathsf{tick}, s.\ \exists D_{\mathbb{C}}, D_{\mathbb{S}}, s'.\ \mu(\sigma, s, \tau, D) = (\mathsf{RCom}(D_{\mathbb{C}}, D_{\mathbb{S}}), s') \wedge D_{\mathbb{S}} \subseteq D$$
$$\forall \sigma, s, \tau.\ \exists D_{\mathbb{C}}, s'.\ \mu(\sigma, s, \tau, \mathsf{tick}) \in \{(\mathsf{PCom}(D_{\mathbb{C}}), s'), (\mathsf{NoCom}, s')\}.$$

The first three arguments of μ are the trace prefix σ (containing all of the past excluding the present), the state of the enforcer s, and the current timestamp τ. In R-steps, μ's fourth argument is a new database D and μ returns $\mathsf{RCom}(D_{\mathbb{C}}, D_{\mathbb{S}})$. In P-steps, μ's fourth argument is the special symbol tick and the enforcer can return either $\mathsf{PCom}(D_{\mathbb{C}})$ or NoCom. This induces a trace transduction:

Definition 2. *For any* $\sigma \in \mathbb{T}$ *and enforcer* $\mathcal{E} = (\mathcal{S}, s_0, \mu)$*, the* enforced trace $\mathcal{E}(\sigma)$ *is defined co-recursively in Figure 2.*

To compute the enforced trace $\mathcal{E}(\sigma)$ from the original SuE trace σ, the update function μ is called once on every time-point to generate an R-command (lines 6–7) and once before each clock tick to generate a P-command (lines 3–5).

The enforcer's correctness with respect to a target property P is typically expressed in terms of *soundness* and *transparency* [31]. A sound enforcer ensures that the modified trace always complies with P, while a transparent enforcer modifies the system's behavior *only when necessary* to ensure compliance.

Definition 3. *An enforcer* $\mathcal{E}$ *is* sound *with respect to a property* P *iff for any* $\sigma \in \mathbb{T}_\omega$*,* $\mathcal{E}(\sigma) \in P$*. An enforcer* $\mathcal{E} = (\mathcal{S}, s_0, \mu)$ *is* transparent *with respect to a property* P *iff for any* $\sigma \in P$*,* $\mathcal{E}(\sigma) = \sigma$*. A property* P *(resp. a formula* φ*) is* enforceable *iff there exists a sound enforcer with respect to* P *(resp.* $\mathcal{L}(\varphi)$*).*

2.2 Metric first-order temporal logic

Metric first-order temporal logic (MFOTL) [9,12] is an expressive logic for specifying trace properties. In this paper, we extend MFOTL with function applications in terms, aggregations [8], and non-recursive let bindings [45]. Our MFOTL syntax is defined by the following grammar (extensions highlighted):

$$t ::= c \mid x \mid f(t,\dots,t)$$
$$\varphi ::= e(t,\dots,t) \mid t \approx c \mid \neg\varphi \mid \varphi \land \varphi \mid \exists x.\ \varphi \mid \bigcirc_I \varphi \mid \CIRCLE_I \varphi \mid \varphi \,\mathsf{U}_I\, \varphi \mid \varphi \,\mathsf{S}_I\, \varphi$$
$$\mid\ x,\dots,x \leftarrow \omega(t,\dots,t;x,\dots,x)\ \varphi \ \mid\ \mathsf{let}\, e(x,\dots,x) = \varphi \ \mathsf{in}\ \varphi.$$

In the above, $e \in \mathbb{E}$, $c \in \mathbb{D}$, $i \in \mathbb{N}$, x ranges over a set $\mathbb{V}$ of variables, f over a set $\mathbb{F}$ of function names, and ω over a set $\Omega \supseteq \{\mathtt{SUM}, \mathtt{AVG}, \mathtt{STD}, \mathtt{MED}, \mathtt{CNT}, \mathtt{MIN}, \mathtt{MAX}\}$ of aggregation operators. In a subformula $\mathsf{let}\, e(\bar{t}) = \varphi_1 \ \mathsf{in}\ \varphi_2$, the event e is not allowed to appear in φ_1. We extend the arity function a to functions and aggregation operators so that for any $f \in \mathbb{F}$, $a(f) \in \mathbb{N}$ is the number of arguments of f, and for any $\omega \in \Omega$, $a(\omega)$ is a pair in $\mathbb{N}^2$ such that $a(\omega)_1$ and $a(\omega)_2$ are the input and output arities of ω, respectively. We define the shorthands $\top := p \lor \neg p$, $\bot := \neg\top$, $\varphi \longrightarrow \psi := \neg\varphi \lor \psi$, and the operators "once" ($\blacklozenge_I \varphi := \top \,\mathsf{S}_I\, \varphi$), "eventually" ($\Diamond_I \varphi := \top \,\mathsf{U}_I\, \varphi$), "always" ($\Box_I \varphi := \neg \Diamond_I \neg\varphi$), and "historically" ($\blacksquare_I \varphi := \neg \blacklozenge_I \neg\varphi$).The interval $[0,\infty)$ can be omitted in subscripts.

Next, we present the semantics of MFOTL, deferring the semantics of our extensions to Section 3. A *valuation* $v : \mathbb{V} \to \mathbb{D}$ maps variables to domain elements in $\mathbb{D}$. Under a valuation v, a variable x evaluates to $[\![x]\!]_v = v(x)$ and a constant $c \in \mathbb{D}$ to $[\![c]\!]_v = c$. We write $v[x \mapsto d]$ for the mapping v updated with the assignment $x \mapsto d$, where $x \in \mathbb{V}$ and $d \in \mathbb{D}$. The sequent $v, i \vDash_\sigma \varphi$ (defined in Figure 3 for a fixed, infinite σ) denotes that φ is satisfied at time-point i of trace σ under valuation v (i.e., v is a *satisfaction*). The property induced by a formula φ is $\mathcal{L}(\varphi) = \{\sigma \in \mathbb{T}_\omega \mid \exists v.\ v, 0 \vDash_\sigma \varphi\}$, and we say that a formula φ is *enforceable* when there exists a sound enforcer for $\mathcal{L}(\varphi)$.

We write $\mathsf{fv}(\varphi)$ and $\mathsf{const}(\varphi)$ for the set of free variables and constants of formula φ, respectively. The *active domain* $\mathsf{AD}_{\sigma,E}(\varphi)$ of a formula φ over a finite trace $\sigma = \langle(\tau_i, D_i)_{0 \le i < |\sigma|}\rangle$ and set of event names $E \subseteq \mathbb{E}$ is $\mathsf{const}(\varphi) \cup \Big(\bigcup_{0 \le j < |\sigma|} \{d \mid d \text{ is one of } d_k \text{ in } e(d_1, ..., d_{a(e)}) \in D_j \text{ and } e \in E\}\Big)$. Intuitively, the active domain consists of all domain values present in the trace as well as all constants occurring in the formulae.

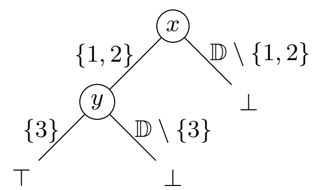
2.3 Partitioned decision trees

Let $\textsc{Sat}_\varphi(v, i, \sigma)$ be a function that returns true iff $v, i \vDash_\sigma \varphi$, i.e., iff a trace σ satisfies φ at i under v, and false otherwise. A *monitor* for a formula φ is an algorithm that computes $\textsc{Sat}_\varphi(v, i, \sigma)$ by incrementally observing σ's prefixes.

Inspired by binary decision diagrams [34], Lima et al. [33] introduce partitioned decision trees (PDTs) to compactly represent sets of valuations. PDTs

$$
\begin{array}{lll}
v,i \vDash t \approx c & \text{iff } [\![t]\!]_v = c \quad | \quad v,i \vDash e(t_1,...,t_{a(e)}) \text{ iff } e([\![t_1]\!]_v,...,[\![t_{a(e)}]\!]_v) \in D_i \\
v,i \vDash \exists x.\ \varphi & \text{iff } v[x \mapsto d], i \vDash \varphi \text{ for some } d \in \mathbb{D} \quad | \quad v,i \vDash \neg\varphi \quad \text{iff } v,i \nvDash \varphi \\
v,i \vDash \bigcirc_I \varphi & \text{iff } v,i+1 \vDash \varphi \text{ and } \tau_{i+1} - \tau_i \in I \quad | \quad v,i \vDash \varphi \wedge \psi \text{ iff } v,i \vDash \varphi \text{ and } v,i \vDash \psi \\
v,i \vDash \bullet_I \varphi & \text{iff } i > 0 \text{ and } v,i-1 \vDash \varphi \text{ and } \tau_i - \tau_{i-1} \in I \\
v,i \vDash \varphi \,\mathsf{U}_I\, \psi & \text{iff } v,j \vDash \psi \text{ for some } j \geq i \text{ with } \tau_j - \tau_i \in I \text{ and } v,k \vDash \varphi \text{ for all } i \leq k < j \\
v,i \vDash \varphi \,\mathsf{S}_I\, \psi & \text{iff } v,j \vDash \psi \text{ for some } j \leq i \text{ with } \tau_i - \tau_j \in I \text{ and } v,k \vDash \varphi \text{ for all } j < k \leq i
\end{array}
$$

Fig. 3: MFOTL semantics for a fixed, infinite trace σ



$\varphi := A(x) \wedge B(y)$

$\sigma := (0, \{A(1), A(2), B(3)\})...$

(a) PDT of φ's satisfactions on σ

```
1 let find d parts = case parts of
2     [(D, pdt)] ⇒ (D, pdt)
3   | (D, pdt) :: parts ⇒
4       if d ∈ D then (D, pdt) else find d parts
5 let specialize pdt v = case pdt of Leaf ℓ ⇒ ℓ
6   | Node x parts ⇒
7       let (_, pdt′) = find (v x) parts in
8       specialize pdt′
```

(b) Specialization of PDTs

Fig. 4: Partitioned decision trees (PDTs)

are trees whose internal nodes are labeled with free variables, whose edges are marked with sets of elements that partition $\mathbb{D}$, and whose leaves contain data of interest, e.g., Boolean values. The corresponding algebraic data type is $\mathsf{Pdt}\ a = \mathsf{Leaf}\ a \mid \mathsf{Node}\ \mathbb{V}\ (\mathcal{P}_c(\mathbb{D}) \times \mathsf{Pdt}\ a))$, where $\mathcal{P}_c(X)$ denotes the set of finite or co-finite subsets of X. An example of a PDT storing the satisfactions of the formula $\varphi := A(x) \wedge B(y)$ on a trace $\sigma := (0, \{A(1), A(2), B(3)\})...$ is shown in Figure 4a. Given a specific valuation v, the value $\textsc{Sat}_\varphi(v, i, \sigma)$ (indicating if v is a satisfaction) can be extracted from a PDT of $\textsc{Sat}_\varphi(\bullet, i, \sigma)$ using the specialize function shown in Figure 4b: for any leaf, the stored value is immediately returned (l. 8); for any node labeled by a variable x, the child whose edge label contains the value $v(x)$ is selected, and specialization continues from that child (l. 9–10).

Lima et al. [33] describe a monitoring algorithm for MFOTL based on PDTs. They first define a series of functional operations on PDTs, and then describe a monitoring algorithm combining these operations. For example, to compute $\textsc{Sat}_{\varphi_1 \wedge \varphi_2}(\bullet, i, \sigma)$, they apply a function $\mathsf{apply2}\ (\lambda b_1\, b_2.\ b_1 \wedge b_2)$ on the PDTs p_1 and p_2 of $\textsc{Sat}_{\varphi_1}(\bullet, i, \sigma)$ and $\textsc{Sat}_{\varphi_2}(\bullet, i, \sigma)$. This function is such that

$$\forall f, p_1, p_2, v.\ \mathsf{specialize}\,(\mathsf{apply2}\, f\, p_1\, p_2)\ v = f\,(\mathsf{specialize}\, p_1\, v)\,(\mathsf{specialize}\, p_2\, v).$$

Hence, applying $\mathsf{apply2}\ (\lambda b_1\, b_2.\ b_1 \wedge b_2)$ correctly evaluates the conjunction. Compared to table-based monitoring algorithms [9], PDT-based algorithms lift many of the restrictions on the supported MFOTL fragment imposed in previous work [9,40], thus significantly increasing expressivity.

2.4 Enforcement algorithm

Not all MFOTL formulae are enforceable, e.g., $\forall x.\ A(x) \longrightarrow B(x)$ is enforceable only if A is suppressable or B is causable. MFOTL enforceability is undecidable [24], yet there are syntactic fragments that guarantee enforceability.

Hublet et al. [25, Section 4] define such an enforceable fragment, called EMFOTL. EMFOTL is defined using type sequents $\Gamma \vdash \varphi : \alpha$, where the context $\Gamma : \mathbb{E} \to \{\mathbb{C}, \mathbb{S}\}$ is a mapping from event names to $\{\mathbb{C}, \mathbb{S}\}$, φ is an MFOTL formula, and $\alpha \in \{\mathbb{C}, \mathbb{S}\}$ is a type. Intuitively, a formula types to $\mathbb{C}$ under Γ ("φ is causable under Γ") if it can be enforced by causing events $e_c(...)$ such that $\Gamma(e_c) = \mathbb{C}$ and suppressing events $e_s(...)$ such that $\Gamma(e_s) = \mathbb{S}$. Conversely, it types to $\mathbb{S}$ under Γ ("φ is suppressable under Γ") if $\neg\varphi$ can be enforced under the same conditions on Γ. EMFOTL is defined as the set of all φ for which $\exists \Gamma.\, \Gamma \vdash \varphi : \mathbb{C}$. The types $\mathbb{C}$ and $\mathbb{S}$ overload the names of the sets of suppressable and causable event names so that only events $e(...)$ with $e \in \mathbb{C}$ (resp. $e \in \mathbb{S}$) can type to $\mathbb{C}$ (resp. $\mathbb{S}$).

The complete set of typing rules by Hublet et al. is given in Appendix A.

Example 1. Consider the formula $\varphi = \Box(\forall x.\ A(x) \longrightarrow \Diamond_{[0,30]} B(x))$ with $A \in \mathbb{O}$ and $B \in \mathbb{C}$. The formula φ can be shown enforceable using the rules

$$\frac{\vdash \varphi : \mathrm{PG}(x)^- \quad \Gamma \vdash \varphi : \mathbb{C}}{\Gamma \vdash \forall x.\, \varphi : \mathbb{C}}\,\forall^{\mathbb{C}} \quad \frac{\Gamma(e) = \mathbb{C} \quad e \in \mathbb{C}}{\Gamma \vdash e(t_1, ..., t_{a(e)}) : \mathbb{C}}\,\mathbb{E}^{\mathbb{C}} \quad \frac{}{\vdash e(..., x, ...) : \mathrm{PG}(x)^+}\,\mathbb{E}^+_{\mathrm{PG}}$$

$$\frac{\Gamma \vdash \varphi : \mathbb{C}}{\Gamma \vdash \Box\, \varphi : \mathbb{C}}\,\Box^{\mathbb{C}} \frac{a < \infty \quad \Gamma \vdash \varphi : \mathbb{C}}{\Gamma \vdash \Diamond_{[0,a]}\, \varphi : \mathbb{C}}\,\Diamond^{\mathbb{C}} \frac{\Gamma \vdash \psi : \mathbb{C}}{\Gamma \vdash \varphi \longrightarrow \psi : \mathbb{C}} \longrightarrow^{\mathbb{C}\mathrm{R}} \frac{\vdash \varphi : \mathrm{PG}(x)^+}{\vdash \varphi \longrightarrow \psi : \mathrm{PG}(x)^-} \longrightarrow^-_{\mathrm{PG}}$$

as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash A(x) : \mathrm{PG}(x)^+}\,\mathbb{E}^+_{\mathrm{PG}}}{\vdash A(x) \longrightarrow \Diamond_{[0,30]} B(x) : \mathrm{PG}(x)^-} \longrightarrow^-_{\mathrm{PG}} \quad \cfrac{\cfrac{30 < \infty \quad \cfrac{B \in \mathbb{C}}{B : \mathbb{C} \vdash B(x) : \mathbb{C}}\,\mathbb{E}^{\mathbb{C}}}{B : \mathbb{C} \vdash \Diamond_{[0,30]} B(x) : \mathbb{C}}\,\Diamond^{\mathbb{C}}}{B : \mathbb{C} \vdash A(x) \longrightarrow \Diamond_{[0,30]} B(x) : \mathbb{C}} \longrightarrow^{\mathbb{C}\mathrm{R}}}{B : \mathbb{C} \vdash \forall x.\ A(x) \longrightarrow \Diamond_{[0,30]} B(x) : \mathbb{C}}\,\forall^{\mathbb{C}}}{B : \mathbb{C} \vdash \Box(\forall x.\ A(x) \longrightarrow \Diamond_{[0,30]} B(x)) : \mathbb{C}}\,\Box^{\mathbb{C}}\,.$$

Each rule shows how to enforce the corresponding MFOTL operator. The $\forall^{\mathbb{C}}$ rule expresses that to cause $\forall x.\ \varphi$ (i.e., $\Gamma \vdash \forall x.\, \varphi : \mathbb{C}$), it is sufficient to (i) cause φ for any valuation (i.e., $\Gamma \vdash \varphi : \mathbb{C}$) and (ii) ensure that all x's values for which φ must be caused can be computed from the arguments of present or past events (i.e., $\vdash \varphi : \mathrm{PG}(x)^-$). Condition (ii), called *past-guardedness*, excludes formulas for which an infinite number of events must be caused. It is checked by other past-guardedness rules that derive sequents $\vdash \varphi : \mathrm{PG}(x)^+$ (resp. $\vdash \varphi : \mathrm{PG}(x)^-$) that mean "whenever φ is true (resp. false) for some valuation v, then $v(x)$ must be the argument of an event in the trace in the past or present". The $\mathbb{E}^+_{\mathrm{PG}}$ rule is the base case, whereas the $\longrightarrow^-_{\mathrm{PG}}$ rule states that when φ's satisfactions provide such values for x, then $\varphi \longrightarrow \psi$'s violations also do (since $\neg(\varphi \longrightarrow \psi)$ implies φ). The $\Box^{\mathbb{C}}$, $\longrightarrow^{\mathbb{C}\mathrm{R}}$, and $\Diamond^{\mathbb{C}}$ rules show how to enforce the other operators: to cause $\Box\, \varphi$, one must cause φ (at all times); to cause $\varphi \longrightarrow \psi$, one must cause ψ (when φ is false); to cause $\Diamond_{[0,a]}\, \varphi$ where $a < \infty$, one must cause φ (in at most b time units).

let $\mathsf{enf}\,(\sigma, X, ts, D) =$
  if $D \neq \mathsf{tick}$ **then** ▷ R-step
    let $\varPhi = \bigwedge_{(\xi,v,+)\in X} \xi(ts)[v] \wedge \bigwedge_{(\xi,v,-)\in X} \neg\xi(ts)[v]$ **in**
    let $(D_C, D_S, X') = \mathsf{enf}^{+}_{ts,\bot}(\varPhi, \sigma \cdot (ts, D \cup \{\mathtt{TP}\}), \emptyset, \emptyset)$ **in**
    $(\mathsf{RCom}(D_C, D_S), X')$
  else ▷ P-step
    let $\varPhi = \bigwedge_{(\xi,v,+)\in X} \xi(ts)[v] \wedge \bigwedge_{(\xi,v,-)\in X} \neg\xi(ts)[v]$ **in**
    let $(D_C, D_S, X') = \mathsf{enf}^{+}_{ts,\top}(\varPhi, \sigma \cdot (ts, \emptyset), \emptyset, \emptyset)$ **in**
    if $\mathtt{TP} \in D_C$ **then** $(\mathsf{PCom}(D_C \setminus \{\mathtt{TP}\}), X')$ **else** (NoCom, X)

let $\mathsf{enf}^{+}_{ts,b}\,(\varphi, \sigma, X, v) =$ **case** φ **of**
  $e(\bar{t}) \Rightarrow (\{e(\llbracket \bar{t} \rrbracket_v)\}, \emptyset, \emptyset)$
  $\mid \varphi_1 \longrightarrow^{\mathrm{CR}} \varphi_2 \Rightarrow \mathsf{enf}^{+}_{ts,b}(\varphi_2, \sigma, X, v)$
  $\mid \forall^{\mathbb{C}} x.\ \varphi_1 \Rightarrow \mathsf{fp}(\sigma, X, \mathsf{enf}^{+}_{\mathsf{all},\varphi_1,v,ts,b})$
  $\mid \Diamond^{\mathbb{C}}_{[0,a]}\, \varphi_1 \Rightarrow$
    if $a = 0 \wedge b$ **then**
      $\mathsf{enf}^{+}_{ts,b}(\varphi_1, \sigma, X, v)$
    else
      $(\emptyset, \emptyset, \{(\lambda\tau'.\, \Diamond_{[0,a-(\tau'-\tau)]}$
        $(\mathtt{TP} \wedge \varphi_1), v, +)\})$
  $\mid \Box^{\mathbb{C}}\, \varphi_1 \Rightarrow$
    $\mathsf{enf}^{+}_{ts,b}(\varphi_1, \sigma, X, v) \uplus$
      $(\emptyset, \emptyset, \{(\lambda\tau'.\, \Box\, \varphi_1, v, +)\})$
  $\ldots$
let $\mathsf{enf}^{-}_{ts,b}\,(\varphi, \sigma, X, v) = \ldots$

let $(\uplus)\ (D_C, D_S, X)\ (D'_C, D'_S, X') =$
  $(D_C \cup D'_C, D_S \cup D'_S, X \cup X')$
let $\mathsf{fp}\,(\sigma \cdot (\tau, D), X, f) =$
  $(D_C, D_S) \leftarrow (\emptyset, \emptyset);$ $r \leftarrow \mathsf{None}$
  while $(D_C, D_S, X) \neq r$ **do**
    $r \leftarrow (D_S, D_C, X)$
    $(D_C, D_S, X) \leftarrow r \uplus$
      $f(\sigma \cdot (\tau, (D \setminus D_S) \cup D_C), X)$
  (D_C, D_S, X)
let $\mathsf{enf}^{+}_{\mathsf{all},\varphi_1,v,ts,b}\,(\sigma, X) =$
  $r \leftarrow (\emptyset, \emptyset, \emptyset)$
  for $d \in \mathsf{AD}_{\sigma,\mathbb{E}}(\varphi_1)$ **do**
    if $\neg\textsc{Sat}^{*}_{\neg\varphi_1}(v[d/x], |\sigma| - 1, \sigma, X)$
    then $r \leftarrow r \uplus$
      $\mathsf{enf}^{+}_{ts,b}(\varphi_1, \sigma, X, v[d/x])$
  r

Fig. 5: Proactive real-time first-order enforcement algorithm [25, Algorithm 2]

The EMFOTL enforcement algorithm [25, Algorithm 2] is shown in Figure 5. Its state is a set $X \subseteq \mathsf{fo}$ of *future obligations*. The set fo of future obligations contains all triples (ξ, v, p) where ξ is a function $\mathbb{N} \to \text{EMFOTL}$, v a valuation, and $p \in \{+, -\}$. At every time-point i with timestamp ts, the algorithm enforces $\varPhi = \bigwedge_{(\xi,v,+)} \xi(ts)[v] \wedge \bigwedge_{(\xi,v,-)} \neg\xi(ts)[v]$ by causing or suppressing events and updating the future obligations to be enforced at $i + 1$.

The algorithm uses a $\textsc{Sat}^{*}$ monitor extending $\textsc{Sat}$ (Section 2.3) over finite traces in two ways: (1) $\textsc{Sat}^{*}$ inputs a set X of obligations assumed to hold after the last time-point. For example, $\textsc{Sat}^{*}_{\Box A}(v, 0, (0, \{A\}), \{(\lambda\tau.\ \Box A, \emptyset, +)\})$ holds: if A holds at time-point 0 and $\Box A$ is assumed to hold at time-point 1, then $\Box A$ holds at time-point 0; and (2) $\textsc{Sat}^{*}$ always returns a conservative evaluation of the formula when future information is lacking. For example, if A occurs at time-point 0, we can conclude that $\Diamond A$ holds ($\textsc{Sat}^{*}_{\Diamond A}(v, 0, (0, \{A\}), \emptyset)$), but not necessarily that $\Box A$ holds ($\neg\textsc{Sat}^{*}_{\Box A}(v, 0, (0, \{A\}), \emptyset)$) at time-point 0. A fixpoint computation is used in cases that require recursively enforcing multiple subformulae (e.g., causing $\forall x.\ \varphi$ or $\varphi_1 \wedge \varphi_2$). A special causable event $\mathtt{TP}$ denotes the *existence of a time-point*. Such an event is always present in R-steps, where a time-point already exists, but not in P-steps. In P-steps, causation of $\mathtt{TP}$ leads to the insertion of a time-point (i.e., a PCom).

Example 2. The algorithm from Figure 5 enforces the formula φ in Example 1 over the trace $\sigma = \langle (0, \{A(1)\}), (50, \{B(2)\}) \rangle$ as follows.

Initially, $ts = 0$, $D = \{A(1)\}$, and we have one future obligation corresponding to φ, namely $\mathsf{fo} = (\lambda\tau.\ \varphi, \emptyset, +)$. The algorithm performs an R-step on the first time-point; the formula to be enforced is $\Phi = \varphi$ (l. 3). Since $\varphi = \Box\,\psi$ with $\psi = \forall x.\ A(x) \longrightarrow \Diamond_{[0,30]}\, B(x)$, the algorithm generates the same future obligation fo and proceeds with enforcing ψ (l. 20–22). Next, since $\psi = \forall x.\ \chi$ where $\chi = A(x) \longrightarrow \Diamond_{[0,30]}\, B(x)$, the algorithm performs a fixpoint computation (l. 13; 27–33). In each iteration of this computation, the algorithm enforces χ under all valuations $\{x \mapsto d\}_{d\in\mathbb{D}}$ for which χ is not yet satisfied (l. 34–40). Here, the only such valuation is $v = \{x \mapsto 1\}$. Since $\chi = A(x) \longrightarrow \chi'$ where $\chi' = \Diamond_{[0,30]}\, B(x)$ and the rule $\longrightarrow^{\mathbb{C}\mathrm{R}}$ was used to type χ in Example 1, the algorithm enforces χ' under v (l. 12). It does so by generating the future obligation $\mathsf{fo}' = (\lambda\tau.\ \Diamond_{[0,30-\tau]}(\mathtt{TP} \wedge B(x)), \{x \mapsto 1\}, +)$ (l. 19). After generating fo and fo', the formula Φ holds and the computation terminates, returning $\mathsf{RCom}(\emptyset, \emptyset)$.

Next, the algorithm performs a P-step with $ts = 0$. The formula to be enforced, computed from fo and fo', is $\Phi = \Box\,\psi \wedge \Diamond_{[0,30]}(\mathtt{TP} \wedge B(1))$ (l. 7). To satisfy Φ's two conjuncts, the future obligations fo and $\mathsf{fo}'' = (\lambda\tau.\ \Diamond_{[0,30-\tau]}(\mathtt{TP} \wedge B(1)), \emptyset, +)$ are generated. The logic used to enforce $\Box$ and $\Diamond$ is the same as above; the enforcement of $\wedge$ uses a fixpoint computation (omitted in Figure 5). As generating fo and fo' suffices to satisfy Φ, the algorithm returns NoCom.

Since there is no time-point with timestamp 1 in the trace, the enforcer then performs a P-step with $ts = 1$. The formula to be enforced is $\Phi = \Box\,\psi \wedge \Diamond_{[0,29]}(\mathtt{TP} \wedge B(1))$; note the smaller bound on $\Diamond$ due to the new ts. The algorithm again generates the future obligations $\{\mathsf{fo}, \mathsf{fo}''\}$. Similarly, a P-step is performed for $ts = 2, \ldots, 29$, propagating $\{\mathsf{fo}, \mathsf{fo}''\}$. Each of these P-steps returns NoCom.

When ts reaches 30, the algorithm enforces $\Phi = \Box\,\psi \wedge \Diamond_{[0,0]}(\mathtt{TP} \wedge B(1))$. Since $\Diamond$'s interval is $[0, 0]$, this conjunct can only be enforced by causing $\mathtt{TP} \wedge B(1)$ (l. 16), i.e., causing both $\mathtt{TP}$ and $B(1)$. The future obligation fo is also generated. The algorithm returns $\mathsf{PCom}(\{B(1)\})$, inserting a time-point $(30, \{B(1)\})$ in σ.

Beyond this time-point, the trace always satisfies ψ and the set of future obligations is just $\{\mathsf{fo}\}$. Therefore, the trace is not further modified.

3 An Extended Enforceable Fragment of MFOTL

We now describe the semantics, typing rules, and monitoring and enforcement algorithms for our three extensions. All proofs of soundness and transparency are given in Appendix A.

3.1 Function applications

Assume that every function symbol $f \in \mathbb{F}$ is associated with a (terminating) function $\hat{f} : \mathbb{D}^{a(f)} \to \mathbb{D}$. Our semantics of terms is standard:

$$[\![c]\!]_v = c \qquad [\![x]\!]_v = v(x) \qquad [\![f(t_1, \ldots, t_{a(f)})]\!]_v = \hat{f}([\![t_1]\!]_v, \ldots, [\![t_{a(f)}]\!]_v)$$

Monitorability. To ensure that only finitely many function calls are needed to decide whether a given formula is satisfied, restrictions must be imposed. In contrast to classical monitorability which focuses on *informative prefixes* [29], our definition focuses on ensuring finite evaluation steps of first-order formulae.

Example 3. Given a binary function $\mathsf{eq} \in \mathbb{F}$ such that $\mathsf{eq}(x, y) := \textbf{if } x = y \textbf{ then } 1$ $\textbf{else } 0$ used to compare two variables, and some $f \in \mathbb{F}$, consider the formulae

$$\varphi_1 := \forall x, y.\ B(x) \wedge B(y) \wedge \neg(\mathsf{eq}(x, y) \approx 1) \longrightarrow A(f(x, y))$$
$$\varphi_2 := \forall x, y.\ A(f(x, y)) \longrightarrow B(x) \wedge B(y) \wedge \neg(\mathsf{eq}(x, y) \approx 1).$$

The formula φ_1 is monitorable: whenever two B events occur for different values of x and y, the event $A(f(x, y))$ also occurs. In contrast, the formula φ_2 cannot be monitored without further assumptions about f: when some $A(z)$ is true, the set of pairs (x, y) such that $z = f(x, y)$ may be neither finite nor co-finite.

The key difference between the formulae is that, when φ_1 is false, there are always events in the present that contain x and y as parameters. There are finitely many such events, and hence the full set of satisfactions can be obtained by filtering satisfactions of $B(x) \wedge B(y) \wedge \neg(\mathsf{eq}(x, y) \approx 1)$ based on the value of $A(f(x, y))$. In contrast, when φ_2 is false, all values of x and y for which $A(f(x, y))$ is true (or, alternatively, $B(x) \wedge B(y) \wedge \neg(\mathsf{eq}(x, y) \approx 1)$ is false) would need to be checked, but the set of such values may be infinite.

Based on these observations, we adopt the following notion of monitorability:

Definition 4. *A closed MFOTL formula φ is monitorable iff for any of its quantified subformulae $Qx.\ \psi$, where $Q \in \{\forall, \exists\}$, either $\vdash \psi : PG^+(x)$, or $\vdash \psi : PG^-(x)$, or x does not appear inside any function argument in ψ.*

Note that the definition of rule $\mathbb{E}^+_{\mathrm{PG}}$ shown in Example 1 is unchanged, i.e., a variable is only past-guarded when it occurs directly as an argument of a predicate, and not within a function application.

Monitoring. We now describe how to extend the PDTs from Section 2.3 to efficiently monitor formulae with function applications. Instead of trees labeled by variable names, we consider trees labeled with elements of the type

$$\mathsf{lbl} = \mathsf{LVar}\ \mathit{ident} \mid \mathsf{LEx}\ \mathit{ident} \mid \mathsf{LAll}\ \mathit{ident} \mid \mathsf{LClos}\ \mathit{ident}\ (\mathit{term}\ \mathsf{list}),$$

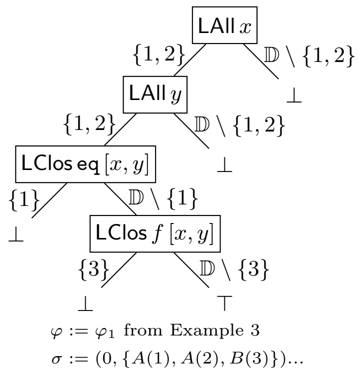
containing either free variables (LVar), existentially quantified variables (LEx), universally quantified variables (LAll), or closures with a function name and a list of terms (LClos). An example of an extended PDT is shown in Figure 6a.

We call a PDT *well-formed* with respect to a set of variables V iff:

1. Any $\mathsf{LClos}\, f\, \bar{t}$ node with $z \in \mathsf{fv}(\bar{t}) \cap V$ has an $\mathsf{LEx}\, z$ or $\mathsf{LAll}\, z$ node higher up.

This condition ensures that the value of all terms with free variables in V labeling a node can be computed using the knowledge of the value of variables higher up.

Given a PDT representing satisfactions $\textsc{Sat}_\varphi(\bullet, i, \sigma)$ well-formed with respect to the set of all variables in φ, a valuation v can be checked as in Figure 6b. In Appendix A, we extend Lima et al.'s [33] algorithm to use the new PDTs and show that it monitors all formulae covered by Definition 4.



(a) PDT of φ's satisfactions on σ

```
let specialize pdt v = case pdt of Leaf ℓ ⇒ ℓ
    | Node (LVar x) parts ⇒
        let (_, pdt′) = find parts (v x) in
        specialize pdt′ v
    | Node (LEx x) parts ⇒
        ⋁_{(D,pdt′)∈parts, |D|<∞} ⋁_{d∈D} specialize pdt′ v[x ↦ d]
        ∨ ⋁_{(D,pdt′)∈parts, |D|=∞} specialize pdt′ v
    | Node (LAll x) parts ⇒
        ⋀_{(D,pdt′)∈parts, |D|<∞} ⋀_{d∈D} specialize pdt′ v[x ↦ d]
        ∧ ⋀_{(D,pdt′)∈parts, |D|=∞} specialize pdt′ v
    | LClos f t̄ ⇒ specialize (find parts ⟦f(t̄)⟧_v) v
```

(b) Specialization of extended PDTs

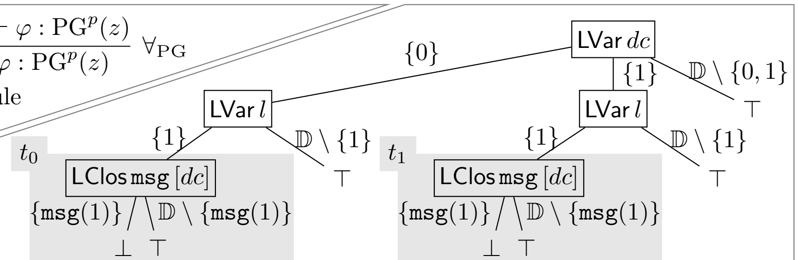
Fig. 6: Extended PDTs

Example 4. Consider the formula $\varphi_{\mathsf{Grubbs}}$ from Section 1. Let $\varphi'_{\mathsf{Grubbs}} := dc, l \leftarrow \mathtt{GRUBBS}(dc, c;)(\mathsf{cntReboots}(dc, c))) \land l \approx 1$ and $\varphi''_{\mathsf{Grubbs}} := \varphi'_{\mathsf{Grubbs}} \longrightarrow \mathsf{alert}(\mathtt{msg}(dc))$, where $\mathtt{msg}(dc)$ abbreviates the string term in $\varphi_{\mathsf{Grubbs}}$'s alert event. Note that only variable dc occurs within a function argument. By Definition 4, the formula $\varphi_{\mathsf{Grubbs}}$ is monitorable iff $\forall l.\ \varphi''_{\mathsf{Grubbs}}$ is either $\mathrm{PG}^+(dc)$ or $\mathrm{PG}^-(dc)$. In Example 7, we will show that $\varphi'_{\mathsf{Grubbs}}$ is $\mathrm{PG}^+(dc)$. Using rules $\longrightarrow^-_{\mathrm{PG}}$ and $\forall_{\mathrm{PG}}$ (see (i) below), we show that $\forall l.\ \varphi''_{\mathsf{Grubbs}}$ is also $\mathrm{PG}^+(dc)$. Thus, $\varphi_{\mathsf{Grubbs}}$ is monitorable.

Suppose that $\varphi'_{\mathsf{Grubbs}}$ holds for $(dc, l) \in \{(0,1),(1,1)\}$ and $\mathsf{alert}(m)$ holds iff $m = \mathtt{msg}(1)$. Monitoring $\varphi''_{\mathsf{Grubbs}}$, our extended SAT computes the PDT below (ii).

$$\frac{x \neq z \quad \Gamma \vdash \varphi : \mathrm{PG}^p(z)}{\Gamma \vdash \forall x.\ \varphi : \mathrm{PG}^p(z)}\ \forall_{\mathrm{PG}}$$

(i) $\forall_{PG}$ rule



(ii) PDT

To enumerate the values of dc for which $\varphi''_{\mathsf{Grubbs}}$ is violated, we evaluate the closures. In the subtree marked with t_0, dc is equal to 0. We obtain $\mathtt{msg}(0) \in \mathbb{D} \setminus \{\mathtt{msg}(1)\}$ and t_0 reduces to $\top$. In the subtree marked with t_1, dc is equal to 1 and hence t_1 reduces to $\bot$. The formula is thus violated only for $v = \{dc \mapsto 1, l \mapsto 1\}$.

Enforceability. Our enforcement algorithm (Figure 5) does not terminate in general if functions are naïvely applied. Consider $\Box(\forall x.\ A(x) \longrightarrow A(x+1))$, where A is causable. If $A(i)$ occurs in the present, the algorithm causes $A(i+1)$, then $A(i+2)$, $A(i+3)$, etc. This formula would thus require infinitely many events to be caused once some $A(x)$ occurs. Hence, further restrictions must be introduced to define a fragment of extended EMFOTL that is realistically enforceable.

Key to these restrictions is the notion of a *stable function*:

Definition 5. *Let $\preceq$ be a well-founded relation on $\mathbb{D}$. A function $f : \mathbb{D}^k \to \mathbb{D}$ is* $\preceq$-stable *iff there exists a finite $C_f \subseteq \mathbb{D}$ such that for any $d_{\mathsf{sup}} \in \mathbb{D}$ and $d_1, \ldots, d_{a(f)} \preceq d_{\mathsf{sup}}$, either $f(d_1, \ldots, d_{a(f)}) \preceq d_{\mathsf{sup}}$ or $f(d_1, \ldots, d_{a(f)}) \in C_f$.*

A $\preceq$-stable function can only produce outputs that are smaller than one of its inputs with respect to some well-founded relation $\preceq$, or are in some finite set C_f. This guarantees that the number of *distinct* domain elements obtainable by repeatedly applying stable functions to an initial, finite set of domain elements is finite. For example, if $\mathbb{D} = \mathbb{N}$, then $f_1 = \lambda x.\ \max(x-1, 2)$ is $\leq$-stable, but $f_2 = \lambda x.\ x{+}1$ is not. Applying f_1 repeatedly to elements in a set $\{d_1, \ldots, d_k\} \subseteq \mathbb{N}$ only produces natural numbers in $\{0, \ldots, \max_{1 \leq i \leq k} d_i\}$ or the natural number 2, while applying f_2 repeatedly to $\{0\}$ reaches all of $\mathbb{N}$.

Formally, for $F \subseteq \mathbb{F}$, $X \subseteq \mathbb{D}$, and $n \geq 0$, define cl^n inductively as follows:

$$\mathsf{cl}^0(F, X) = X \qquad \forall i \geq 0.\ \mathsf{cl}^{i+1}(F, X) = X \cup \bigcup_{f \in F} f((\mathsf{cl}^i(F, X))^{a(f)}).$$

Further, define $\mathsf{cl}(F, X)$ as $\lim_{n\infty} \mathsf{cl}^n(F, X)$. We have:

Lemma 1. *$\mathsf{cl}(F, X)$ is finite for a finite set of stable functions F and a finite X.*

Back to our enforcement setup, if the parameters of all caused events are obtained by applying stable functions to existing domain elements, then only finitely many events may be caused and the enforcement algorithm terminates. In fact, we can be slightly more permissive: causation of events with parameters *not* obtained by applying stable functions is admissible as long as these parameters cannot be further used to derive parameters of caused events. Denoting by $\mathbb{F}_s$ the subset of all stable functions in $\mathbb{F}$, we get our final lemma:

Lemma 2. *Let $\overline{D} \in \mathbb{DB}^\omega$, $k \geq 1$, and disjoint $\mathbb{C}_s, \mathbb{C}_n \subseteq \mathbb{C}$ such that $\forall i \geq 2$,*

$$D_i - D_{i-1} \subseteq \{e(d_1, ..., d_{a(e)}) \mid e \in \mathbb{C} \land \forall i\, \exists f \in \mathsf{cl}(\mathbb{F}_s, D_{i-1}), \overline{d'} \in \mathsf{AD}_{D_i, \overline{\mathbb{C}_n}}(\varphi)^{a(f)}.\ d_i = \hat{f}(\overline{d'})\}$$
$$\cup \{e(d_1, ..., d_{a(e)}) \mid e \in \mathbb{C}_s \land \forall i\, \exists f \in \mathsf{cl}^k(\mathbb{F}, D_{i-1}), \overline{d'} \in \mathsf{AD}_{D_i, \overline{\mathbb{C}_n}}(\varphi)^{a(f)}.\ d_i = \hat{f}(\overline{d'})\},$$

where $\mathsf{AD}_{D_i, E}(\varphi) := \mathsf{AD}_{\langle (0, D_i) \rangle, E}(\varphi)$, then $\overline{D}$ is eventually constant.

This lemma ensures that if we can (i) partition the set of causable events $\mathbb{C}$ into two sets of *strict causable events* $\mathbb{C}_s$ and *nonstrict causable events* $\mathbb{C}_n$, (ii) ensure that the parameters of existing nonstrict causable events cannot be used to compute the parameters of newly caused events, and (iii) ensure that the parameters of newly caused, strict causable events are obtained from existing domain elements by applying only stable functions, then only finitely many new domain elements can be generated through causation. As a consequence, the enforcement loop $\mathsf{fp}(\sigma, X, \mathsf{enf}^+_{\mathsf{all}, \varphi, v, ts, b})$ in Figure 5 terminates.

To check (i)–(iii), we type event names to elements in $\{\mathbb{C}_n, \mathbb{C}_s, \mathbb{S}_n, \mathbb{S}_s\}$, rather than just $\{\mathbb{C}, \mathbb{S}\}$, and store additional typing judgments $x : \mathrm{PG}^+_E$ if the current value of x is the parameter of some event $e \in E$ in the past or present. The type lattice is modified as shown in Figure 7, with solid lines representing $\sqsubseteq$ (oriented bottom-up) and dotted lines representing an operator $\neg$ that exchanges causability and suppressability. We then replace the rules $\forall^{\mathbb{C}}$ from Example 1 by the rules in Figure 8, where $\mathbb{C}_\alpha$ matches $\mathbb{C}_s$ or $\mathbb{C}_n$ and $\mathsf{fn}(\varphi)$ denotes the set of all functions symbols in φ. All PG rules are updated with the subscript E.

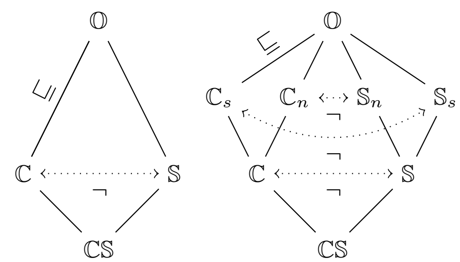


Fig. 7: Hublet et al.'s type lattice [25] (left) and our extended type lattice (right)

$$\frac{\Gamma \vdash \varphi : \tau' \quad \tau \sqsubseteq \tau'}{\Gamma \vdash \varphi : \tau}\,\mathsf{cast} \qquad \frac{\Gamma, x : \mathrm{PG}^+_E \vdash \varphi : \mathbb{C}_\alpha \quad \vdash \varphi : \mathrm{PG}^-_E(x)}{\Gamma \vdash \forall x.\, \varphi : \mathbb{C}_\alpha}\,\forall^{\mathbb{C}} \qquad \frac{\overline{t}_i = x}{\vdash e(\overline{t}) : \mathrm{PG}^+_{\{e\}}(x)}\,\mathbb{E}^+_{\mathrm{PG}}$$

$$\frac{\Gamma \vdash \varphi : \mathbb{C}_\alpha}{\Gamma \vdash \Box\, \varphi : \mathbb{C}_\alpha}\,\Box^{\mathbb{C}} \qquad \frac{a < \infty \quad \Gamma \vdash \varphi : \mathbb{C}_\alpha}{\Gamma \vdash \Diamond_{[0,a]}\, \varphi : \mathbb{C}_\alpha}\,\Diamond^{\mathbb{C}} \qquad \frac{\Gamma \vdash \psi : \mathbb{C}_\alpha}{\Gamma \vdash \varphi \longrightarrow \psi : \mathbb{C}_\alpha}\,\longrightarrow^{\mathbb{C}\mathrm{R}} \qquad \frac{\vdash \varphi : \mathrm{PG}^+_E(x)}{\vdash \varphi \longrightarrow \psi : \mathrm{PG}^-_E(x)}\,\longrightarrow^-_{\mathrm{PG}}$$

$$\frac{e \in \mathbb{C} \quad \Gamma(e) = \mathbb{C}_\alpha \quad \forall x \in \bigcup_{i=1}^k \mathsf{fv}(t_i).\, \exists E \subseteq \Gamma^{-1}(\overline{\mathbb{C}_n}).\, \Gamma(x) = \mathrm{PG}^+_E \quad \bigcup_{i=1}^k \mathsf{fn}(t_i) \subseteq \mathbb{F}_s}{\Gamma \vdash e(t_1, ..., t_k) : \Gamma(e)}\,\mathbb{E}^{\mathbb{C}_\alpha}$$

$$\frac{e \in \mathbb{C} \quad \Gamma(e) = \mathbb{C}_n \quad \forall x \in \bigcup_{i=1}^k \mathsf{fv}(t_i).\, \exists E \subseteq \Gamma^{-1}(\overline{\mathbb{C}_n}).\; \Gamma(x) = \mathrm{PG}^+_E}{\Gamma \vdash e(t_1, ..., t_k) : \mathbb{C}_n}\,\mathbb{E}^{\mathbb{C}_n}$$

Fig. 8: Selected modified typing rules for function applications (cf. Example 1)

Example 5. In $\varphi_{\mathsf{Grubbs}}$, the concatenation function ($\hat{\ }$) within the term in alert is not stable. However, $\varphi_{\mathsf{Grubbs}}$ is still enforceable by causing $\mathsf{alert}(\mathtt{msg}(dc))$ whenever $\varphi'_{\mathsf{Grubbs}}$ holds. In our type system, this is reflected by the fact that if alert types to $\mathbb{C}_n$ in Γ, the $\mathbb{E}^{\mathbb{C}_n}$ rule can be applied to derive $\Gamma \vdash \mathsf{alert}(\mathtt{msg}(dc)) : \mathbb{C}_n$. This rule accepts non-stable functions such as ($\hat{\ }$) in the argument of alert. However, it still requires some non-$\mathbb{C}_n$ event to guard the variable dc in the argument. The non-causable reboot event provides such a guard, as we show in Example 7.

In contrast, a formula such as $\Box(\forall x.\, \mathsf{alert}(x) \longrightarrow \mathsf{alert}(x \,\hat{\ }\, x))$ cannot be typed to $\mathbb{C}$ by causing $\mathsf{alert}(x \,\hat{\ }\, x)$: using alert as a guard for x precludes $\mathsf{alert} : \mathbb{C}_n$, but $\mathsf{alert} : \mathbb{C}_n$ would be required to cause the right-hand side as it contains ($\hat{\ }$).

Enforcement. With the additional restrictions that we just introduced and our extended monitor, the enforcement algorithm proposed by Hublet et al. [25, Algorithm 2] can be reused when function applications are introduced. The modified termination and correctness proofs rely on Lemma 2 (see Appendix A).

3.2 Aggregations

Assume that every aggregation operator $\omega \in \Omega$ is associated with a (terminating) function $\hat{\omega} : (\mathbb{D}^{a(\omega)_1})^* \to (\mathbb{D}^{a(\omega)_2})^*$ that maps a multiset of $a(\omega)_1$-tuples into a multiset of $a(\omega)_2$-tuples. Our semantics of MFOTL aggregations is as follows:

$$v, i \vDash_\sigma \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi \ \text{ iff } v(\overline{x}) \in \omega(M) \text{ where } \overline{z} = \mathsf{fv}(\varphi) \setminus \overline{y} \text{ and}$$
$$M = \left[[\![t]\!]_{v[\overline{z} \mapsto \overline{d}]} \mid v[\overline{z} \mapsto \overline{d}], i \vDash_\sigma \varphi, \overline{d} \in \mathbb{D}^{|\overline{z}|} \right] \text{ and } |\overline{y}| > 0 \text{ implies } M \neq [\ ],$$

where $v(\overline{x}) := (v(x_1), \ldots, v(x_{|x|}))$ and $[\![t]\!]_v := ([\![t_1]\!]_v, \ldots, [\![t_{|t|}]\!]_v)$. Note the last condition, which specifies that when there is at least one group variable, the aggregation is only satisfied when at least one valuation satisfies φ. A similar approach is followed in most SQL implementations: aggregation over an empty set without grouping returns a default value (such as 0 for sums), whereas aggregation over an empty set with grouping returns an empty result set. Our definition of aggregation generalizes over that of past monitoring tools [9] by supporting operators that return tuples, rather than a single value. Various algorithms (e.g., clustering algorithms) can thus be implemented as aggregation operators.

Monitorability. Monitoring an aggregation $\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi$, where t is a sequence of terms that may contain function applications, requires that the above set M is finite. Hence, there must exist only finitely many valuations of $\overline{z} := \mathsf{fv}(\varphi) \setminus \overline{y}$ satisfying φ. We modify Definition 4 accordingly.

Definition 6. *An MFOTL formula φ is monitorable iff the condition in Definition 4 holds, and, additionally, for any subformula $\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \psi$ of φ, we have $\vdash \psi : PG^+(z)$ for all variables $z \in \mathsf{fv}(\psi) \setminus \overline{y}$.*

Monitoring. We now show how to transform a PDT of φ into a PDT of $\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi$, imposing the following additional constraint on the PDT of φ:

2. All $\mathsf{LVar}\ y$ nodes with y in $\overline{y}$ appear above all $\mathsf{LVar}\ y'$ nodes with $y' \in \mathsf{fv}(\varphi) \setminus \overline{y}$.

This condition allows collecting values to be placed in the PDT *below* all nodes labeled with the group variables. Our algorithm (Figure 9) inputs $\overline{x}$, $\overline{t}$, and $\overline{y}$, a PDT *pdt* for φ, and a list $\overline{z}$ containing a linearization of the set $\overline{x} \cup \overline{y}$. The variable appearing in nodes of *pdt* are assumed to form, top-down, a subsequence of $\overline{z}$.

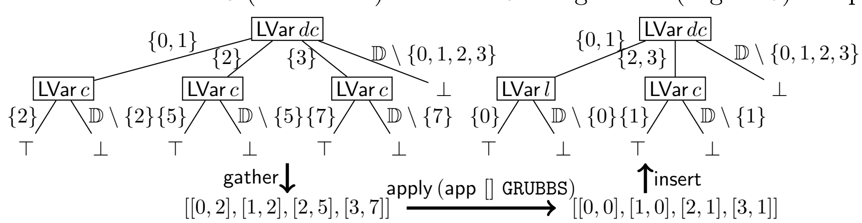
The algorithm proceeds in three steps, exemplified in Figure 10. First, the original PDT with Boolean leaves is transformed into a PDT with nodes in $\{\mathsf{LVar}\ y \mid y \in \overline{y}\}$ and leaves containing the multiset M. This is done using the gather function (l. 7–18) that uses standard $\mathsf{concat} : \mathsf{list}\,\mathsf{list}\, a \to \mathsf{list}\, a$ and $\mathsf{map} : (a \to b) \to \mathsf{list}\, a \to \mathsf{list}\, b$ functions as well as a function applyn that provides an analogue of apply2 for lists of PDTs. The function traverses the tree top-down, collecting constraints on the value of different variables and terms in a list sv. At the leaves, that list is converted into a list of satisfactions vs that are then used to compute all possible evaluations of $\overline{t}$. In a second step, the aggregation operator ω is applied at the leaves using apply to obtain a PDT with leaves carrying $\omega(M)$. The function agg (l. 19) wraps ω to map any empty multiset to None when $|\overline{y}| > 0$. Third and finally, this PDT is transformed into a Boolean PDT, inserting the new variables $\overline{x}$ at their correct position in $\overline{z}$ using insert (l. 20–29), which relies on a function all_leaves (see Appendix A) that gathers all elements stored in the leaves of a PDT. Being able to insert the $\overline{x}$ at any position is important, since the monitoring algorithm requires free variables in a PDT to be ordered according to their De Bruijn indices in the overall formula. We show:

Lemma 3. *Let $\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi$ be monitorable and $\overline{z} = \mathsf{fv}(\varphi) \setminus \overline{y}$. Let pdt be well-formed with respect to the bound variables in φ. Further assume that condition 2. above holds for pdt and that pdt stores $\textsc{Sat}_\varphi(\bullet, i, \sigma)$. Then* aggregate $\overline{x}\ \overline{t}\ \overline{y}\ \overline{z}$ *pdt stores $\textsc{Sat}_{\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi}(\bullet, i, \sigma)$.*

```
let distribute f x (D, pdt) = if |D| < ∞ then [({d}, f d pdt) | d ∈ D] else [(D, x)]
let tabulate t̄ sv vs = case sv of [ ] ⇒ [⟦t̄⟧_v | v ∈ vs]
    | (x, D) :: sv′ where x ∈ 𝕍 ⇒ tabulate t̄ sv′ [v[x ↦ d] | d ∈ D, v ∈ vs]
    | (t, D) :: sv′ ⇒ tabulate t̄ sv′ [v | v ∈ vs, ⟦t⟧_v ∈ D]
let gather sv t̄ ȳ pdt = let f t (D, pdt) = (D, gather (sv · (t, D)) t ȳ pdt) in
    case pdt of Leaf ℓ ⇒ if ℓ = ⊤ then Leaf (tabulate t̄ sv [∅]) else Leaf [ ]
    | Node (LVar x) parts ⇒  if x ∉ ȳ then applyn (∪) (map (f x) parts) else
        let g d pdt = gather {v[x ↦ d] | v ∈ vs} t̄ ȳ pdt in
        Node (LVar v) (concat (map (distribute g [ ]) parts)
    | Node (LEx x) parts ⇒ applyn (∪) (map (f x) parts)
    | Node (LAll x) parts ⇒ applyn (∩) (map (f x) parts)
    | Node (LClos h t̄ _) parts ⇒ applyn (∪) (map (h(t̄)) parts)
let agg ȳ ω M = if |ȳ| > 0 ∧ M = [ ] then None else ω M
let insert v x̄ z̄ pdt = case z̄, pdt of
      x :: z̄′, _ where x ∈ x̄ ⇒ let D = map (λv. v x) (all_leaves pdt) in
        if D = [ ] then Leaf ⊥
        else Node (LVar y, distribute (λd pdt. insert v[x ↦ d] x̄ z̄′ pdt) ⊥ (D, pdt))
    | y :: z̄′, Node (LVar y′, parts) where y = y′ ⇒
        Node (LVar y′, map (λ(D, pdt). (D, insert x z̄ pdt))) parts
    | _ :: z̄′, Node _ ⇒ insert v x̄ z̄′ pdt
    | _, Leaf (Some vs) ⇒ if ∃v′ ∈ vs. ∀x ∈ dom v. v x = v′ x then ⊤ else ⊥
    | _, Leaf None ⇒ ⊥
let aggregate ω x̄ t̄ ȳ z̄ pdt = insert ∅ x̄ z̄ (apply (agg ȳ ω) (gather [ ] t̄ ȳ pdt))
```

Fig. 9: Computing aggregations in PDTs

Example 6. In $\varphi_{\mathsf{Grubbs}}$, let cntReboots hold for $(dc, c) \in \{(0,2),(1,2),(2,5),(3,7)\}$. Assume that the GRUBBS function maps data centers 0 and 1 to cluster $l = 0$ and data centers 2 and 3 (as outliers) to $l = 1$. Our algorithm (Figure 9) computes:



Note that the intermediate PDTs are just leaves as there is no grouping variable.

Enforceability. Aggregations are generally not causable. Formula $\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi$ is suppressable iff $\overline{y}$ is non-empty and $\exists z_1, \ldots, z_k.\ \varphi$ is suppressable, where $\overline{z} = \mathsf{fv}(\varphi) \setminus \overline{y}$ (rule $\mathsf{agg}^{\mathbb{S}}$ in Figure 11). Aggregations can provide past-guardedness in two ways: $\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi$ types to $\mathrm{PG}^p(v)$ iff either (a) $v \in \overline{x}$, $p = +$, all free variables of $\overline{t}$ are past-guarded in φ, and the events used to guard these free variables are not used for causation in Γ (rule $\mathsf{agg}_{\mathrm{PG},\overline{x}}$) or (b) $v \in \overline{y}$ and v is past-guarded in f (rule $\mathsf{agg}_{\mathrm{PG},\overline{y}}$). The last condition in (a) means that Γ is now relevant for past-guardedness; it excludes non-enforceable formulae (e.g., $\forall x.\, x \leftarrow$

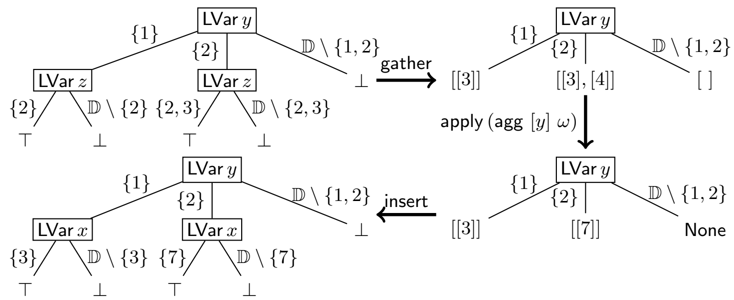


Fig. 10: Formula $x \leftarrow \mathtt{SUM}(z+1; y)\ A(y,z)$ with $D = \{A(1,2), A(2,2), A(2,3)\}$

$$\frac{\forall z \in \mathsf{fv}(\varphi) \setminus \overline{y}.\ \vdash \varphi : \mathrm{PG}(z)^{+}_{E_z} \quad \Gamma, \forall z.\ z : \mathrm{PG}^{+}_{E_z} \vdash \varphi : \mathbb{S}_\alpha \quad |\overline{y}| > 0}{\Gamma \vdash \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi : \mathbb{S}_\alpha}\ \mathsf{agg}^{\mathbb{S}}$$

$$\frac{v \in \overline{x} \quad \forall u \in \mathsf{fv}(\overline{t}).\ \exists E_u \subseteq \Gamma^{-1}(\overline{\mathbb{C}}).\ \Gamma \vdash \varphi : \mathrm{PG}^{+}_{E_u}(u)}{\Gamma \vdash \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi : \mathrm{PG}^{+}_{\bigcup_{u \in \mathsf{fv}(\overline{t})} E_u}}\ \mathsf{agg}_{\mathrm{PG},\overline{x}}$$

$$\frac{v \in \overline{y} \quad \Gamma \vdash \varphi : \mathrm{PG}^{p}_{E}(v)}{\Gamma \vdash \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi : \mathrm{PG}^{p}_{E}(v)}\ \mathsf{agg}_{\mathrm{PG},\overline{y}}$$

Fig. 11: Additional typing rules for aggregations

$\mathtt{SUM}(y;)A(y) \longrightarrow A(x))$. Other past-guardedness rules have the same Γ on the LHS of all of their sequents. The rules in Figure 11 are sound (Appendix A).

Enforcement. To support the suppression of aggregations as given by rule $\mathsf{agg}^{\mathbb{S}}$ above, an additional case is added to the function enf^{-}:

$$\mid \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi_1 \ \Rightarrow\ \mathsf{enf}^{+}_{ts,b}(\neg(\exists z_1, \ldots, z_k.\ \varphi_1), \sigma, X, v).$$

3.3 let bindings

We adopt the semantics of let bindings introduced by Zingg et al. [45]:

$$v, i \vDash_\sigma \mathsf{let}\, e(\overline{x}) = \varphi \,\mathsf{in}\, \psi \qquad \text{iff } v, i \vDash_{\sigma[e \Rrightarrow (\lambda i. \{\overline{d} \in \mathbb{D}^{|\overline{x}|} \mid v[\overline{x} \mapsto \overline{d}], i \vDash \varphi\})]} \psi.$$

where $\sigma[e \Rrightarrow R]$ denotes the trace obtained from σ by adding, at each time-point i, all events $e(\overline{d})$ such that $\overline{d} \in R(i)$. With this semantics, let bindings can be soundly unrolled by substituting every occurrence of $e(\overline{t})$ in ψ with $\varphi[\overline{x} \mapsto \overline{t}]$. The enforcement algorithm requires no extension if unrolling is performed prior to typing and enforcement. In fact, with memoization (Section 4) such unrolling should not lead to any significant runtime overhead.

When applied naïvely after unrolling, type inference for the enforcement type system becomes prohibitively slow. To avoid this issue, we introduce the typing rules in Figure 12, proved sound in Appendix A. The rule let allows φ_1's enforceability type to be reused in φ_2. Additionally, it extends Γ with judgments of the form $\mathsf{let}_e : \bot$ and $\mathsf{let}_{e,i,p} : E$ denoting the existence of a let-bound predicate e and past-guardedness of e's ith argument, respectively. The $\mathsf{let}_{\mathrm{PG}}$ rule extracts past-guardedness information for let-bound predicates from Γ.

$$\dfrac{\mathsf{let}_e \in \operatorname{dom} \Gamma \quad \Gamma(\mathsf{let}_{e,i,p}) = E \quad \bar{t}_i = x}{\Gamma \vdash e(\bar{t}) : \mathrm{PG}^p_E(x)}\ \mathsf{let}_{\mathrm{PG}}$$

$$\dfrac{\Gamma \vdash \varphi_1 : \tau_1 \qquad \Gamma \cup \{\mathsf{let}_{e,i,p} : E \mid \Gamma \vdash \varphi_1 : \mathrm{PG}^p_E(x_i)\}, \mathsf{let}_e : \bot, e : \tau_1 \vdash \varphi_2 : \tau_2}{\Gamma \vdash \mathsf{let}\, e(x_1, \ldots, x_k) = \varphi_1 \ \mathsf{in}\ \varphi_2 : \tau_2}\ \mathsf{let}$$

Fig. 12: Additional typing rules for let bindings

The full typing of the formula in Section 1 is given in Appendix B.

Example 7. Rule $\mathsf{agg}_{\mathrm{PG},\overline{x}}$ proves that dc is past-guarded by cntReboots in $\varphi''_{\mathsf{Grubbs}}$ if cntReboots is not in $\mathbb{C}$. It also proves that dc is past-guarded by badReboot in $c \leftarrow \mathtt{CNT}(i; dc)(\blacklozenge_{[0,1800)}(\mathsf{badReboot}(s, dc) \wedge \mathsf{tp}(i)))$ if badReboot is not in $\mathbb{C}$. Note that dc is past-guarded by reboot in $\mathsf{reboot}(s, dc) \wedge \neg \bullet (\neg \mathsf{reboot}(s, dc)\ \mathsf{S}\ \mathsf{intendReboot}(s, dc))$. We can then use let, $\mathsf{let}_{\mathrm{PG}}$, and the past-guardedness facts established above to show that dc is past-guarded by reboot in $\varphi''_{\mathsf{Grubbs}}$.

Theorem 1. *Let φ be a closed EMFOTL formula with function applications, aggregations, and* let *bindings. Let* enf' *be the extended* enf *function. Denote* $\mathsf{unroll}(\varphi)$ *the formula obtained by unrolling* let *in φ. Then the enforcer* $\mathcal{E}_\varphi = (\mathcal{P}(\mathsf{fo}), \{(\mathsf{unroll}(\varphi), \emptyset, +)\}, \mathsf{enf}')$ *is sound with respect to* $\mathcal{L}(\varphi)$.

We also prove $\mathcal{E}_\varphi$'s transparency for a fragment of EMFOTL in Appendix A.

4 Implementation and Optimizations

We have implemented our extensions in an open-source tool, called EnfGuard (available at [26]), consisting of about 11,000 lines of OCaml code. To ease code reuse, all MFOTL-related function are packaged into a separate library.

EnfGuard support two types of functions: built-in functions, such as arithmetic operations, and user-defined functions. In addition to SQL-style aggregations, EnfGuard also supports user-defined aggregations. User-defined functions and aggregations are provided by the user in a Python file. The user must specify each function's signature and whether it is stable, and ensure that it terminates. The enforcer calls Python functions via the `pyml` bindings during monitoring. Support for Python functions makes EnfGuard more easily extendable.

EnfGuard's implementation includes three main optimizations:

Associative and commutative (AC) rewriting. Multiple binary conjunctions and disjunctions are replaced by n-ary ones and standard AC-rewriting is applied before enforcement starts. When enforcing an n-ary operator, the enforcement algorithm is called only once on each conjunct or disjunct inside the fixpoint computation, which exponentially reduces the number of calls in the best case.

Memoization. When the trace changes due to causation or suppression, a naïve algorithm drops the previously computed truth values and recomputes new ones. Given φ, we compute the set of *relevant event names* $\mathsf{RE}(\varphi)$ and *relevant future obligations* $\mathsf{RFO}(\varphi)$ that can affect the truth value of φ under assumptions (see Appendix C). When enforcement causes new events D^+ or future obligations O, we compute the sets $\{e \mid e(\overline{v}) \in D^+\} \cap \mathsf{RE}(\varphi)$ and $O \cap \mathsf{RFO}(\varphi)$ first. If both are empty, the previous verdict is still valid and can be returned.

Subformulae skipping. Our algorithm does not evaluate subformulae known to be true whenever certain event names do not presently exist. For every subformula φ, we precompute the *present filter* $f_\varphi := \mathfrak{F}_\top(\varphi)$ such that

$$\begin{aligned} \mathfrak{F}_b(\top) &= \lambda D.\ b & \mathfrak{F}_\top(e(\bar{t})) &= \lambda D.\ \exists \bar{t}.\ e(\bar{t}) \in D \\ \mathfrak{F}_b(\neg\varphi) &= \mathfrak{F}_{\neg b}(\varphi) & \mathfrak{F}_\top(\varphi \wedge \psi) &= \lambda D.\ \mathfrak{F}_\top(\varphi)(D) \wedge \mathfrak{F}_\top(\psi)(D) \\ \mathfrak{F}_b(\exists x.\ \varphi) &= \mathfrak{F}_b(\varphi) & \mathfrak{F}_\bot(\varphi \wedge \psi) &= \lambda D.\ \mathfrak{F}_\bot(\varphi)(D) \vee \mathfrak{F}_\bot(\psi)(D) \\ \mathfrak{F}_b(\varphi) &= \lambda D.\ \top & &\text{for any } \varphi = \bullet_I\, \psi, \bigcirc_I\, \psi, \psi_1\ \mathsf{U}_I\ \psi_2, \psi_1\ \mathsf{S}_I\ \psi_2. \end{aligned}$$

Whenever $f_\varphi(D)$ evaluates to false on the current database, we immediately return without causing or suppressing any events.

5 Evaluation

Our evaluation of EnfGuard answers the following research questions:
RQ1. Can EnfGuard's EMFOTL fragment formalize real-world policies?
RQ2. At what event rates can EnfGuard perform real-time enforcement?
RQ3. Does EnfGuard's performance improve upon the state-of-the-art?

To evaluate EnfGuard, we introduce what is, to the best of our knowledge, the largest set of runtime enforcement benchmarks to date. We first present these benchmarks (Section 5.1) and then report on our results (Section 5.2).

5.1 Benchmarks and evaluation setup

We use six benchmarks, each of which pairs a set of policies and a set of logs:

- GDPR: 6 formulae encoding privacy policies and a log of a job application system produced over a period of a year [3,25].
- GPDR$^{\text{FUN}}$: Variants of the six GDPR formulae that use custom Python functions to store and look up data ownership and consent, with the same log.
- NOKIA: 11 formulae encoding data usage policies of a distributed system used in Nokia's mobile data collection campaign [7] and a log of this system [28] spanning one day. The system's original event rate was about 100 events/s.
- IC: 8 formulae encoding various policies of a large Web3 distributed platform [43] and 3 platform execution logs [6] having 100–150 events/s.
- AGG: 6 fraud detection formulae [8] using aggregations and 2 synthetic logs.
- CLUSTER: 2 outlier detection formulae using aggregation operators implemented in Python and 3 synthetic logs.

Figure 13 shows benchmark statistics. For each benchmark, we report the number of formulae and logs, the maximal formula size (defined as its number of operators without unrolling let), the maximal log size (defined as its number of events), and the maximum log event rate (defined as the average number of events per second of real-time execution). We also indicate whether the formulae use let bindings (Let), aggregations (Agg.), and function applications (Fun.), possibly defined in Python (). Appendix D lists all formulae used.

In this evaluation, we compare EnfGuard to three tools: EnfPoly [24] and WhyEnf [25], the only existing MFOTL enforcement tools, and MonPoly [9],

	Log statistics					Formulae statistics					Tool support			
Name	Source	Real	#logs	max |log|	max er	max $\|\varphi\|$	let bindings	Aggreg.	Functions	#formulae	EnfGuard	WhyEnf	EnfPoly	MonPoly
GDPR	[3,25]	✓	1	5,631	10^{-4}	72				6	6	6	2	6
GPDR$^{\text{FUN}}$	[3,25]	✓	1	5,631	10^{-4}	108				6	6			
NOKIA	[28,7]	✓	1	9,458,824	109	44			✓	11	11	11	5	11
IC	[6]	✓	3	634,789	147	179	✓		✓	8	8			8
AGG	[8]		2	100,000		34		✓	✓	6	6			6
CLUSTER	new		1	5,000		42	✓		✓	2	2			
									Total:	39	39	17	7	31
						Rewriting required:					no	no	yes	yes

Fig. 13: Benchmarks' logs (left), formulae (middle), and tool support (right)

a state-of-the-art MFOTL monitor with aggregations [8], let bindings [45], and built-in functions. As monitoring is a simpler task than enforcement, MonPoly's performance is intended to suggest the likely 'best achievable' results for comparable expressivity, rather than a standard to achieve. All measurements are performed on an AMD Ryzen™ 5 5600X (6 cores) with 16 GB RAM.

5.2 Results

We now present the results of our experiments and answer the research questions.

RQ1: Expressiveness. Figure 13 (right) shows the number of policies each tool supports across all benchmarks. EnfGuard supports all 39 policies, whereas MonPoly supports 31 formulae (all except those containing user-defined constructs), but requires manual rewriting of formulae into its monitorable fragment. WhyEnf and EnfPoly support just 17 and 7 policies, respectively. Both tools cannot enforce formulae with function applications, aggregations, or let bindings. Without let, formulae can become much larger (up to 20 times in practical examples [6]) and difficult to read and maintain. Aggregations strictly increase the policy language's expressiveness [21]: some requirements [6,8] cannot be expressed without them. EnfPoly is additionally restricted to past-only policies.

RQ2: Maximum event rate. Figure 14 shows each tool's average latency ($\mathsf{avg}_\ell(a)$, in ms), maximum latency ($\mathsf{max}_\ell(a)$, in ms) and average event rate avg_{er} for the largest trace acceleration $a \in \{2^0, \ldots, 2^9\}$ such that $\mathsf{max}_\ell(a) \leq \frac{1}{a}$. A trace acceleration is the ratio between the speed that a trace is provided to the enforcer and the trace's real-time behavior (captured by its timestamps). The inequality captures that latency is smaller than the interval between two timestamps in the accelerated trace, i.e., that a tool can process the trace in real time. We report averages over 5 repetitions of each benchmark's largest log.

Except for one formula in IC, EnfGuard can enforce all policies in real time, with event rates ranging from 20–200 events/s when frequent aggregation and causation is involved (AGG, CLUSTER, some of IC) to over 1,000–14,000 events/s in contexts when few commands are emitted and policies are simpler (GDPR, NOKIA). Our experiments show maximum latency values below 20 ms in most cases, and below 100 ms in all but 4 benchmarks using commodity hardware.

			EnfGuard				WhyEnf				EnfPoly				MonPoly			
	Policy φ	$\lvert\varphi\rvert$	a	avg$_{er}$	avg$_\ell$	max$_\ell$	a	avg$_{er}$	avg$_\ell$	max$_\ell$	a	avg$_{er}$	avg$_\ell$	max$_\ell$	a	avg$_{er}$	avg$_\ell$	max$_\ell$
GDPR	consent	22	12.8e6	1619	.39	2	.8e6	101	7.6	30	51.2e6	6480	.17	1	51.2e6	6934	.20	1
	deletion	14	25.6e6	3238	.28	2	25.6e6	3238	.20	1					51.2e6	6934	.20	1
	gdpr	72	6.4e6	810	.87	3	.2e6	25	33	110					25.6e6	3465	.13	1
	information	16	12.8e6	1619	.33	2	6.4e6	810	1.1	5.2					51.2e6	6934	.15	1
	lawfulness	17	12.8e6	1619	.35	2	6.4e6	810	1.3	4.4	51.2e6	6480	.17	1	51.2e6	6934	.15	1
	sharing	19	12.8e6	1619	.32	2	3.2e6	405	3.0	15					51.2e6	6934	.20	1
NOKIA	del-1-2	37	32	3503	5	19	not real-time								128	14035	.21	5
	del-2-3	20	128	14013	.58	6	256	28026	.26	2					512	56139	.17	1
	del-3-2	20	128	14013	.55	6	512	56052	.26	2					512	56139	.17	1
	delete	10	128	14013	.54	5	256	28026	.25	2	512	56052	.16	1	512	56138	.17	1
	ins-1-2	25	64	7007	1.1	11	error†								not real-time			
	ins-2-3	20	32	3053	1.5	23	error†								32	3509	2.8	19
	ins-3-2	20	32	3503	5.9	29	256	28026	.28	2					256	28069	.40	3
	insert	10	128	14013	.65	7	256	28026	.26	2	512	56052	.22	2	512	56139	.21	1
	script1	44	128	14013	.64	6	256	28026	.28	2	512	56052	.19	1	512	56139	.24	1
	select	13	128	14013	.54	5	256	28026	.25	2	512	56052	.16	1	512	56139	.16	1
	update	8	128	14013	.53	6	256	28026	.24	2	512	56052	.16	1	512	56139	.16	1

			EnfGuard				MonPoly			
	Policy φ	$\lvert\varphi\rvert$	a	avg$_{er}$	avg$_\ell$	max$_\ell$	a	avg$_{er}$	avg$_\ell$	max$_\ell$
IC	validation	166	128	3744	.26	5	256	7489	.36	4
	clean_logs	48	2	59	2.7	281	128	3744	.14	3
	finalization	58	not real-time				128	3744	.14	3
	divergence	50	128	3744	.23	3	128	3744	.19	3
	height	162	128	3744	.24	3	not real-time			
	logging	179	64	1872	.23	10	2	59	.25	381
	reboot	79	2	59	2.4	276	128	3744	.16	3
	unauthorized	64	128	3744	.23	3	2	59	3.0	300
AGG	p1	21	64	640	5.1	9.4	512	5120	.16	1
	p2	22	32	320	13	27	512	5120	.33	1
	p3	27	8	80	44	102	512	5120	.39	1
	p4	31	2	20	54	392	512	5120	.48	1
	p5	32	64	640	6.3	11	512	5120	.25	1
	p6	34	64	640	6.8	12	512	5120	.31	1

			EnfGuard			
	Policy φ	$\lvert\varphi\rvert$	a	avg$_{er}$	avg$_\ell$	max$_\ell$
GDPR$^{\text{FUN}}$	fconsent	25	12.8e6	1619	.30	2
	fmanagement	22	25.6e6	1619	.31	2
	fdeletion	17	25.6e6	3238	.30	2
	fgdpr	108	6.4e6	3238	.93	4
	finformation	23	12.8e6	1619	.44	3
	fsharing	20	12.8e6	1619	.32	2
CL.	dbscan	42	32	160	17	31
	grubbs	42	32	160	14	32

† The tool returns incorrect results on test cases. The formula is not correctly enforced.

Fig. 14: Latency and processing time for the largest a such that $\mathsf{max}_\ell(a) \leq 1/a$.

RQ3: Comparison with the state-of-the-art. Our comparison on the GDPR benchmarks shows EnfGuard to be 1.5–30× faster than WhyEnf and up to 4 times slower than the much less expressive, table-based EnfPoly. Likely due to its more complex data structures, EnfGuard is sometimes slower than WhyEnf on small formulae (NOKIA), but with a latency still below 10 ms. The large gdpr formula exhibits EnfGuard's performance advantage over WhyEnf: while WhyEnf, with an event rate of only 25, suffers a significant slowdown compared to the same benchmark's other formulae, EnfGuard is still able to process 810 events per second. The comparison with MonPoly reveals potential for further optimizations, especially for aggregations (AGG). However, the performance gap between EnfGuard and MonPoly is smaller for large formulae (IC), with the two tools showing incomparable performance on complex formulae.

6 Conclusions and Future Work

We presented EnfGuard, the first proactive enforcement tool for rich policies written in metric first-order temporal logic with function applications, aggregations, and let bindings. Our evaluation shows that EnfGuard can be used in many real-world systems, like Web3, data management, or financial systems.

In future, we will further optimize EnfGuard to benefit from MonPoly's efficient table-based approach on a subset of EnfGuard's policy language.

Acknowledgments. Hublet is supported by the Swiss National Science Foundation grant "Model-driven Security & Privacy" (204796). Lima and Traytel are supported by a Novo Nordisk Fonden start package grant (NNF20OC0063462). We thank the anonymous CAV reviewers for their insightful feedback.

Disclosure of interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aceto, L., Cassar, I., Francalanza, A., Ingolfsdottir, A.: Bidirectional runtime enforcement of first-order branching-time properties. Logical Methods in Computer Science **19** (2023)
2. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: On first-order runtime enforcement of branching-time properties. Acta Informatica pp. 1–67 (2023)
3. Arfelt, E., Basin, D., Debois, S.: Monitoring the GDPR. In: Sako, K., Schneider, S.A., Ryan, P.Y.A. (eds.) 24th European Symposium on Research in Computer Security (ESORICS). LNCS, vol. 11735, pp. 681–699. Springer (2019)
4. Basin, D., Dardinier, T., Hauser, N., Heimes, L., Huerta y Munive, J.J., Kaletsch, N., Krstić, S., Marsicano, E., Raszyk, M., Schneider, J., et al.: Verimon: a formally verified monitoring tool. In: International Colloquium on Theoretical Aspects of Computing. pp. 1–6. Springer (2022)
5. Basin, D., Debois, S., Hildebrandt, T.: Proactive enforcement of provisions and obligations. J. Comput. Secur. **32**(3), 247–289 (2024)
6. Basin, D., Dietiker, D.S., Krstić, S., Pignolet, Y.A., Raszyk, M., Schneider, J., Ter-Gabrielyan, A.: Monitoring the internet computer. In: International Symposium on Formal Methods. pp. 383–402. Springer (2023)
7. Basin, D., Harvan, M., Klaedtke, F., Zalinescu, E.: Monitoring data usage in distributed systems. IEEE Transactions on Software Engineering **39**(10), 1403–1426 (2013)
8. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal methods in system design **46**, 262–285 (2015)
9. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. Journal of the ACM (JACM) **62**(2), 1–45 (2015)
10. Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. In: Workshop on Foundations of Computer Security (FCS). Citeseer (2002)
11. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K., Lime, D.: UPPAAL-Tiga: Time for playing games! In: Damm, W., Hermanns, H. (eds.) International Conference Computer Aided Verification (CAV). LNCS, vol. 4590, pp. 121–125. Springer (2007)
12. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Transactions on Database Systems (TODS) **20**(2), 149–186 (1995)
13. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6605, pp. 272–275. Springer (2011)

14. Erlingsson, Ú., Schneider, F.: SASI enforcement of security policies: a retrospective. In: Kienzle, D., Zurko, M.E., Greenwald, S., Serbau, C. (eds.) Workshop on New Security Paradigms. pp. 87–95. ACM (1999)
15. Falcone, Y., Jéron, T., Marchand, H., Pinisetty, S.: Runtime enforcement of regular timed properties by suppressing and delaying events. Science of Computer Programming **123**, 2–41 (2016)
16. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. **23**(2), 255–284 (2021)
17. Falcone, Y., Pinisetty, S.: On the runtime enforcement of timed properties. In: Finkbeiner, B., Mariani, L. (eds.) 19th International Conference on Runtime Verification, (RV). LNCS, vol. 11757, pp. 48–69. Springer (2019)
18. Fredrikson, M., Joiner, R., Jha, S., Reps, T.W., Porras, P.A., Saïdi, H., Yegneswaran, V.: Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 548–563. Springer (2012)
19. Grubbs, F.E.: Sample criteria for testing outlying observations. Ann. Math. Statist. **21**(4), 27–58 (1950)
20. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. **5**(2), 192–206 (2012)
21. Hella, L., Libkin, L., Nurmonen, J., Wong, L.: Logics with aggregate operators. J. ACM **48**(4), 880–907 (2001). https://doi.org/10.1145/502090.502100
22. Hildebrandt, T., Mukkamala, R.R., Slaats, T., Zanitti, F.: Contracts for cross-organizational workflows as timed dynamic condition response graphs. The Journal of Logic and Algebraic Programming **82**(5-7), 164–185 (2013)
23. Hofmann, T., Schupp, S.: TACoS: A tool for MTL controller synthesis. In: Calinescu, R., Pasareanu, C.S. (eds.) International Conference on Software Engineering and Formal Methods (SEFM). LNCS, vol. 13085, pp. 372–379. Springer (2021)
24. Hublet, F., Basin, D., Krstić, S.: Real-time policy enforcement with metric first-order temporal logic. In: European Symposium on Research in Computer Security. pp. 211–232. Springer (2022)
25. Hublet, F., Lima, L., Basin, D., Krstić, S., Traytel, D.: Proactive real-time first-order enforcement. In: International Conference on Computer Aided Verification. pp. 156–181. Springer (2024)
26. Hublet, François and Lima, Leonardo and Basin, David and Krstić, Srđan and Traytel, Dmitriy: EnfGuard (2025), https://github.com/runtime-enforcement/enfguard
27. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: International Conference Formal Methods in Computer-Aided Design (FMCAD). pp. 117–124. IEEE (2006)
28. Kiukkonen, N., Blom, J., Dousse, O., Gatica-Perez, D., Laurila, J.: Towards rich mobile phone datasets: Lausanne data collection campaign. Proc. ICPS, Berlin **68**(7) (2010)
29. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods Syst. Des. **19**(3), 291–314 (2001). https://doi.org/10.1023/A:1011254632723, https://doi.org/10.1023/A:1011254632723
30. Li, G., Jensen, P., Larsen, K., Legay, A., Poulsen, D.: Practical controller synthesis for $\mathrm{MTL}_{0,\infty}$. In: Erdogmus, H., Havelund, K. (eds.) ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 102–111. ACM (2017)
31. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for runtime security policies. International Journal of Information Security **4**, 2–16 (2005)

32. Lima, L., Herasimau, A., Raszyk, M., Traytel, D., Yuan, S.: Explainable online monitoring of metric temporal logic. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 473–491. Springer (2023)
33. Lima, L., Huerta y Munive, J.J., Traytel, D.: Explainable online monitoring of metric first-order temporal logic. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 288–307. Springer (2024)
34. Minato, S.i.: Binary decision diagrams and applications for VLSI CAD, vol. 342. Springer Science & Business Media (1995)
35. Ngo, M., Massacci, F., Milushev, D., Piessens, F.: Runtime enforcement of security policies on black box reactive programs. In: Rajamani, S.K., Walker, D. (eds.) 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 43–54. ACM (2015)
36. Peter, H., Ehlers, R., Mattmüller, R.: Synthia: Verification and synthesis for timed automata. In: Gopalakrishnan, G., Qadeer, S. (eds.) International Conference on Computer Aided Verification (CAV). LNCS, vol. 6806, pp. 649–655. Springer (2011)
37. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H.: TiPEX: A tool chain for timed property enforcement during execution. In: International Conference on Runtime Verification (RV). pp. 306–320. Springer (2015)
38. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena Timo, O.: Runtime enforcement of timed properties revisited. Formal Methods Syst. Des. **45**, 381–422 (2014)
39. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H.: Predictive runtime enforcement. Formal Methods Syst. Des. **51**(1), 154–199 (2017)
40. Raszyk, M., Basin, D., Krstić, S., Traytel, D.: Efficient evaluation of arbitrary relational calculus queries. Logical Methods in Computer Science **19** (2023)
41. Renard, M., Rollet, A., Falcone, Y.: GREP: games for the runtime enforcement of properties. In: Yevtushenko, N., Cavalli, A., Yenigün, H. (eds.) International Conference on Testing Software and Systems (ICTSS). LNCS, vol. 10533, pp. 259–275. Springer (2017)
42. Schneider, F.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000)
43. The DFINITY Team: The Internet Computer for geeks. Cryptology ePrint Archive, Paper 2022/087 (2022), https://eprint.iacr.org/2022/087
44. Zhu, S., Tabajara, L., Li, J., Pu, G., Vardi, M.: A symbolic approach to safety LTL synthesis. In: Strichman, O., Tzoref-Brill, R. (eds.) International Haifa Verification Conference (HVC). LNCS, vol. 10629, pp. 147–162. Springer (2017)
45. Zingg, S., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: Verified first-order monitoring with recursive rules. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 236–253. Springer (2022)

A Additional Definitions and Proofs

A.1 Past-guarded fragment

We defined the *extended active domain* $\mathsf{AD}^*_{\sigma,E}(\varphi)$ as $\mathsf{AD}^*_{\sigma,E}(\varphi) := \{0\} \cup \mathsf{cl}^{\delta(\varphi)}(\Omega, \mathsf{AD}_{\sigma,E}(\varphi))$, where $\delta(\varphi)$ is the maximum depth of nested aggregations in φ.

$$\frac{\bar{t}_i = x}{\vdash e(\bar{t}) : \mathrm{PG}(x)^+}\ \mathbb{E}^+_{\mathrm{PG}} \quad \frac{\vdash \varphi : \mathrm{PG}(x)^{\neg p}}{\vdash \neg\varphi : \mathrm{PG}(x)^p}\ \neg_{\mathrm{PG}} \quad \frac{x \neq z \quad \vdash \varphi : \mathrm{PG}(z)^p}{\vdash \exists x.\ \varphi : \mathrm{PG}(z)^p}\ \exists_{\mathrm{PG}}$$

$$\frac{\vdash \varphi : \mathrm{PG}(x)^+}{\vdash \varphi \wedge \psi : \mathrm{PG}(x)^+}\ \wedge^{\mathrm{L}+}_{\mathrm{PG}} \quad \frac{\vdash \psi : \mathrm{PG}(x)^+}{\vdash \varphi \wedge \psi : \mathrm{PG}(x)^+}\ \wedge^{\mathrm{R}+}_{\mathrm{PG}} \quad \frac{\vdash \varphi : \mathrm{PG}(x)^- \quad \vdash \psi : \mathrm{PG}(x)^-}{\vdash \varphi \wedge \psi : \mathrm{PG}(x)^-}\ \wedge^-_{\mathrm{PG}}$$

$$\frac{0 \notin I \quad \vdash \varphi : \mathrm{PG}(x)^+}{\vdash \varphi\, \mathsf{S}_I\, \psi : \mathrm{PG}(x)^+}\ \mathsf{S}^{\mathrm{L}+}_{\mathrm{PG}} \quad \frac{\vdash \psi : \mathrm{PG}(x)^+}{\vdash \varphi\, \mathsf{S}_I\, \psi : \mathrm{PG}(x)^+}\ \mathsf{S}^{\mathrm{R}+}_{\mathrm{PG}} \quad \frac{0 \in I \quad \vdash \psi : \mathrm{PG}(x)^-}{\vdash \varphi\, \mathsf{S}_I\, \psi : \mathrm{PG}(x)^-}\ \mathsf{S}^-_{\mathrm{PG}}$$

$$\frac{0 \notin I \quad \vdash \varphi : \mathrm{PG}(x)^+}{\vdash \varphi\, \mathsf{U}_I\, \psi : \mathrm{PG}(x)^+}\ \mathsf{U}^{\mathrm{L}+}_{\mathrm{PG}} \quad \frac{\vdash \varphi : \mathrm{PG}(x)^+ \quad \vdash \psi : \mathrm{PG}(x)^+}{\vdash \varphi\, \mathsf{U}_I\, \psi : \mathrm{PG}(x)^+}\ \mathsf{U}^{\mathrm{LR}+}_{\mathrm{PG}}$$

Past-guardedness

$$\frac{0 \in I \quad \vdash \psi : \mathrm{PG}(x)^-}{\vdash \varphi\, \mathsf{U}_I\, \psi : \mathrm{PG}(x)^-}\ \mathsf{U}^-_{\mathrm{PG}} \quad \frac{\vdash \varphi : \mathrm{PG}(x)^+}{\vdash \bullet_I\, \varphi : \mathrm{PG}(x)^+}\ \bullet^+_{\mathrm{PG}}$$

$$\frac{}{\Gamma \vdash \top : \mathbb{C}}\ \top^{\mathbb{C}} \quad \frac{}{\Gamma \vdash \bot : \mathbb{S}}\ \bot^{\mathbb{S}} \quad \frac{e \in \mathbb{C} \quad \Gamma(e) = \mathbb{C}}{\Gamma \vdash e(t_1, \ldots, t_k) : \mathbb{C}}\ \mathbb{E}^{\mathbb{C}} \quad \frac{e \in \mathbb{S} \quad \Gamma(e) = \mathbb{S}}{\Gamma \vdash e(t_1, \ldots, t_k) : \mathbb{S}}\ \mathbb{E}^{\mathbb{S}}$$

$$\frac{\Gamma \vdash \varphi : \mathbb{S}}{\Gamma \vdash \neg\varphi : \mathbb{C}}\ \neg^{\mathbb{C}} \quad \frac{\Gamma \vdash \varphi : \mathbb{C}}{\Gamma \vdash \neg\varphi : \mathbb{S}}\ \neg^{\mathbb{S}} \quad \frac{\Gamma \vdash \varphi : \mathbb{C}}{\Gamma \vdash \exists x.\ \varphi : \mathbb{C}}\ \exists^{\mathbb{C}} \quad \frac{\Gamma \vdash \varphi : \mathbb{S} \quad \vdash \varphi : \mathrm{PG}(x)^+}{\Gamma \vdash \exists x.\ \varphi : \mathbb{S}}\ \exists^{\mathbb{S}}$$

$$\frac{\Gamma \vdash \varphi : \mathbb{C} \quad \Gamma \vdash \psi : \mathbb{C}}{\Gamma \vdash \varphi \wedge \psi : \mathbb{C}}\ \wedge^{\mathbb{C}} \quad \frac{\Gamma \vdash \varphi : \mathbb{S}}{\Gamma \vdash \varphi \wedge \psi : \mathbb{S}}\ \wedge^{\mathbb{S}\mathrm{L}} \quad \frac{\Gamma \vdash \psi : \mathbb{S}}{\Gamma \vdash \varphi \wedge \psi : \mathbb{S}}\ \wedge^{\mathbb{S}\mathrm{R}}$$

$$\frac{0 \in I \quad \Gamma \vdash \psi : \mathbb{C}}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathbb{C}}\ \mathsf{S}^{\mathbb{C}} \quad \frac{0 \notin I \quad \Gamma \vdash \varphi : \mathbb{S}}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathbb{S}}\ \mathsf{S}^{\mathbb{S}\mathrm{L}} \quad \frac{0 \in I \quad \Gamma \vdash \varphi : \mathbb{S} \quad \Gamma \vdash \psi : \mathbb{S}}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathbb{S}}\ \mathsf{S}^{\mathbb{S}\mathrm{LR}}$$

$$\frac{\Gamma \vdash \psi : \mathbb{S}}{\Gamma \vdash \varphi\, \mathsf{U}_I\, \psi : \mathbb{S}}\ \mathsf{U}^{\mathbb{S}} \quad \frac{b \neq \infty \quad \Gamma \vdash \psi : \mathbb{C}}{\Gamma \vdash \varphi\, \mathsf{U}_{[0,b]}\, \psi : \mathbb{C}}\ \mathsf{U}^{\mathbb{C}\mathrm{R}} \quad \frac{b \neq \infty \quad \Gamma \vdash \varphi : \mathbb{C} \quad \Gamma \vdash \psi : \mathbb{C}}{\Gamma \vdash \varphi\, \mathsf{U}_{[a,b]}\, \psi : \mathbb{C}}\ \mathsf{U}^{\mathbb{C}\mathrm{LR}}$$

Typing of formulae as causable/suppressable

$$\frac{\Gamma \vdash \varphi : \mathbb{C} \quad b > 0}{\Gamma \vdash \bigcirc_{[0,b)}\, \varphi : \mathbb{C}}\ \bigcirc^{\mathbb{C}} \quad \frac{\Gamma \vdash \varphi : \mathbb{S}}{\Gamma \vdash \bigcirc_I\, \varphi : \mathbb{S}}\ \bigcirc^{\mathbb{S}}$$

Fig. 15: Typing rules for EMFOTL from Hublet et al. [25, Section 4]

$$\frac{\mathsf{let}_e \notin \mathrm{dom}\,\Gamma}{\Gamma \vdash e(t_1, \ldots, t_i = x, \ldots, t_k) : \mathrm{PG}^{+}_{\{e\}}(x)}\ \mathbb{E}^{+}_{\mathrm{PG}} \qquad \frac{}{\Gamma \vdash x = c : \mathrm{PG}^{+}_{\emptyset}(x)}\ =^{+}_{\mathrm{PG}}$$

$$\frac{\Gamma \vdash \varphi : \mathrm{PG}^{\neg p}_{E}(x)}{\Gamma \vdash \neg\varphi : \mathrm{PG}^{p}_{E}(x)}\ \neg_{\mathrm{PG}} \qquad \frac{x \neq z \quad \Gamma \vdash \varphi : \mathrm{PG}^{p}_{E}(z)}{\Gamma \vdash \exists x.\ \varphi : \mathrm{PG}^{p}_{E}(z)}\ \exists_{\mathrm{PG}} \qquad \frac{x \neq z \quad \Gamma \vdash \varphi : \mathrm{PG}^{p}_{E}(z)}{\Gamma \vdash \forall x.\ \varphi : \mathrm{PG}^{p}_{E}(z)}\ \forall_{\mathrm{PG}}$$

$$\frac{\Gamma \vdash \varphi : \mathrm{PG}^{+}_{E}(x)}{\Gamma \vdash \varphi \wedge \psi : \mathrm{PG}^{+}_{E}(x)}\ \wedge^{\mathrm{L}+}_{\mathrm{PG}} \qquad \frac{\Gamma \vdash \psi : \mathrm{PG}^{+}_{E}(x)}{\Gamma \vdash \varphi \wedge \psi : \mathrm{PG}^{+}_{E}(x)}\ \wedge^{\mathrm{R}+}_{\mathrm{PG}} \qquad \frac{\Gamma \vdash \varphi : \mathrm{PG}^{-}_{E}(x) \quad \Gamma \vdash \psi : \mathrm{PG}^{-}_{E'}(x)}{\Gamma \vdash \varphi \wedge \psi : \mathrm{PG}^{-}_{E \cup E'}(x)}\ \wedge^{-}_{\mathrm{PG}}$$

$$\frac{v \in \overline{x} \quad \forall u \in \mathsf{fv}(\overline{t}).\ \exists E_u \subseteq \Gamma^{-1}(\overline{\mathbb{C}}).\ \Gamma \vdash \varphi : \mathrm{PG}^{+}_{E_u}(u)}{\Gamma \vdash \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi : \mathrm{PG}^{+}_{\bigcup_{u \in \mathsf{fv}(\overline{t})} E_u}(v)}\ \mathsf{agg}_{\mathrm{PG},\overline{x}}$$

$$\frac{v \in \overline{y} \quad \Gamma \vdash \varphi : \mathrm{PG}^{p}_{E}(v)}{\Gamma \vdash \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi : \mathrm{PG}^{p}_{E}(v)}\ \mathsf{agg}_{\mathrm{PG},\overline{y}} \qquad \frac{\mathsf{let}_e \in \mathrm{dom}\,\Gamma \quad \Gamma(\mathsf{let}_{e,i,p}) = E}{\Gamma \vdash e(x_1, \ldots, x_i = x, \ldots, x_k) : \mathrm{PG}^{p}_{E}(x)}\ \mathsf{let}_{\mathrm{PG}}$$

$$\frac{0 \notin I \quad \Gamma \vdash \varphi : \mathrm{PG}^{+}_{E}(x)}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathrm{PG}^{+}_{E}(x)}\ \mathsf{S}^{\mathrm{L}+}_{\mathrm{PG}} \qquad \frac{\Gamma \vdash \psi : \mathrm{PG}^{+}_{E}(x)}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathrm{PG}^{+}_{E}(x)}\ \mathsf{S}^{\mathrm{R}+}_{\mathrm{PG}} \qquad \frac{0 \in I \quad \Gamma \vdash \psi : \mathrm{PG}^{-}_{E}(x)}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathrm{PG}^{-}_{E}(x)}\ \mathsf{S}^{-}_{\mathrm{PG}}$$

$$\frac{0 \notin I \quad \Gamma \vdash \varphi : \mathrm{PG}^{+}_{E}(x)^{+}}{\Gamma \vdash \varphi\, \mathsf{U}_I\, \psi : \mathrm{PG}^{+}_{E}(x)}\ \mathsf{U}^{\mathrm{L}+}_{\mathrm{PG}} \qquad \frac{\Gamma \vdash \varphi : \mathrm{PG}^{+}_{E}(x) \quad \Gamma \vdash \psi : \mathrm{PG}^{+}_{E'}(x)}{\Gamma \vdash \varphi\, \mathsf{U}_I\, \psi : \mathrm{PG}_{E \cup E'}(x)^{+}}\ \mathsf{U}^{\mathrm{LR}+}_{\mathrm{PG}}$$

Past-guardedness

$$\frac{0 \in I \quad \Gamma \vdash \psi : \mathrm{PG}^{-}_{E}(x)}{\Gamma \vdash \varphi\, \mathsf{U}_I\, \psi : \mathrm{PG}^{-}_{E}(x)}\ \mathsf{U}^{-}_{\mathrm{PG}} \qquad \frac{\Gamma \vdash \varphi : \mathrm{PG}^{+}_{E}(x)}{\Gamma \vdash \bullet_I\, \varphi : \mathrm{PG}^{+}_{E}(x)}\ \bullet^{+}_{\mathrm{PG}}$$

$$\frac{\Gamma \vdash \varphi : \tau' \quad \tau \sqsubseteq \tau'}{\Gamma \vdash \varphi : \tau}\ \mathsf{cast} \qquad \frac{}{\Gamma \vdash \top : \mathbb{C}_\alpha}\ \top^{\mathbb{C}} \qquad \frac{}{\Gamma \vdash \bot : \mathbb{S}_\alpha}\ \bot^{\mathbb{S}}$$

$$\frac{e \in \mathbb{C} \vee \mathsf{let}_e \in \mathrm{dom}\,\Gamma \quad \Gamma(e) = \mathbb{C}_s \quad \forall x \in \bigcup_{i=1}^{k} \mathsf{fv}(t_i).\ \exists E \subseteq \Gamma^{-1}(\overline{\mathbb{C}_n}).\ \Gamma(x) = \mathrm{PG}^{+}_{E} \quad \bigcup_{i=1}^{k} \mathsf{fn}(t_i) \subseteq \mathbb{F}_s}{\Gamma \vdash e(t_1, \ldots, t_k) : \mathbb{C}_s}\ \mathbb{E}^{\mathbb{C}_s}$$

$$\frac{e \in \mathbb{C} \vee \mathsf{let}_e \in \mathrm{dom}\,\Gamma \quad \Gamma(e) = \mathbb{C}_n \quad \forall x \in \bigcup_{i=1}^{k} \mathsf{fv}(t_i).\ \exists E \subseteq \Gamma^{-1}(\overline{\mathbb{C}_n}).\ \Gamma(x) = \mathrm{PG}^{+}_{E}}{\Gamma \vdash e(t_1, \ldots, t_k) : \mathbb{C}_n}\ \mathbb{E}^{\mathbb{C}n}$$

$$\frac{\mathsf{let}_e \in \mathrm{dom}\,\Gamma \quad \Gamma(e) = \mathbb{S}_s \quad \forall x \in \bigcup_{i=1}^{k} \mathsf{fv}(t_i).\ \exists E \subseteq \Gamma^{-1}(\overline{\mathbb{C}_n}).\ \Gamma(x) = \mathrm{PG}^{+}_{E} \quad \bigcup_{i=1}^{k} \mathsf{fn}(t_i) \subseteq \mathbb{F}_s}{\Gamma \vdash e(t_1, \ldots, t_k) : \mathbb{S}_s}\ \mathbb{E}^{\mathbb{S}_s}$$

$$\frac{\mathsf{let}_e \in \mathrm{dom}\,\Gamma \quad \Gamma(e) = \mathbb{S}_n \quad \forall x \in \bigcup_{i=1}^{k} \mathsf{fv}(t_i).\ \exists E \subseteq \Gamma^{-1}(\overline{\mathbb{C}_n}).\ \Gamma(x) = \mathrm{PG}^{+}_{E}}{\Gamma \vdash e(t_1, \ldots, t_k) : \mathbb{S}_n}\ \mathbb{E}^{\mathbb{S}n}$$

$$\frac{e \in \mathbb{S} \quad \Gamma(e) = \mathbb{S}}{\Gamma \vdash e(t_1, \ldots, t_k) : \mathbb{S}_0}\ \mathbb{E}^{\mathbb{S}0} \qquad \frac{\Gamma \vdash \varphi : \mathbb{S}_\alpha}{\Gamma \vdash \neg\varphi : \mathbb{C}_\alpha}\ \neg^{\mathbb{C}} \qquad \frac{\Gamma \vdash \varphi : \mathbb{C}_\alpha}{\Gamma \vdash \neg\varphi : \mathbb{S}_\alpha}\ \neg^{\mathbb{S}}$$

$$\frac{\Gamma, x : \mathrm{PG}^{+}_{\emptyset} \vdash \varphi : \mathbb{C}_\alpha}{\Gamma \vdash \exists x.\ \varphi : \mathbb{C}_\alpha}\ \exists^{\mathbb{C}} \qquad \frac{\Gamma, x : \mathrm{PG}^{+}_{E} \vdash \varphi : \mathbb{S}_\alpha \quad \Gamma \vdash \varphi : \mathrm{PG}^{+}_{E}(x)}{\Gamma \vdash \exists x.\ \varphi : \mathbb{S}_\alpha}\ \exists^{\mathbb{S}}$$

$$\frac{\Gamma \vdash \varphi : \mathbb{C}_\alpha \quad \Gamma \vdash \psi : \mathbb{C}_\alpha}{\Gamma \vdash \varphi \wedge \psi : \mathbb{C}_\alpha}\ \wedge^{\mathbb{C}} \qquad \frac{\Gamma \vdash \varphi : \mathbb{S}_\alpha}{\Gamma \vdash \varphi \wedge \psi : \mathbb{S}_\alpha}\ \wedge^{\mathbb{S}\mathrm{L}} \qquad \frac{\Gamma \vdash \psi : \mathbb{S}_\alpha}{\Gamma \vdash \varphi \wedge \psi : \mathbb{S}_\alpha}\ \wedge^{\mathbb{S}\mathrm{R}}$$

$$\frac{0 \in I \quad \Gamma \vdash \psi : \mathbb{C}_\alpha}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathbb{C}_\alpha}\ \mathsf{S}^{\mathbb{C}} \qquad \frac{0 \notin I \quad \Gamma \vdash \varphi : \mathbb{S}_\alpha}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathbb{S}_\alpha}\ \mathsf{S}^{\mathbb{S}\mathrm{L}} \qquad \frac{0 \in I \quad \Gamma \vdash \varphi : \mathbb{S}_\alpha \quad \Gamma \vdash \psi : \mathbb{S}_\alpha}{\Gamma \vdash \varphi\, \mathsf{S}_I\, \psi : \mathbb{S}_\alpha}\ \mathsf{S}^{\mathbb{S}\mathrm{LR}}$$

$$\frac{\Gamma \vdash \psi : \mathbb{S}_\alpha}{\Gamma \vdash \varphi\, \mathsf{U}_I\, \psi : \mathbb{S}_\alpha}\ \mathsf{U}^{\mathbb{S}} \qquad \frac{b \neq \infty \quad \Gamma \vdash \psi : \mathbb{C}_\alpha}{\Gamma \vdash \varphi\, \mathsf{U}_{[0,b]}\, \psi : \mathbb{C}_\alpha}\ \mathsf{U}^{\mathbb{C}\mathrm{R}} \qquad \frac{b \neq \infty \quad \Gamma \vdash \varphi : \mathbb{C}_\alpha \quad \Gamma \vdash \psi : \mathbb{C}_\alpha}{\Gamma \vdash \varphi\, \mathsf{U}_{[a,b]}\, \psi : \mathbb{C}_\alpha}\ \mathsf{U}^{\mathbb{C}\mathrm{LR}}$$

$$\frac{\Gamma \vdash \varphi : \mathbb{C}_\alpha \quad b > 0}{\Gamma \vdash \bigcirc_{[0,b)}\, \varphi : \mathbb{C}_\alpha}\ \bigcirc^{\mathbb{C}} \qquad \frac{\Gamma \vdash \varphi : \mathbb{S}_\alpha}{\Gamma \vdash \bigcirc_I\, \varphi : \mathbb{S}_\alpha}\ \bigcirc^{\mathbb{S}}$$

$$\frac{\Gamma \cup \{\mathsf{let}_{e,i,p} : E \mid \Gamma \vdash \varphi_1 : \mathrm{PG}^{p}_{E}(x_i)\}, \mathsf{let}_e : \bot \vdash \varphi_2 : \tau_2}{\Gamma \vdash \mathsf{let}\, e(x_1, \ldots, x_k) = \varphi_1\ \mathsf{in}\ \varphi_2 : \tau_2}\ \mathsf{let}_{\mathbb{O}}$$

$$\frac{\Gamma \vdash \varphi_1 : \tau_1 \quad \Gamma \cup \{\mathsf{let}_{e,i,p} : E \mid \Gamma \vdash \varphi_1 : \mathrm{PG}^{p}_{E}(x_i)\}, \mathsf{let}_e : \bot, e : \tau_1 \vdash \varphi_2 : \tau_2}{\Gamma \vdash \mathsf{let}\, e(x_1, \ldots, x_k) = \varphi_1\ \mathsf{in}\ \varphi_2 : \tau_2}\ \mathsf{let}$$

Typing of formulae as causable/suppressable

$$\frac{\forall z \in \mathsf{fv}(\varphi) \setminus \overline{y}.\ \Gamma \vdash \varphi : \mathrm{PG}(z)^{+}_{E_z} \quad \Gamma, \forall z.\ z : \mathrm{PG}_{E_z} \vdash \varphi : \mathbb{S}_\alpha \quad |\overline{y}| > 0}{\Gamma \vdash \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi : \mathbb{S}_\alpha}\ \mathsf{agg}^{\mathbb{S}}$$

Fig. 16: Extended typing rules for EMFOTL

Figure 16 (top) shows the full, extended EMFOTL past-guardedness rules. If φ has no let bindings, then none of the past-guardedness rules uses the context Γ. In this case, we write $\vdash \varphi : \mathrm{PG}^p_E(x)$ instead of $\Gamma \vdash \varphi : \mathrm{PG}^p_E(x)$.

We prove:

Lemma 4. *Let φ be an EMFOTL formula without let bindings. For $p \in \{+,-\}$, if $\vdash \varphi : PG^p_E(x)$, then x is past-guarded in $p\varphi$, i.e., for any v, i such that if $v, i \models p\varphi$ and $x \in \operatorname{dom} v$, we have $v(x) \in \mathsf{AD}^*_{\sigma_{..i},E}(\varphi)$.*

Proof. Similar to the proof of Lemma 1 in [25].

A.2 Monitoring MFOTL with function applications and aggregations

In the following, we assume that α-conversion has been applied to ensure that all bound variables are distinct from free variables and that each bound variable is bound by a single quantifier.

Each internal node of a PDT has $k \geq 1$ subtrees, each of which is labeled by a finite or cofinite set $D_k \subseteq \mathbb{D}$ such that the $\{D_i\}_{1\leq i\leq k}$ are a partition of $\mathbb{D}$. As a result, exactly one of the D_i must be infinite. In the following, we call the corresponding ith subtree of a PDT its *infinite subtree* and the other subtrees of this PDT its *finite subtrees*.

We first show that well-formed PDTs map every valuation to a Boolean value.

Definition 7. *A PDT pdt is* well-formed *with respect to a set of variables V iff for any node n labeled by $\mathsf{LClos}\, f\, \bar{t}$ it contains, for any $1 \leq i \leq |t|$ and $z \in \mathsf{fv}(\bar{t}_i) \cap V$, there exists a node n' in pdt such that n' is labeled by $\ell \in \{\mathsf{LEx}\, z, \mathsf{LAll}\, z\}$ and n is contained in a finite subtree of n'.*

Lemma 5. *Let pdt be a PDT and V be the set of all variables occurring in at least one label of pdt. If pdt is well-formed with respect to V, then* specialize *pdt v terminates and returns a Boolean.*

Proof. The only potential source of non-termination in the definition of specialize is the evaluation of $[\![f(\bar{t})]\!]_v$ when specialize reaches a $\mathsf{LClos}\, f\, \bar{t}$ node. Evaluating $[\![f(\bar{t})]\!]_v$ succeeds iff for all $1 \leq i \leq |t|$, $\mathsf{fv}(\bar{t}_i) \subseteq \operatorname{dom} v$. Visiting $\mathsf{LEx}\, z$ or $\mathsf{LVar}\, z$ adds z to $\operatorname{dom} v$, and hence the definition of well-formedness ensures that all $\mathsf{fv}(\bar{t}_i)$ are in $\operatorname{dom} v$. As a consequence, all evaluations of $[\![f(\bar{t})]\!]_v$ succeed.

Recall the definition of monitorability:

Definition 4. *An MFOTL formula φ without let bindings is monitorable iff both of the following conditions hold:*

1. *For any quantified subformula $Qx.\ \psi$ of φ, $Q \in \{\forall, \exists\}$, either $\vdash \psi : PG^+_E(x)$ for some E, or $\vdash \psi : PG^-_E(x)$ for some E, or x does not appear inside any function argument in ψ.*
2. *For any subformula $\overline{x} \leftarrow \omega(\bar{t}; \overline{y})\ \psi$ of φ and any $z \in \mathsf{fv}(\psi) \setminus \overline{y}$, we have $\vdash \psi : PG^+_E(z)$ for some E.*

Next, we present an extension of the monitoring algorithm in [32,33] that can monitor all MFOTL formulae that are monitorable as per Definition 4. Our extended algorithm is applied after unrolling let bindings. For space reasons, we describe a slightly simplified algorithm with the following restrictions:

- We cover only the $\wedge$, $\exists$, $\neg$, S, and U operators as well as aggregations.
- We do not cover the PG rules for S and U. As in Hublet et al. [25], covering these rules requires an extension of the present algorithm that can return *approximate* verdicts (i.e., conservative verdicts for formulae containing future operators based only on the knowledge of past and present events). This extension is implemented in both WhyEnf and EnfGuard.

Algorithm 1 contains helper functions on PDTs that were introduced in the PDT-based monitor WhyMon [33]. To be able to efficiently apply functions on pairs of PDTs (pdt_1, pdt_2)—typically, using the apply2 function in Algorithm 1—it is convenient to assume that the sequences of labels in the nodes of the two PDTs are consistent, i.e., that if a node labeled by ℓ occurs above a node labeled by ℓ' in pdt_1, then no node labeled by ℓ' occurs above a node labeled by ℓ in pdt_2, and vice versa exchanging pdt_1 and pdt_2. This is ensured by computing a fixed order of labels $\overline{\ell}$ that has to be respected in all PDTs that may be combined using apply2 and similar functions. We will use the following definitions:

Definition 8. *A label sequence $\overline{\ell}$ is* well-formed *iff*

1. *All* LVar *nodes in $\overline{\ell}$ appear before all* LEx *and* LAll *nodes;*
2. *All* LEx *and* LAll *nodes in $\overline{\ell}$ appear before all* LCons *nodes;*
3. *$\overline{\ell}$ contains no duplicates;* and
4. *$\overline{\ell}$ never contains two of* LVar z, LEx z, *and* LAll z *for the same variable z.*

Definition 9. *A PDT pdt is* adapted *to a (well-formed) label sequence $\overline{\ell}$ iff $\overline{\ell}$ contains all labels of nodes in pdt and, whenever a node labeled by ℓ_1 occurs above a node labeled by ℓ_2 in pdt, then ℓ_1 appears before ℓ_2 in $\overline{\ell}$.*

The function apply2 (resp. apply3) in Algorithm 1 takes a sequence $\overline{\ell}$ of variables and two (resp. three) PDT arguments adapted to $\overline{\ell}$. Being adapted to the same sequence of labels, such PDTs are pairwise consistent. The return value of apply2 (resp. apply3) is another PDT adapted to $\overline{\ell}$.

Our monitoring algorithm uses the following datatypes.

Definition 10. *Let $\mathbb{I}$ be the set (and type) of non-empty intervals of $\mathbb{N}$ and* Lbl *the type of labels. Define the following algebraic datatypes:*

$$
\begin{aligned}
\mathsf{Buf} &:= [(\mathbb{N}, \mathbb{N}, \mathsf{Pdt\ Bool})] \\
\mathsf{MFormula} &:= \mathsf{MPred}\ \mathbb{E}\ [\mathsf{Term}]\ [\mathsf{Lbl}] \mid \mathsf{MEq\ Term}\ \mathbb{D}\ [\mathsf{Lbl}] \\
&\mid \mathsf{MAnd\ MFormula\ MFormula}\ (\mathsf{Buf}, \mathsf{Buf})\ [\mathsf{Lbl}] \\
&\mid \mathsf{MExists}\ \mathbb{V}\ \mathsf{MFormula} \mid \mathsf{MNeg\ MFormula}\ [\mathsf{Lbl}] \\
&\mid \mathsf{MSince\ MFormula}\ \mathbb{I}\ \mathsf{MFormula}\ (\mathsf{Buf}, \mathsf{Buf})\ (\mathsf{Pdt\ SInfo})\ [\mathsf{Lbl}] \\
&\mid \mathsf{MUntil\ MFormula}\ \mathbb{I}\ \mathsf{MFormula}\ (\mathsf{Buf}, \mathsf{Buf})\ [(\mathbb{N}, \mathbb{N})]\ (\mathsf{Pdt\ UInfo})\ [\mathsf{Lbl}] \\
&\mid \mathsf{MAgg}\ \Omega\ [\mathbb{V}]\ [\mathbb{V}]\ [\mathbb{V}]\ \mathsf{Formula\ MFormula}\ [\mathsf{Lbl}]
\end{aligned}
$$

```
let all_leaves pdt =
    case pdt of
        Leaf a ⇒ {a}
      | Node _ parts ⇒ fold (λs (_, pdt). s ∪ all_leaves pdt) ∅ parts
let simplify′ pdt =
    case pdt, all_leaves pdt of
        Leaf a, _ | _, {a} ⇒ Leaf a, {a}
      | Node t parts ⇒
          let l = map (λ(D, pdt). (D, simplify′ pdt)) parts in
          map (λ(D, (pdt, _)). (D, pdt)) l, fold (λ s (_, (_, s′)). s ∪ s′) ∅ l
let simplify pdt = fst (simplify′ pdt)          // Ensures ∀v. specialize (simplify pdt) v = specialize pdt v
let merge2 f parts₁ parts₂ =                                                  // Helper function for apply2
    case parts₁ of
        [ ] ⇒ parts₂
      | (D₁, pdt₁) : parts₁ ⇒
          [(D₁ ∩ D₂, f pdt₁ pdt₂) | (D₂, pdt₂) ∈ parts₂ ∧ D₁ ∩ D₂ ≠ ∅]
         · merge2 f parts₁ [(D₂ \ D₁, f pdt₁ pdt₂) | (D₂, pdt₂) ∈ parts₂ ∧ D₂ \ D₁ ≠ ∅]
let apply1 f pdt =                              // Ensures ∀v. specialize (apply1 f pdt) v = f (specialize pdt v)
    case pdt of
        Leaf a ⇒ Leaf (f a)
      | Node t parts ⇒ Node t (map (λ(D, pdt). (D, apply1 f pdt)) parts)
let apply2 ℓ̄ f pdt₁ pdt₂ =                             // Ensures ∀v. specialize (apply2 ℓ̄ f pdt₁ pdt₂) v
    case pdt₁, pdt₂, ℓ̄ of                                   // = f (specialize pdt₁ v) (specialize pdt₂ v)
        Leaf a₁, Leaf a₂, _ ⇒ Leaf (f a₁ a₂)
      | Leaf a₁, Node ℓ₂ parts₂, ℓ : ℓ̄ if ℓ = ℓ₂ ⇒
          Node ℓ₂ (map (λ(D, pdt). (D, apply1 (f a₁) pdt)) parts₂)
      | Node ℓ₁ parts₁, Leaf a₂, ℓ : ℓ̄ if ℓ = ℓ₁ ⇒
          Node ℓ₁ (map (λ(D, pdt). (D, apply1 (λa₁. f a₁ a₂) pdt)) parts₁)
      | Node ℓ₁ parts₁, Node ℓ₂ parts₂, ℓ : ℓ̄ if ℓ = ℓ₁ = ℓ₂ ⇒
          Node ℓ₁ (merge2 (apply2 ℓ̄ f) parts₁ parts₂)
      | Node ℓ₁ parts₁, Node ℓ₂ parts₂, ℓ : ℓ̄ if ℓ = ℓ₁ ≠ ℓ₂ ⇒
          Node ℓ₁ (map (λ(D, pdt). (D, apply2 ℓ̄ f pdt pdt₂)) parts₁)
      | Node ℓ₁ parts₁, Node ℓ₂ parts₂, ℓ : ℓ̄ if ℓ = ℓ₂ ≠ ℓ₁ ⇒
          Node ℓ₂ (map (λ(D, pdt). (D, apply2 ℓ̄ f pdt₁ pdt)) parts₂)
      | Node ℓ₁ parts₁, Node ℓ₂ parts₂, ℓ : ℓ̄ if ℓ ≠ ℓ₂ ∧ ℓ ≠ ℓ₁ ⇒
          apply ℓ̄ f pdt₁ pdt₂
      | _, _, [ ] ⇒ fail
let applyn ℓ̄ f pdts =                                                          // Similar for nary f
    apply1 f (fold_right (λpdt pdt′. apply2 ℓ̄ (:) pdt pdt′) pdts (Leaf [ ]))
let apply3 ℓ̄ f pdt₁ pdt₂ pdt₃ =                                                // Similar for trinary f
    applyn ℓ̄ (λ[a₁, a₂, a₃]. f a₁ a₂ a₃) [pdt₁, pdt₂, pdt₃]
let split_prod ℓ̄ pdt = apply1 ℓ̄ (λ(a₁, _). a₁) pdt, apply1 ℓ̄ (λ(_, a₂). a₂) pdt          // Split pairs
```

Algorithm 1: Functions on PDTs

The following overloaded function pdts *can be used to extract all PDTs of a* Buf *or* $\mathsf{MFormula}$ *object as follows:*

$$\mathsf{pdts}\,(\mathit{buf}) := \{\mathit{pdt} \mid (\_\,, \_\,, \mathit{pdt}) \in \mathit{buf}\}$$

$$\mathsf{pdts}\,(\varphi) := \begin{cases} \mathsf{pdts}(\mathit{buf}_1) \cup \mathsf{pdts}(\mathit{buf}_2) \\ \qquad \textit{if } \varphi = \mathsf{MAnd}\,\varphi_1\,\varphi_2\,(\mathit{buf}_1, \mathit{buf}_2)\,\overline{\ell} \\ \mathsf{pdts}(\mathit{buf}_1) \cup \mathsf{pdts}(\mathit{buf}_2) \cup \{\mathit{aux}\} \\ \qquad \textit{if } \varphi = \mathsf{MSince}\,\varphi_1\,I\,\varphi_2\,(\mathit{buf}_1, \mathit{buf}_2)\,\mathit{aux}\,\overline{\ell} \\ \mathsf{pdts}(\mathit{buf}_1) \cup \mathsf{pdts}(\mathit{buf}_2) \cup \{\mathit{aux}\} \\ \qquad \textit{if } \varphi = \mathsf{MUntil}\,\varphi_1\,I\,\varphi_2\,(\mathit{buf}_1, \mathit{buf}_2)\,\mathit{tstps}\,\mathit{aux}\,\overline{\ell} \\ \emptyset \qquad \textit{otherwise.} \end{cases}$$

Furthermore, define as $\mathsf{lb} : \mathsf{MFormula} \to [\mathsf{Lbl}]$ *the function that returns the sequence of labels stored in the last parameter of any* $\mathsf{MFormula}$*. Finally, we naturally relate* $\mathsf{MFormula}$ *objects to MFOTL formulae using an* $\lhd$ *relation in* $\mathcal{P}(\mathsf{MFormula} \times \mathrm{MFOTL})$ *defined inductively as follows:*

$$\frac{\mathsf{reorder}\,\overline{\ell}\,(\mathsf{filter}\,(\lambda x.\,\nexists z.\,x = \mathsf{LCons}\,z)\,(\mathsf{map}\,\mathsf{lbl\_of\_term}\,\overline{t})) \leqslant \overline{\ell}}{\mathsf{MPred}\,e\,\overline{t}\,\overline{\ell} \lhd e(\overline{t})} \qquad \frac{[\mathsf{lbl\_of\_term}] \leqslant \overline{\ell}}{\mathsf{MEq}\,t\,c\,\overline{\ell} \lhd t \approx c}$$

$$\frac{\varphi_1 \lhd \Phi_1 \quad \varphi_2 \lhd \Phi_2 \quad \mathsf{lb}(\varphi_1) = \mathsf{lb}(\varphi_2) = \overline{\ell}}{\mathsf{MAnd}\,\varphi_1\,\varphi_2\,(\mathit{buf}_1, \mathit{buf}_2)\,\overline{\ell} \lhd \Phi_1 \wedge \Phi_2}$$

$$\frac{\varphi_1 \lhd \Phi_1 \quad \mathsf{lb}(\varphi_1) = \mathsf{ex\_label}\,x\,\overline{\ell} \quad \mathsf{LEx}\,x \textit{ is the first } \mathsf{LEx}\,z \textit{ or } \mathsf{LAll}\,z \textit{ label in } \overline{\ell}}{\mathsf{MExists}\,x\,\varphi_1\,\overline{\ell} \lhd \exists x.\ \Phi_1}$$

$$\frac{\varphi_1 \lhd \Phi_1 \quad \mathsf{lb}(\varphi_1) = \mathsf{map}\,\mathsf{neg\_label}\,\overline{\ell}}{\mathsf{MNeg}\,\varphi_1\,\overline{\ell} \lhd \neg\Phi_1} \qquad \frac{\varphi_1 \lhd \Phi_1 \quad \varphi_2 \lhd \Phi_2 \quad \mathsf{lb}(\varphi_1) = \mathsf{lb}(\varphi_2) = \overline{\ell}}{\mathsf{MSince}\,\varphi_1\,I\,\varphi_2\,(\mathit{buf}_1, \mathit{buf}_2)\,\mathit{aux}\,\overline{\ell} \lhd \Phi_1\,\mathsf{S}_I\,\Phi_2}$$

$$\frac{\varphi_1 \lhd \Phi_1 \quad \varphi_2 \lhd \Phi_2 \quad \mathsf{lb}(\varphi_1) = \mathsf{lb}(\varphi_2) = \overline{\ell}}{\mathsf{MUntil}\,\varphi_1\,I\,\varphi_2\,(\mathit{buf}_1, \mathit{buf}_2)\,\mathit{tstps}\,\mathit{aux}\,\overline{\ell} \lhd \Phi_1\,\mathsf{U}_I\,\Phi_2}$$

$$\frac{\varphi_1 \lhd \Phi_1 \quad \mathsf{lb}(\varphi_1) = \mathsf{agg\_labels}\,\overline{\ell}\,\overline{y}\,(\mathsf{lbl}\,\Phi_1)}{\mathsf{MAgg}\,\omega\,\overline{x}\,\overline{t}\,\overline{y}\,\Phi_1\,\varphi_1\,\overline{\ell} \lhd \overline{x} \leftarrow \omega(\overline{t};\overline{y})\ \Phi_1.}$$

Algorithms 2 and 3 show our variant of a (standard) monitoring algorithm for $\varphi\,\mathsf{S}_I\,\psi$ and $\varphi\,\mathsf{U}_I\,\psi$ operators [9,33] using Boolean PDTs. For each S or U subformula, the monitor maintains an auxiliary state (aux for S, $(\mathit{tstps}, \mathit{aux})$ for U). The update functions take as input a sequence of labels $\overline{\ell}$, the interval I, the auxiliary state, and a buffer buf that stores evaluations of φ and ψ at past timepoints. Each such evaluation is reported as a triple $(\mathit{ts}, \mathit{tp}, \mathit{pdt})$ where ts is timestamp, tp a timepoint, and pdt a PDT adapted to $\overline{\ell}$ representing the truth value of the respective formula at timepoint tp with timestamp ts. The update functions return a pair of an updated auxiliary state and a sequence of evaluations $(\mathit{ts}, \mathit{tp}, \mathit{pdt})$ of the overall formula at all timepoints for which an evaluation could be computed using the provided input.

```
let since_init =
    Leaf {beta_alphas_in = [ ]; beta_alphas_out = [ ]}
let since_update1 I ts tp b_α b_β aux =
    let out, in = if b_α then aux.beta_alphas_out, aux.beta_alphas_in else [ ], [ ] in
    let out = if b_β then out · [ts] else out in
    let out′ = filter (λts′. ∀i ∈ I. ts − ts′ < i) out in
    let in′ = filter (λts′. ts − ts′ ∈ I) in · filter (λts′. ts − ts′ ∈ I) out in
    {beta_alphas_in = in′; beta_alphas_out = out′}, ¬(in′ = [ ])
let since_update ℓ̄ I buf aux =
    case buf of
        (ts_α, tp_α, e_α) : es_α, (ts_β, tp_β, e_β) : es_β if (ts_α, tp_α) = (ts_β, tp_β) ⇒
          let aux, b = split_prod ℓ̄ ((simplify ∘ apply3) ℓ̄ (since_update1 I ts_α tp_α) e_α e_β aux) in
          let aux, bs = since_update ℓ̄ I (es_α, es_β) aux in
          aux, (tp_α, ts_α, b) : bs
      | _, _ ⇒ aux, [ ]
```

Algorithm 2: Monitoring S_I

```
let until_init =
    Leaf {n_alpha_in = [ ]; n_alpha_out = [ ]; beta_in = [ ]; beta_out = [ ]}
let until_update1 I ts tp aux =
    let out_¬α = filter (λ(ts′, tp′). ∀i ∈ I. ts′ − ts > i) aux.n_alpha_out in
    let in_¬α = filter (λ(ts′, tp′). tp′ ≥ tp) (aux.n_alpha_out · aux.n_alpha_in) in
    let out_β = filter (λ(ts′, tp′). ∀i ∈ I. ts′ − ts > i) aux.beta_out in
    let in_β = filter (λ(ts′, tp′). ts′ − ts ∈ I) (aux.beta_out · aux.beta_in) in
    let b = ∃(ts′, tp′) ∈ in_β. ∄(ts″, tp″) ∈ in_¬α. tp″ ∈ [tp, tp′] in
    {n_alpha_in = in_α; n_alpha_out = out_α; beta_in = in_β; beta_out = out_β}, b
let load1 I ts tp b_α b_β aux =
    let out_¬α = if ¬b_α then aux.n_alpha_out · [(ts, tp)] else aux.n_alpha_out in
    let out_β = if b_β then aux.beta_out · [(ts, tp)] else aux.beta_out in
    (out_α, out_β)
let load ts buf aux =
    case buf of
        (ts_α, tp_α, e_α) : es_α, (ts_β, tp_β, e_β) : es_β if (ts_α, tp_α) = (ts_β, tp_β) ⇒
          let aux = apply3 ℓ̄ (load1 I ts_α tp_α) e_α e_β aux in
          load ts_α (es_α, es_β) aux
      | _, _ ⇒ ts, buf, aux
let until_update ℓ̄ I buf tstps aux =
    let ts′, buf, aux = load ⊥ buf aux in
    let until_loop_update tstps aux =
        case tstps of
            (ts, tp) : tstps if ts′ ≠ ⊥ ∧ ∀i ∈ I. ts′ − ts > i ⇒
              let aux, b = split_prod ℓ̄ (apply1 ℓ̄ ((simplify ∘ until_update1) I ts tp) aux) in
              let aux, bs = until_loop_update tstps aux in
              aux, (tp_α, ts_α, b) : bs
          | _, _ ⇒ aux, [ ]
    in until_loop_update tstps aux
```

Algorithm 3: Monitoring U_I

```
let buf2_take f buf =
    case buf of
        (ts₁, tp₁, es₁) : buf₁, (ts₂, tp₂, es₂) : buf₂ if (ts₁, tp₁) = (ts₂, tp₂) ⇒
          let es, buf = buf2_take f (buf₁, buf₂) in (ts₁, tp₁, f es₁ es₂) : es, buf
      | _ ⇒ [ ], buf
let tstps2_add tstps es₁ es₂ =
    case es₁, es₂ of
        (ts₁, tp₁, _) : es₁, (ts₂, tp₂, _) : es₂ if (ts₁, tp₁) = (ts₂, tp₂) ⇒
          tstps2_add (tstps [(ts, tp)]) es₁ es₂
      | (ts₁, tp₁, _) : es₁, (ts₂, tp₂, _) : _ if tp₁ < tp₂ | (ts₁, tp₁, _) : es₁, [ ] ⇒
          let tstps = tstps · (if ∀(ts′, tp′) ∈ tstps. tp′ < tp₁ then [(ts₁, tp₁)] else [ ]) in
          tstps2_add tstps es₁ es₂
      | (ts₁, tp₁, _) : _, (ts₂, tp₂, _) : es₂ | [ ], (ts₂, tp₂, _) : es₂ ⇒
          let tstps = tstps · (if ∀(ts′, tp′) ∈ tstps. tp′ < tp₂ then [(ts₂, tp₂)] else [ ]) in
          tstps2_add tstps es₁ es₂
      | [ ], [ ] ⇒ tstps
let apply1_label f g pdt =
    case pdt of
        Leaf a ⇒ Leaf (f a)
      | Node t parts ⇒ Node (g t) (map (λ(D, pdt). (D, apply1_label f g pdt)) parts)
let neg_label ℓ =
    case t of
        LAll z ⇒ LEx z
      | LEx z ⇒ LAll z
      | _ ⇒ ℓ
let ex_label x ℓ̄ = map (λℓ. if ℓ = LEx x then LVar x else ℓ) ℓ̄
let neg_apply1 f pdt = apply1_label f neg_label pdt
let quant_exists x pdt = apply1_label (λx. x) (λz. if z = LVar x then LEx x else z) pdt
let reorder x̄ ȳ =
    case x̄ of
        x : x̄ if x ∈ ȳ ⇒ x : reorder x̄ (ȳ \ x)
      | x : x̄ ⇒ reorder x̄ ȳ
      | [ ] ⇒ ȳ
let agg_labels ℓ̄ ȳ ℓ̄′ = reorder (filter (λℓ. ∃y ∈ ȳ. ℓ = LVar y) ℓ̄) ℓ̄′
```

Algorithm 4: Auxiliary functions

```
let fv φ =
    case φ of
        e(t₁, ..., t_k) ⇒ ⋃_{i=1}^{k} (fv t_i)
      | t ≈ c ⇒ fv t
      | φ₁ ∧ φ₂ ⇒ fv φ₁ ∪ fv φ₂
      | ∃x. φ₁ ⇒ fv φ₁ \ x
      | ¬φ₁ ⇒ fv φ₁
      | φ₁ S_I φ₂ ⇒ fv φ₁ ∪ fv φ₂
      | φ₁ U_I φ₂ ⇒ fv φ₁ ∪ fv φ₂
      | x̄ ← ω(t̄; ȳ) φ ⇒ x̄ ∪ ȳ
let lbl_of_term t =
    case t of
        x if x ∈ 𝕍 ⇒ LVar x
      | c if c ∈ 𝔻 ⇒ LCons c
      | e(ū) ⇒ LClos e ū
let lbl′ φ =
    case φ of
        e(t₁, ..., t_k) ⇒ [ ], {LClos e ū | 1 ≤ i ≤ k, t_i = e(ū)}
      | t ≈ c ⇒ [ ], {LClos e ū | t = e(ū)}
      | ∃x. φ₁ ⇒ let x̄₁, t̄₁ = lbl′ φ₁ in [LEx x] · x̄₁, t̄₁
      | ¬φ₁ ⇒ let x̄₁, t̄₁ = lbl′ φ₁ in (map neg_label x̄₁), t̄₁
      | φ₁ ∧ φ₂ | φ₁ S_I φ₂ | φ₁ U_I φ₂ ⇒
          let x̄₁, t̄₁ = lbl′ φ₁ and x̄₂, t̄₂ = lbl′ φ₂ in x̄₁ · x̄₂, t̄₁ ∪ t̄₂
      | x̄ ← ω(t̄; ȳ) φ₁ ⇒ [ ], ∅
let lbl φ = let x̄, t̄ = lbl′ φ in sorted_list {LVar z | z ∈ fv φ} · x̄ · sorted_list t̄
    // lbl assumes the existence of a total order on labels and a function sorted_list : {a} → [a]
```

Algorithm 5: Free variables, terms, and labels

Let $\mathsf{fv}(\varphi)$ and $\mathsf{bv}(\varphi)$ denote the bound variables of a formula φ, defined as follows:

$$\mathsf{fv}(\varphi) = \begin{cases} \mathsf{fv}(\varphi_1) \cup \mathsf{fv}(\varphi_2) & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \text{ or } \varphi_1 \, \mathsf{S}_I \, \varphi_2 \text{ or } \varphi_1 \, \mathsf{U}_I \, \varphi_2 \\ \mathsf{fv}(\varphi_1) \setminus \{x\} & \text{if } \varphi = \exists x.\ \varphi_1 \\ \mathsf{fv}(\varphi_1) & \text{if } \varphi = \neg \varphi_1 \\ \overline{x} \cup \overline{y} & \text{if } \varphi = \overline{x} \leftarrow \omega(\overline{t}; \overline{y}) \ \varphi_1 \\ \mathsf{fv}(\varphi_1) \setminus \overline{x} \cup \mathsf{fv}(\varphi_2) & \text{if } \varphi = \mathsf{let}\, e(\overline{x}) = \varphi_1 \mathsf{~in~} \varphi_2 \\ \emptyset & \text{if } \varphi = e(\overline{t}) \text{ or } t \approx c \end{cases}$$

$$\mathsf{bv}(\varphi) = \begin{cases} \mathsf{bv}(\varphi_1) \cup \mathsf{bv}(\varphi_2) & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \text{ or } \varphi_1 \, \mathsf{S}_I \, \varphi_2 \text{ or } \varphi_1 \, \mathsf{U}_I \, \varphi_2 \\ \mathsf{bv}(\varphi_1) \cup \{x\} & \text{if } \varphi = \exists x.\ \varphi_1 \\ \mathsf{bv}(\varphi_1) & \text{if } \varphi = \neg \varphi_1 \\ \mathsf{fv}(\varphi_1) \setminus \overline{y} \cup \mathsf{bv}(\varphi_1) & \text{if } \varphi = \overline{x} \leftarrow \omega(\overline{t}; \overline{y}) \ \varphi_1 \\ \mathsf{bv}(\varphi_1) \cup \mathsf{bv}(\varphi_2) & \text{if } \varphi = \mathsf{let}\, e(\overline{x}) = \varphi_1 \mathsf{~in~} \varphi_2 \\ \emptyset & \text{if } \varphi = e(\overline{t}) \text{ or } t \approx c \end{cases}$$

Definition 11. *Given a well-formed label sequence $\overline{\ell}'$, we write $\overline{\ell} \leqslant \overline{\ell}'$ iff $\overline{\ell}$ is a (well-formed) subsequence of $\overline{\ell}'$.*

Our monitoring algorithm is shown in Algorithm 6. We prove:

```
let init ℓ̄ φ =
    case φ of
        e(t₁, ..., t_k) ⇒ MPred e (t₁, ..., t_k) ℓ̄
      | t ≈ c ⇒ MEq t c ℓ̄
      | φ₁ ∧ φ₂ ⇒ MAnd (init ℓ̄ φ₁) (init ℓ̄ φ₂) ([ ], [ ]) ℓ̄
      | ∃x. φ₁ ⇒ MExists x (init (ex_label x ℓ̄) φ₁) ℓ̄
      | ¬φ₁ ⇒ MNeg (init (map neg_label ℓ̄) φ₁) ℓ̄
      | φ₁ S_I φ₂ ⇒ MSince (init ℓ̄ φ₁) I (init ℓ̄ φ₂) ([ ], [ ], [ ]) since_init
      | φ₁ U_I φ₂ ⇒ MUntil (init ℓ̄ φ₁) I (init ℓ̄ φ₂) ([ ], [ ], [ ]) [ ] until_init ℓ̄
      | x̄ ← ω(t̄; ȳ) φ₁ ⇒ MAgg ω x̄ t̄ ȳ φ₁ (init (agg_label ℓ̄ ȳ (lbl Φ₁)) φ₁) ℓ̄
let pdt_of ℓ̄ ū M =
    case ℓ̄ of
        [ ] ⇒ Leaf (M ≠ ∅)
      | LCons c : ℓ̄ ⇒
          pdt_of ℓ̄ ū {(d₁, ..., d_k) ∈ M | ∀1 ≤ i ≤ k. ū_i = LCons c ⇒ d_i = c}
      | (t = LVar _) : ℓ̄ | (t = LClos _ _) : ℓ̄ ⇒
          let M = {d | (d₁, ..., d_k) ∈ M, ∀1 ≤ i ≤ k. ū_i = t ⇒ d_i = d} in
          let g = λd. {(d₁, ..., d_k) ∈ M | ∀1 ≤ i ≤ k. ū_i = t ⇒ d_i = d} in
          Node t (map (λ d. ({d}, pdt_of ℓ̄ ū (g d))) M · [(𝔻 \ M, Leaf ⊥)])
let eval φ (σ = ⟨τ, D⟩_{1≤i≤|σ|}) i =
    case φ of
        MPred e (t₁, ..., t_k) ℓ̄ ⇒
          let M = {(d₁, ..., d_k) | e(d₁, ..., d_k) ∈ D_i} in
          let ℓ̄′ = [lbl_of_term t_i | 1 ≤ i ≤ k] in
          [(τ_i, i, pdt_of (reorder ℓ̄ ℓ̄′) ℓ̄′ M)], φ
      | MEq t c ℓ̄ ⇒
          [(τ_i, i, Node t [({c}, ⊤), (𝔻 \ {c}, ⊥)])], φ
      | MAnd φ₁ φ₂ (buf₁, buf₂) ℓ̄ ⇒
          let es₁, φ₁ = eval φ₁ σ i in
          let es₂, φ₂ = eval φ₂ σ i in
          let es, buf′ = buf2_take ((simplify ∘ apply2) ℓ̄ (λb₁ b₂. b₁ ∧ b₂)) (buf₁ · es₁, buf₂ · es₂) in
          es, MAnd φ₁ φ₂ buf′
      | MExists x φ₁ ℓ̄ ⇒
          let es₁, φ₁ = eval φ₁ σ i in
          map (λ(ts, tp, pdt). (ts, tp, quant_exists x pdt)) es₁, MExists x φ₁
      | MNeg φ₁ ⇒
          let es₁, φ₁ = eval φ₁ σ i in
          map (λ(ts, tp, pdt). (ts, tp, neg_apply1 (λb. ¬b))) es₁, MNeg φ₁
      | MSince φ₁ I φ₂ (buf₁, buf₂) aux ℓ̄ ⇒
          let es₁, φ₁ = eval φ₁ σ i in
          let es₂, φ₂ = eval φ₂ σ i in
          let buf′ = (buf₁ · es₁, buf₂ · es₂) in
          let es, aux′ = since_update ℓ̄ I buf′ aux in
          es, MSince φ₁ φ₂ buf′ aux′
      | MUntil φ₁ I φ₂ buf tstps aux ℓ̄ ⇒
          let es₁, φ₁ = eval φ₁ σ i in
          let es₂, φ₂ = eval φ₂ σ i in
          let buf′ = (buf₁ · es₁, buf₂ · es₂) in
          let tstps′ = tstps2_add tstps es₁ es₂ in
          let es, aux′ = until_update ℓ̄ I buf′ aux in
          es, MUntil φ₁ φ₂ buf′ tstps′ aux′
      | MAgg ω v̄ w̄ ȳ Φ₁ φ₁ ℓ̄ ⇒
          let es₁, φ₁ = eval φ₁ σ i in
          let es = map (aggregate ω v̄ w̄ ȳ (reorder [x | LVar x ∈ ℓ̄] (v̄ · ȳ))) es in
          es, MAgg ω v̄ w̄ ȳ Φ₁ φ₁
```

Algorithm 6: Monitoring algorithm for monitorable MFOTL formulae

Lemma 6. *Let Φ be an MFOTL formula without* let *bindings that is monitorable as per Definition 4. Let $\varphi \lhd \Phi$ such that $\overline{\ell} := \mathsf{lb}(\varphi)$ is well-formed. Let $p \in \{+,-\}$ and assume that $\vdash \Phi : PG^p_E(x)$. Define $+\varphi := \varphi$, $-\varphi :=$ $\mathsf{MNeg}\,\varphi\,(\mathsf{lb}\,\varphi)$. Let $(es, \varphi') = \mathsf{eval}\,\overline{\ell}\,(p\varphi)\,\sigma\,i$ and (ts, tp, pdt) in es. Then:*

(i) Let n be a Leaf *node of pdt with Boolean value $b = \top$ if $p = +$ and $b = \bot$ if $p = -$. There exists a node n' in pdt labeled by* $\mathsf{LVar}\,x$ *such that n is in a finite subtree of n'.*

(ii) Let n be a node labeled by $\mathsf{LVar}\,x$ *in pdt. The infinite subtree of n is reduced to* $\mathsf{Leaf}\,(\neg b)$.

Proof. By induction on the derivation of $\vdash \Phi : \mathrm{PG}^p_E(x)$.

- Rule $\mathbb{E}^+_{\mathrm{PG}}$: In this case, $\Phi = e(t_1, \ldots, t_i = x, \ldots, t_k)$, $p = +$, $E = \{e\}$, $\varphi = \mathsf{MPred}\,e\,(t_1, \ldots, t_k)$. The function eval returns a single triple $(\tau_i, i, pdt = \mathsf{pdt\_of}\,(\mathsf{reorder}\,\overline{\ell}\,\overline{\ell}')\,\overline{\ell}'\,M)$ where $\overline{\ell}'$ is $[\mathsf{lbl\_of\_term}\,\overline{t}_i \mid 1 \le i \le k]$ and M is a set of k-tuples in $\mathbb{D}$. Let $\overline{\ell}_2 = \mathsf{filter}\,(\lambda x.\,\nexists z.\,x = \mathsf{LCons}\,z)\,\overline{\ell}'$ and $\overline{\ell}_3 = \mathsf{filter}\,(\lambda x.\,\nexists z.\,x = \mathsf{LCons}\,z)\,(\mathsf{reorder}\,\overline{\ell}\,\overline{\ell}')$. First, observe that all labels in $\overline{\ell}_2$ are in $\mathsf{lbl}\,\Phi$ as well (see Algorithm 5). Since $\mathsf{reorder}\,\overline{\ell}_2\,\overline{\ell'} \leqslant \overline{\ell}$ by $\varphi \lhd \Phi$, it follows that all labels in $\overline{\ell}_2$ are in $\overline{\ell}$. Now, under this assumption, remark that $\mathsf{reorder}\,\overline{\ell}\,\overline{\ell}'$ (defined in Algorithm 4) returns an interleaving of a subsequence of $\overline{\ell}$ with the LCons labels in $\overline{\ell}'$, and hence $\overline{\ell}_3 \leqslant \overline{\ell}$. The function $\mathsf{pdt\_of}\,(\mathsf{reorder}\,\overline{\ell}\,\overline{\ell}')$ returns a PDT composed of a single chain of nodes whose labels are exactly those in $\overline{\ell}_3$, with all infinite subtrees reduced to $\mathsf{Leaf}\,\bot$. The label $\mathsf{LVar}\,x$ is in $\overline{\ell}_2$ (by definition of $\overline{\ell}'$, lbl_of_term, and $\overline{\ell}_2$), hence also in $\overline{\ell}$ and finally in $\overline{\ell}_3$. Now, there is a single $\mathsf{Leaf}\,\top$ node located at the bottom of the tree within the finite subtree of all LVar nodes, which proves (i). Property (ii) is straightforward by definition of pdt_of.
- Rule $=^+_{\mathrm{PG}}$: In this case, $\Phi = x = c$, $p = +$, $E = \{e\}$, $\varphi = \mathsf{MEq}\,x\,c$. The function eval returns a single triple $(\tau_i, i, pdt = \mathsf{Node}\,(\mathsf{LVar}\,x)\,[(\{c\}, \top), (\mathbb{D} \setminus \{c\}, \bot)])$. The only $\mathsf{Leaf}\,\top$ node is located in the finite subtree of an $\mathsf{LVar}\,x$ node, proving (i). The only $\mathsf{LVar}\,x$ node has its only infinite subtree reduced to $\bot$, proving (ii).
- Rule $\neg_{\mathrm{PG}}$: In this case, $\Phi = \neg\Phi_1$, $\vdash \Phi_1 : \mathrm{PG}^{\neg p}_E(x)$, $\varphi_1 \lhd \Phi_1$, $\varphi = \mathsf{MNeg}\,\varphi_1\,\overline{\ell}$, $\overline{\ell} = \mathsf{lb}(\varphi_1)$. The algorithm first calls eval on φ_1 (l. 37). Our induction hypothesis applied on Φ_1 and $\overline{\ell}'$ shows that any $\mathsf{Leaf}\,(\neg b)$ node in any PDT in es_1 is located in the finite subtree of an $\mathsf{LVar}\,x$ node. Now, every pdt in es is obtained from such a PDT by applying the neg_apply1 function (l. 38) defined in Algorithm 1. This function exchanges LEx and LAll labels in the PDT and $\top$ and $\bot$ leaves. Hence, to the $\mathsf{Leaf}\,b$ node n in pdt corresponds a $\mathsf{Leaf}\,(\neg b)$ node n_1 in a PDT pdt_1 from es_1. We obtain a node n'_1 labeled by $\mathsf{LVar}\,x$ in pdt_1 such that n_1 is in a finite subtree of n'_1. This node n'_1 is mapped by neg_apply1 to a node n' with the same label in pdt such that n is in a finite subtree of n', yielding (i). Similarly, to every node labeled by $\mathsf{LVar}\,x$ in pdt corresponds a node labeled by $\mathsf{LVar}\,x$ in pdt_1 with an infinite subtree reduced to $\mathsf{Leaf}\,b$, which becomes an infinite subtree reduced to $\mathsf{Leaf}\,(\neg b)$ in pdt. This proves (ii).

- Rule $\exists_{\mathrm{PG}}$: In this case, $\varPhi = \exists z.\ \varPhi_1$, $\vdash \varPhi_1 : \mathrm{PG}^p_E(x)$, $x \neq z$, $\varphi_1 \triangleleft \varPhi_1$, $\varphi = \mathsf{MExists}\, z\, \varphi_1$, $\mathsf{lb}(\varphi_1) = \mathsf{ex\_label}\, x\, \overline{\ell}$, and $\mathsf{LEx}\, x$ is the first $\mathsf{LEx}\, z$ or $\mathsf{LAll}\, z$ label in $\overline{\ell}$. The algorithm first calls eval on φ_1 (l. 34) to obtain a pair (es_1, φ_1'). The definition of $\mathsf{ex\_label}$ (see Algorithm 4) guarantees that $\mathsf{lb}(\varphi_1)$ is well-formed since $\mathsf{LEx}\, x$ is the first quantified label in $\overline{\ell}$. Hence, our induction hypothesis applied on $\varPhi_1$ ensures that any $\mathsf{Leaf}\, b$ node in any PDT in es_1 is located in a finite subtree of an $\mathsf{LVar}\, x$ node. Now, every pdt in es is obtained from such a PDT by applying the $\mathsf{quant\_exists}\, z$ function (l. 35) defined in Algorithm 1. This function replaces $\mathsf{LVar}\, z$ nodes by $\mathsf{LEx}\, z$ nodes and has no effect on other nodes. Hence, to the $\mathsf{Leaf}\, b$ node n in pdt corresponds a $\mathsf{Leaf}\, b$ node n_1 in a PDT pdt_1 from es_1. We obtain a node n_1' labeled by $\mathsf{LVar}\, x \neq \mathsf{LVar}\, z$ in pdt_1 such that n_1 is in a finite subtree of n_1'. This node n_1' is mapped by $\mathsf{quant\_exists}\, z$ to a node n' with the same label in pdt such that n is in a finite subtree of n', yielding (i). Similarly, to every node labeled by $\mathsf{LVar}\, x$ in pdt corresponds a node labeled by $\mathsf{LVar}\, x$ in pdt_1 with an infinite subtree reduced to $\mathsf{Leaf}\, (\neg b)$, which is preserved in pdt. This proves (ii).
- Rule $\wedge^{\mathrm{L+}}_{\mathrm{PG}}$: In this case, $\varPhi = \varPhi_1 \wedge \varPhi_2$, $\vdash \varPhi_1 : \mathrm{PG}^+_E(x)$, $\varphi_1 \triangleleft \varPhi_1$, $\varphi = \mathsf{MAnd}\, \varphi_1\, \varphi_2, (buf_1, buf_2)\, \overline{\ell}$, $\mathsf{lb}(\varphi_1) = \mathsf{lb}(\varphi_2) = \overline{\ell}$. The algorithm for MAll first calls eval on φ_1 and φ_2 with label sequence $\overline{\ell}$ (l. 29–30) to obtain two pairs (es_1, φ_1') and (es_2, φ_2'). It then adds the elements of es_1 and es_2 to buf_1 and buf_2 respectively. Hence, we can use our induction hypothesis to show that at any time, each of the triples (ts_1, tp_1, pdt_1) in buf_1 is such that any $\mathsf{Leaf}\, \top$ node n_1 in pdt_1 is in a finite subtree of a node n_1' labeled with $\mathsf{LVar}\, x$. Every triple (ts, tp, pdt) is obtained by applying $(\mathsf{simplify} \circ \mathsf{apply2})\, \overline{\ell}\, (\lambda b_1\, b_2.\ b_1 \wedge b_2)$ on a pair of PDTs from buf_1 and buf_2. Consider first $pdt' = \mathsf{apply2}\, \overline{\ell}\, (\lambda b_1\, b_2.\ b_1 \wedge b_2)\, pdt_1\, pdt_2$ where pdt_1 stems from buf_1, noting that $pdt = \mathsf{simplify}\, pdt'$. By the definition of $\mathsf{apply2}$ (see Algorithm 1), the $\wedge$ function is only applied after having processed all nodes from both pdt_1 and pdt_2. Given the existence of our $\mathsf{Leaf}\, \top$ node n in pdt, we can find, by definition of $\mathsf{simplify}$, another $\mathsf{Leaf}\, \top$ node n' in pdt' such that the whole path from the root to n' is preserved in pdt by $\mathsf{simplify}$. From this n', we can find another $\mathsf{Leaf}\, \top$ node n_1 in pdt_1 that is used by $\mathsf{apply2}$ to produce the Leaf node n' in l. 20 or 22 of Algorithm 1. This leaf must be $\top$, since otherwise the result of applying $\wedge$ could not be $\top$. By the above, there exists a node n_1' in pdt_1 labeled by $\mathsf{LVar}\, x$ such that n_1 is in a subtree of n_1'. This node must be in $\overline{\ell}$ and have been entered by $\mathsf{apply2}$ on its path to the leaf (l. 25 or 27), and hence there exists a node n'' labeled by $\mathsf{LVar}\, x$ above n' in pdt'. The node n can only be in a finite subtree of n'. This is clear if the node n'' is introduced on l. 24 or 28 in Algorithm 1, since by our induction hypothesis the infinite subtree of n_1' is reduced to $\mathsf{Leaf}\, \bot$. If the node n'' is introduced on l. 26 in Algorithm 1, then observe that the partitions are of the form $\varDelta_{i_1 i_2} = D_{1i_1} \cap (D_{2i_2} \setminus \cup_{i=1}^{i_1-1} D_{1i})$, where $D_{11}, \ldots, D_{1k_1}$ are partitions of n_1' and $D_{21}, \ldots, D_{2k_2}$ are partitions of a node in pdt_2. Assuming that only D_{1k_1} and D_{2k_2} are infinite, only the partition $\varDelta_{i_1 i_2}$ is infinite. This partition is associated with the PDT $pdt'' = \mathsf{apply2}\, \overline{\ell}\, (\lambda b_1\, b_2.\ b_1 \wedge b_2)\, pdt_{1k_1}\, pdt_{2k_2}$ but by our induction hypothesis, pdt_{1k_1} is reduced to $\bot$, hence pdt'' is reduced

to $\mathsf{Leaf}\,\bot$ by $\mathsf{simplify}$. As a consequence n can only be in a finite subtree of n'. Now, observe that the definition of $\mathsf{simplify}$ (see Algorithm 1) preserves node n'', as it contains both $\top$ leaves (in n) and $\bot$ leaves (in its infinite subtree). Hence, there exists a node n''' in pdt labeled with $\mathsf{LVar}\,x$ such that n is in a finite subtree of n'''. This establishes (i). For (ii), it suffices to observe that $\mathsf{apply2}$ processes a $\mathsf{LVar}\,x$ node at most once on each path leading to at least one non-$\bot$ leaf and that, when it does, the only infinite subtree it generates contains $\mathsf{apply2}\,\overline{\ell}\,(\lambda b_1\, b_2.\ b_1 \wedge b_2)\, pdt_1'\, pdt_2'$, where pdt_1' is an infinite subtree of an $\mathsf{LVar}\,x$ node in pdt_1'. By our induction hypothesis, this subtree is reduced to $\mathsf{Leaf}\,\bot$, which yields (ii) by applying the definition of $\mathsf{apply2}$.

- Rule $\wedge^{\mathrm{R}+}_{\mathrm{PG}}$: Similar to the previous case, inverting the roles of pdt_1 and pdt_2.
- Rule $\wedge^{-}_{\mathrm{PG}}$: In this case, $\Phi = \Phi_1 \wedge \Phi_2$, $\vdash \Phi_1 : \mathrm{PG}^{-}_{E}(x)$, $\vdash \Phi_2 : \mathrm{PG}^{-}_{E}(x)$, $\varphi_1 \triangleleft \Phi_1$, $\varphi_2 \triangleleft \Phi_2$, $\varphi = \mathsf{MAnd}\,\varphi_1\,\varphi_2\,(buf_1, buf_2)\,\overline{\ell}$, $\mathsf{lb}(\varphi_1) = \mathsf{lb}(\varphi_2) = \overline{\ell}$. As previously, the algorithm for MAnd first calls eval on φ_1 and φ_2 with label sequence $\overline{\ell}$ (l. 29–30) to obtain two pairs (es_1, φ_1') and (es_2, φ_2'). It then adds the elements of es_1 and es_2 to buf_1 and buf_2 respectively. Hence, we can use our induction hypothesis to show that at any time, each of the triples (ts_i, tp_i, pdt_i) in buf_i, $i \in \{1,2\}$ is such that any $\mathsf{Leaf}\,\top$ node n_i in pdt_i is in a finite subtree of a node n_i' labeled with $\mathsf{LVar}\,x$. Every triple (ts, tp, pdt) is obtained by applying $(\mathsf{simplify} \circ \mathsf{apply2})\,\overline{\ell}\,(\lambda b_1\, b_2.\ b_1 \wedge b_2)$ on a pair of PDTs from buf_1 and buf_2. Consider first $pdt' = \mathsf{apply2}\,\overline{\ell}\,(\lambda b_1\, b_2.\ b_1 \wedge b_2)\, pdt_1\, pdt_2$ where pdt_1 stems from buf_1, noting that $pdt = \mathsf{simplify}\, pdt'$. By the definition of $\mathsf{apply2}$ (see Algorithm 1), the $\wedge$ function is only applied after having processed all nodes from both pdt_1 and pdt_2. As in the previous case, we can find another $\mathsf{Leaf}\,\bot$ node n' in pdt' such that the whole path from the root to n' is preserved in pdt by $\mathsf{simplify}$. From this n', we can find a $\mathsf{Leaf}\,\bot$ node n_1 in either pdt_1 or pdt_2 that is used by $\mathsf{apply2}$ to produce the Leaf node n' in l. 20, 22, or 24 of Algorithm 1. By the above, there exists a node n_1' in pdt_i, $i \in \{1,2\}$ labeled by $\mathsf{LVar}\,x$ such that n_1 is in a subtree of n_1'. The rest of the proof of (i) is as in the previous case. For (ii), it suffices to observe that $\mathsf{apply2}$ processes a $\mathsf{LVar}\,x$ node at most once on each path leading to at least one non-$\top$ leaf and that, when it does, the only infinite subtree it generates contains $\mathsf{apply2}\,\overline{\ell}\,(\lambda b_1\, b_2.\ b_1 \wedge b_2)\, pdt_1'\, pdt_2'$, where pdt_1' is an infinite subtree of an $\mathsf{LVar}\,x$ node in pdt_1' and pdt_2' is an infinite subtree of an $\mathsf{LVar}\,x$ node in pdt_2'. By our induction hypothesis, these subtrees are reduced to $\mathsf{Leaf}\,\top$, which yields (ii) by applying the definition of $\mathsf{apply2}$.
- Rule $\mathrm{PG}^{+}_{\mathsf{agg},x}$: In this case, $\Phi{=}\overline{v} \leftarrow \omega(\overline{w};\overline{y})\ \Phi_1$, $x \in \overline{v}$, $\varphi{=}\mathsf{MAgg}\,\omega\,\overline{v}\,\overline{w}\,\overline{y}\,\overline{z}\,\Phi_1\,\varphi_1$. Now, observe that the aggregation algorithm (Algorithm 9) only introduces $\top$ leaves (l. 29) after recursing on one finite subtree of each $\mathsf{LVar}\,v$ node (l. 25). This immediately shows (i). Furthermore, all the infinite subtrees of $\mathsf{LVar}\,v$ nodes, $v \in \overline{v}$, are reduced to $\bot$ (l. 26), showing (ii).
- Rule $\mathrm{PG}^{+}_{\mathsf{agg},\overline{y}}$: In this case, $\Phi = \overline{v} \leftarrow \omega(\overline{w};\overline{y})\ \Phi_1$, $x \in \overline{y}$, $\vdash \Phi_1 : \mathrm{PG}^{p}_{E}(x)$, $\varphi = \mathsf{MAgg}\,\omega\,\overline{v}\,\overline{w}\,\overline{y}\,\overline{z}\,\Phi_1\,\varphi_1$, $\varphi_1 \triangleleft \Phi_1$, $\mathsf{lb}(\varphi_1) = \mathsf{agg\_labels}\,\overline{\ell}\,\overline{y}\,(\mathsf{lbl}\,\Phi_1)$. The algorithm first calls eval on φ_1 (l. 53). The definition of $\mathsf{reorder}$ (see Algorithm 4) that is used in $\mathsf{agg\_labels}$ (see Algorithm 4), ensures that $\mathsf{lb}(\varphi_1)$ is well-formed since $\mathsf{lbl}\,\Phi_1$ is well-formed (by definition of lbl) and the first argument of

$\mathsf{reorder}$ only contains LVar labels. Hence, by induction hypothesis, for any (ts, tp, pdt_1) in the sequence es_1 returned by eval, any $\mathsf{Leaf}\, b$ node n in pdt_1 is located in a finite subtree of a node n' labeled with $\mathsf{LVar}\, x$. Any pdt in es is obtained (l. 54) by applying $\mathsf{aggregate}\, \omega\, \overline{v}\, \overline{w}\, \overline{y}\, (\mathsf{reorder}\, [x \mid \mathsf{LVar}\, x \in \overline{\ell}]\, (\overline{v} \cdot \overline{y}))$ to such a pdt_1. The subfunction gather (l. 7–18 in Algorithm 9) preserves all $\mathsf{LVar}\, y$ nodes, $y \in \overline{y}$, gathering a non-empty list of tuples from $\top$ leaves only. Function agg (l. 19 in Algorithm 9) maps empty lists to None and non-empty lists to some value since $|\overline{y}| > 0$. Finally, function insert inserts non-$\bot$ leaves only in subtrees that do not contain only None leaves. Hence, any $\mathsf{Leaf}\, \top$ node n in pdt can be mapped to at least one $\mathsf{Leaf}\, \top$ node n_1 in pdt_1 such that both n and n_1 are in the finite subtree of an $\mathsf{LVar}\, x$ node. This proves (i). Similarly, the $\mathsf{Leaf}\, \bot$ infinite subtrees of pdt_1 are unaffected by $\mathsf{aggregate}$, yielding (ii).

Lemma 7. *Let Φ be an MFOTL formula without* let *bindings that is monitorable as per Definition 4. Let $\varphi \lhd \Phi$ such that $\mathsf{lb}(\varphi)$ is well-formed. Let $p \in \{+, -\}$ and assume that $\vdash \Phi : PG^{\varphi}_{E}(x)$. Let $(es, \varphi') = \mathsf{eval}\, \overline{\ell}\, (p\varphi)\, \sigma\, i$ and (ts, tp, pdt) in es. Let n be a node in pdt labeled by $\ell = \mathsf{LClos}\, e\, \overline{t}$, $1 \leq i \leq |\overline{t}|$. Finally, assume that pdt is adapted to $\overline{\ell}$. Then there exists a node n' in pdt labeled by $\mathsf{LVar}\, x$ such that n is in a finite subtree of n'.*

Proof. By systematic inspection of Algorithm 6, observe that any such pdt is obtained by applying $\mathsf{simplify}$ to another PDT pdt'. Hence, n has at least one subtree containing a $\mathsf{Leaf}\, \top$ node n'' and one subtree containing a $\mathsf{Leaf}\, \bot$ node (otherwise, $\mathsf{simplify}$ would have removed n, see Algorithm 1, l. 3). By Lemma 6, there exists a node n' labeled by $\mathsf{LVar}\, x$ such that n'' is in a finite subtree of n'. Since n'' is a child of both n and n', then either n is a child of n' or vice versa. But since $\overline{\ell}$ is well-formed and pdt is adapted to $\overline{\ell}$, the label $\mathsf{LVar}\, z$ cannot come after the label $\mathsf{LClos}\, e\, \overline{t}$, and hence that n must be a child of n' through one of its finite subtrees.

Theorem 2. *Let Φ be an MFOTL formula without* let *bindings that is monitorable as per Definition 4. Let $\varphi \lhd \Phi$ such that $\overline{\ell} := \mathsf{lb}(\varphi)$ is well-formed, and V be the set of bound variables of Φ. Assume that for any subformula $\psi \lhd \Psi$ of φ, for all $pdt \in \mathsf{pdts}(\psi)$, pdt is adapted to $\mathsf{lb}(\psi)$ and well-formed with respect to $\mathsf{bv}(\Psi)$. Then the function $\mathsf{eval}\, \varphi\, \sigma\, i$ returns a pair (es, φ') such that for all $pdt \in \mathsf{pdts}(es) \cup \mathsf{pdts}(\varphi')$, pdt is adapted to $\mathsf{lb}(\varphi)$ and well-formed with respect to V.*

Proof. By structural induction on Φ. Denote $\overline{\ell} := \mathsf{lb}(\varphi)$.

– If $\Phi = e(\overline{t})$, then $\varphi = \mathsf{MPred}\, e\, \overline{t}$ and $V = \emptyset$. The function eval returns a single triple $(\tau_i, i, pdt = \mathsf{pdt\_of}\, (\mathsf{reorder}\, \overline{\ell}\, \overline{\ell}')\, \overline{\ell}'\, M)$ where $\overline{\ell}'$ is $[\mathsf{lbl\_of\_term}\, \overline{t}_i \mid 1 \leq i \leq k]$ and M is a set of k-tuples in $\mathbb{D}$. Let $\overline{\ell}_2 = \mathsf{filter}\, (\lambda x.\, \nexists z.\, x = \mathsf{LCons}\, z)\, \overline{\ell}'$ and $\overline{\ell}_3 = \mathsf{filter}\, (\lambda x.\, \nexists z.\, x = \mathsf{LCons}\, z)\, (\mathsf{reorder}\, \overline{\ell}\, \overline{\ell}')$. First, observe that all labels in $\overline{\ell}_2$ are in $\mathsf{lbl}\, \Phi$ as well (see Algorithm 5). Since $\varphi \lhd \Phi$, it follows that all labels in $\overline{\ell}_2$ are in $\overline{\ell}$. Now, under this assumption, remark that $\mathsf{reorder}\, \overline{\ell}\, \overline{\ell}'$ (defined in Algorithm 4) returns an interleaving of a subsequence of $\overline{\ell}$ with the LCons labels in $\overline{\ell}'$, and hence $\overline{\ell}_3 \leqslant \overline{\ell}$. The function $\mathsf{pdt\_of}\, (\mathsf{reorder}\, \overline{\ell}\, \overline{\ell}')$

clearly returns a PDT adapted to $\overline{\ell}_3$. Since $\overline{\ell}_3 \leqslant \overline{\ell}$, *pdt* is also adapted to $\overline{\ell}$. Since $V = \emptyset$, it is also trivially well-formed with respect to V.

- If $\Phi = t \approx c$, then $\varphi = \mathsf{MEq}\, t\, c$ and $V = \emptyset$. The function eval returns a single triple $(\tau_i, i, pdt = \mathsf{Node}\, t\, [(\{c\}, \top), (\mathbb{D} \setminus \{c\}, \bot)])$. Remark that in this case, $\mathsf{lbl}\, \Phi = [\mathsf{lbl\_of\_term}\, t]$ (see Algorithm 5). Since $\varphi \lhd \Phi$, then $\mathsf{lbl\_of\_term}\, t$ is contained in $\overline{\ell}$ and *pdt* is adapted to $\overline{\ell}$. Since $V = \emptyset$, it is also trivially well-formed with respect to V.
- If $\Phi = \Phi_1 \wedge \Phi_2$, then $\varphi = \mathsf{MAnd}\, \varphi_1\, \varphi_2$, $\varphi_1 \lhd \Phi_1$, $\varphi_2 \lhd \Phi_2$, $\mathsf{lb}(\varphi_1) = \mathsf{lb}(\varphi_2) = \overline{\ell}$ By our induction hypothesis and assumption on $\mathsf{pdts}(\varphi)$, we obtain that the triples in $buf_1 \cdot es_1$ and $buf_2 \cdot es_2$ l. 29–30 contain PDTs that are adapted to $\overline{\ell}$ and well-formed with respect to $\mathsf{bv}(\Phi_1)$ and $\mathsf{bv}(\Phi_2)$, respectively. Each PDT returned by eval on l. 32 is of the form $pdt = \mathsf{apply2}\, \overline{\ell}\, (\lambda b_1\, b_2.\, b_1 \wedge b_2)\, pdt_1\, pdt_2$ where pdt_1 stems from es_1 and pdt_2 from es_2. Since both pdt_1 and pdt_2 are adapted to $\overline{\ell}$, by definition of apply2 (see Algorithm 1), the label sequence on any path in *pdt* is an interleaving of a label sequence on a path in pdt_1 and a label sequence on a path in pdt_2. As a consequence, since pdt_1 and pdt_2 are both adapted to $\overline{\ell}$, then *pdt* is also adapted to $\overline{\ell}$. Now, let $z \in V$ and consider a node labeled n with some label ℓ containing z in *pdt*. Without loss of generality, assume $z \in \mathsf{bv}(\Phi_1)$. Then, since $\mathsf{bv}(\Phi_1) \cap \mathsf{bv}(\Phi_2)$, ℓ labels a node n_1 in pdt_1. Since pdt_1 is well-formed with respect to $\mathsf{bv}(\Phi_1)$, there exists a node n_1' higher up in pdt_1 that is labeled with $\ell' \in \{\mathsf{LEx}\, z, \mathsf{LAll}\, z\}$ and such that n_1 is in a finite subtree of n_1'. The label ℓ' is also in $\overline{\ell}$, since pdt_1 is adapted to $\overline{\ell}$. Moreover, as $\overline{\ell}$ is well-formed, ℓ' appears before ℓ in $\overline{\ell}$. In this case, the definition of apply2 ensures that a node n' labeled by ℓ' and with the same partitions as m' has been inserted into *pdt* above the node n, such that n is in a finite subtree of n'. We conclude that *pdt* is well-formed with respect to V.
- If $\Phi = \exists x.\, \Phi_1$, then $\varphi = \mathsf{MExists}\, x\, \varphi_1$, $\varphi_1 \lhd \Phi_1$, $\mathsf{lb}(\varphi_1) = \mathsf{ex\_label}\, x\, \overline{\ell}$, $\mathsf{LEx}\, x$ is the first quantified label in $\overline{\ell}$, and $V = \mathsf{bv}(\Phi_1) \cup \{x\}$ where by assumption $x \notin \mathsf{bv}(\Phi_1)$. By our induction hypothesis, we obtain that the triples in es_1 l. 34 contain PDTs that are adapted to $\mathsf{ex\_label}\, \ell\, \overline{\ell}$ and well-formed with respect to $\mathsf{bv}(\Phi_1)$. Now, observe that the definition of ex_label ensures that, just as $\mathsf{LEx}\, x$ was the first LEx or LAll label in $\mathsf{lbl}\, \Phi$ (and hence the first LEx or LAll label in $\overline{\ell}$ occurring in $\mathsf{lbl}\, \Phi$), $\mathsf{LVar}\, x$ is now the last LVar label in $\mathsf{ex\_label}\, x\, \overline{\ell}$ occurring in $\mathsf{lbl}\, \Phi_1$. Each returned PDT is of the form $pdt = \mathsf{quant\_exists}\, x\, pdt_1$ where pdt_1 stems from es_1. Since pdt_1 is adapted to $\mathsf{ex\_label}\, x\, \overline{\ell}$ and $\mathsf{quant\_exists}\, x$ replaces any instance of a $\mathsf{LVar}\, x$ label by $\mathsf{LEx}\, x$ (see Algorithm 4), we see that *pdt* is adapted to $\overline{\ell}$.
  Now, let $z \in V = \mathsf{bv}(\Phi_1) \cup \{x\}$ and consider a node n in *pdt* labeled with some label ℓ containing z. Let n_1 be the node in pdt_1 that is mapped to n by $\mathsf{quant\_exists}\, x$. If $z \in \mathsf{bv}(\Phi_1)$, then by assumption $z \neq x$ and, since pdt_1 is well-formed with respect to $\mathsf{bv}(\Phi_1)$, there exists a node n_1' in pdt_1 labeled with $\ell' \in \{\mathsf{LEx}\, z, \mathsf{LAll}\, z\}$ such that n_1 is in a finite subtree of n_1'. This node is mapped by $\mathsf{quant\_exists}\, x$ to a node n' also labeled by ℓ' in *pdt*. Since the two trees are isomorphic, n is in a finite subtree of n'. If $z = x$, we use the definition of monitorability (Definition 4) to obtain $\vdash \Phi_1 : \mathrm{PG}_E^p(x)$ for some E and $p \in \{+, -\}$. The case when x does not appear in any function

application in Φ_1 can be ruled out since only function applications give rise to LClos labels. Using Lemma 7, we can now obtain a node n_1' in pdt_1 that is labeled with $\mathsf{LVar}\,x$ and such that n_1 is in a finite subtree of n_1'. By the same isomorphism as above and the definition of $\mathsf{quant\_exists}\,x$, this proves the existence of a node n' in pdt labeled by $\mathsf{LEx}\,x$ such that n is in a finite subtree of n'.

- If $\Phi = \neg\Phi_1$, then $\varphi = \mathsf{MNeg}\,\varphi_1$, $\varphi_1 \triangleleft \Phi_1$, $\mathsf{lb}(\varphi_1) = \mathsf{map\,neg\_label}\,\overline{\ell}$ $V = \mathsf{bv}(\Phi_1)$. By our induction hypothesis, we obtain that the triples in es_1 l. 37 contain PDTs that are adapted to $\overline{\ell}' := \mathsf{map\,neg\_label}\,\overline{\ell}$ and well-formed with respect to $\mathsf{bv}(\Phi) = V$. Each PDT returned by eval on l. 38 is of the form $pdt = \mathsf{neg\_apply}\,(\lambda b.\,b)\,pdt_1$ where pdt_1 stems from es_1. That is, pdt is obtained from pdt_1 by exchanging LEx and LAll labels and $\top$ and $\bot$ leaves. Since pdt_1 is adapted to $\overline{\ell}'$, pdt is thus adapted to $\mathsf{map\,neg\_label}\,\overline{ell}' = \overline{\ell}$. Moreover, since pdt_1 is adapted well-formed with respect to V, then pdt, that has the same LClos and LVar labels and the same quantified labels modulo the exchange of LAll and LEx, is also well-formed with respect to V.
- The cases of S and U are similar to the case of $\wedge$ above.
- If $\Phi = \overline{v} \leftarrow \omega(\overline{w};\overline{y})\ \Phi_1$, then $\varphi = \mathsf{MAgg}\,\omega\,\overline{v}\,\overline{w}\,\overline{y}\,\Phi_1, \varphi_1$, $\varphi_1 \triangleleft \Phi_1$, $V = \emptyset$, $\mathsf{lb}(\varphi_1) = \mathsf{agg\_labels}\,\overline{\ell}\,\overline{y}\,(\mathsf{lbl}\,\Phi_1)$. The definition of $\mathsf{reorder}$ (see Algorithm 4) that is used in $\mathsf{agg\_labels}$ (see Algorithm 4), ensures that $\mathsf{lb}(\varphi_1)$ is well-formed. By our induction hypothesis, we obtain that the triples in es_1 l. 53 contain PDTs that are adapted to $\overline{\ell}'$ and well-formed with respect to $\mathsf{bv}(\Phi_1)$. Each returned PDT is of the form $pdt = \mathsf{aggregate}\,\omega\,\overline{v}, \overline{w}\,\overline{y}\,\overline{z}\,pdt_1$ where $\overline{z} = \mathsf{reorder}\,[x \mid \mathsf{LVar}\,x \in \overline{\ell}]\,(\overline{v}\cdot\overline{y})$ and pdt_1 stems from es_1. Moreover, using the monitorability of Φ as per Definition 4 and Lemma 6, we know that for any $\mathsf{Leaf}\,\top$ node n in pdt, $z \in \mathsf{fv}(\Phi_1) \setminus \overline{y}$, there exists a node n' in pdt such that n' is labeled by $\mathsf{LVar}\,z$ and n is contained in a finite subtree of n'. Hence, the gather in Algorithm 9, when it reaches l. 10, only finds $\ell = \top$ when sv already contains a finite set of potential values for each $z \in \mathsf{fv}(\Phi_1)$. As a consequence, the function $\mathsf{tabulate}$ terminates (i.e., the lists on l. 4 and 5 can always be computed in finite time), producing a finite set M. The function insert inserts $\mathsf{Node}\,(\mathsf{LVar}\,z)$ variables in the order prescribed by $\overline{z}$, hence pdt is adapted to $\mathsf{map\,LVar}\,\overline{z}$. By definition of lbl (see Algorithm 5), $\mathsf{lbl}\,\Phi = \mathsf{sorted\_list}\,\{\mathsf{LVar}\,z \mid z \in \overline{v} \cup \overline{y}\}$. Since by assumption $\varphi \triangleleft \Phi$, then $\overline{\ell}$ contains all labels in $\mathsf{map\,LVar}\,(\overline{v}\cdot\overline{y})$ (possibly reordered). Hence, $\mathsf{map\,LVar}\,\overline{z} = \mathsf{map\,LVar}\,(\mathsf{reorder}\,[x \mid \mathsf{LVar}\,x \in \overline{\ell}]\,(\overline{v}\cdot\overline{y})) = \mathsf{reorder}\,\overline{\ell}\,(\mathsf{map\,LVar}\,(\overline{v}\cdot\overline{y})) \leqslant \overline{\ell}$, and therefore pdt is adapted to $\overline{\ell}$.

Theorem 3. *Let Φ be a closed MFOTL formula without* let *bindings that is monitorable as per Definition 4. Then the sequence defined by*

$$\begin{aligned} \varphi_{-1} &= \mathsf{init}\,\Phi \\ (es_i, \varphi_i) &= \mathsf{eval}\,\varphi_{i-1}\,\sigma\,i \qquad\qquad i \geq 0 \end{aligned}$$

is such that for any $i \geq 0$, $pdt \in \mathsf{pdts}(es_i) \cup \mathsf{pdts}(\varphi_i')$ and for any valuation v, $\mathsf{specialize}\,pdt\,v$ *terminates and returns a Boolean.*

Proof. By induction on i. First, observe that the definition of init (see Algorithm 6) ensures $\mathsf{init}\,\Phi \lhd \Phi$. Using Theorem 2 with Φ and $\varphi := \mathsf{init}\,\Phi$ and observing that $\mathsf{pdts}(\mathsf{init}\,\Phi)$ only contains PDTs reduced to leaves, we obtain that $\mathsf{eval}\,(\mathsf{init}\,\Phi)\,\sigma\,i$ returns a pair (es_0, φ_0) such that for all $pdt \in \mathsf{pdts}(es_0)$, pdt is well-formed with respect to $\mathsf{bv}(\varphi)$. By systematic inspection of Algorithm 6, we see that $\mathsf{bv}(\varphi)$ is also the set of all variables that can appear in any label of pdt. Hence, by Lemma 5, $\mathsf{specialize}\,pdt\,v$ returns a Boolean. The step case is similar.

Consider the following variant of the $\mathsf{specialize}$ function where conjunctions and disjunctions are computed over both finite and infinite partitions (l. 5–6). This function is not executable; however, it is mathematically well-defined on all PDTs and its output is the same as $\mathsf{specialize}$ on all PDTs that are well-formed with respect to the set V of labels appearing in them.

1 **let** $\mathsf{specialize}'\ pdt\ v =$
2 **case** pdt **of**
3 $\mid \mathsf{Leaf}\,\ell \Rightarrow \ell$
4 $\mid \mathsf{Node}\,(\mathsf{LVar}\,x)\ parts \Rightarrow \mathsf{let}\,(\_, pdt') = \mathsf{find}\,parts\,(v\,x)\ \mathsf{in\ specialize}'\ pdt'\ v$
5 $\mid \mathsf{Node}\,(\mathsf{LEx}\,x)\ parts \Rightarrow \bigvee_{(D,pdt')\in parts} \bigvee_{d\in D} \mathsf{specialize}'\ pdt'\ v[x \mapsto d]$
6 $\mid \mathsf{Node}\,(\mathsf{LAll}\,x)\ parts \Rightarrow \bigwedge_{(D,pdt')\in parts} \bigwedge_{d\in D} \mathsf{specialize}'\ pdt'\ v[x \mapsto d]$
7 $\mid \mathsf{LClos}\ f\ \bar{t} \Rightarrow \mathsf{specialize}'\ (\mathsf{find}\,parts\,[\![f(\bar{t})]\!]_v)\ v$

Algorithm 7: $\mathsf{specialize}'$ function

We have:

Lemma 8. *Let pdt and V be the set of labels appearing in pdt. If pdt is well-formed with respect to V and adapted to a well-formed label sequence $\bar{\ell}$, then for any valuation v,* $\mathsf{specialize}'\,pdt\,v = \mathsf{specialize}\,pdt\,v$.

Proof. Since $\bar{\ell}$ is well-formed, there cannot be any $\mathsf{LVar}\,x$ nodes below a $\mathsf{LEx}\,x$ or $\mathsf{LAll}\,x$ node. Hence, the variables set in the LEx or LAll cases are only relevant when an LClos node is reached. Such a node is never reached in an infinite subtree of an LEx or LAll node since pdt is well-formed with respect to V. Hence, the execution of $\mathsf{specialize}$ and $\mathsf{specialize}'$ on pdt are the same, since the two only differ by the setting of the variables in infinite subtrees of LEx and LAnd nodes.

For aggregations, we prove:

Lemma 9. *Let $\varphi = \overline{x} \leftarrow \omega(\bar{t};\overline{y})\ \varphi_1$ and $\overline{z} = \mathsf{fv}(\varphi_1) \setminus \overline{y}$. Let v be a valuation and pdt_1 a PDT such that* $\mathsf{specialize}'\,pdt_1\,v = \textsc{Sat}_{\varphi_1}(v,i,\sigma)$ *and pdt_1 is adapted to a well-formed label sequence* $\mathsf{map\,lbl\_of\_term}\,\overline{y} \cdot \bar{\ell}$ *for some $\bar{\ell}$. Let* $pdt = \mathsf{aggregate}\,\omega\,\overline{x}\,\bar{t}\,\overline{y}\,\overline{z}\,pdt_1$. *Then* $\mathsf{specialize}'\,pdt\,v = \textsc{Sat}_{\varphi}(v,i,\sigma)$.

Proof. Let $\overline{z}{=}\mathsf{fv}(\varphi_1)\setminus\overline{y}$. Let $pdt_2{=}\mathsf{gather}\,[\,]\,\bar{t}\,\overline{y}\,pdt_1$, $pdt_3{=}\mathsf{apply1}\,[\,]\,(\mathsf{agg}\,\overline{y}\,\omega)\,pdt_2$. Then $pdt = \mathsf{insert}\,\emptyset\,\overline{x}\,\overline{z}\,pdt_3$. The function gather l. 7–18 in Algorithm 9 ensures

$$\mathsf{specialize}'\,pdt_2\,v = [[\![\bar{t}]\!]_{v'} \mid \mathrm{dom}\ v' = \mathsf{fv}(\varphi_1) \wedge v|_{\overline{y}} = v'|_{\overline{y}} \wedge \mathsf{specialize}\,pdt_1\,v'].$$

Functions $\mathsf{apply1}$ and $\mathsf{specialize}'$ commute, and hence

$$\begin{aligned}\mathsf{specialize}'\, pdt_3\, v &= \mathsf{agg}\,\overline{y}\,\omega\,[[\![\overline{t}]\!]_{v'} \mid \text{dom } v' = \mathsf{fv}(\varphi_1) \wedge v|_{\overline{y}} = v'|_{\overline{y}} \wedge \mathsf{specialize}\, pdt_1\, v'] \\ &= \mathsf{agg}\,\overline{y}\,\omega\,[[\![\overline{t}]\!]_{v[\overline{z}\mapsto\overline{d}]} \mid v[\overline{z}\mapsto\overline{d}], i \vDash_\sigma \varphi_1, \overline{d} \in \mathbb{D}^{|\overline{z}|}] \\ &= \mathsf{let}\, M = [[\![\overline{t}]\!]_{v[\overline{z}\mapsto\overline{d}]} \mid v[\overline{z}\mapsto\overline{d}], i \vDash_\sigma \varphi_1, \overline{d} \in \mathbb{D}^{|\overline{z}|}]\ \mathsf{in} \\ &\quad\ \mathbf{if}\ M = [\,] \wedge |\overline{y}| = 0\ \mathbf{then}\ \mathsf{None}\ \mathbf{else}\ \omega\, M\end{aligned}$$

Finally, the function insert l. 7–18 in Algorithm 9 is such that

$$\mathsf{specialize}'\,(\mathsf{insert}\,\emptyset\,\overline{x}\,\overline{z}\, pdt_3)\, v[\overline{x}\mapsto\overline{d}] = \overline{d} \in (\mathsf{specialize}'\, pdt_3\, v)$$

whence for all v with $\overline{x}\cdot\overline{y} \subseteq \text{dom } v$,

$$\begin{aligned}\mathsf{specialize}'\, pdt\, v &= \mathsf{specialize}'\,(\mathsf{insert}\,\emptyset\,\overline{x}\,\overline{z}\, pdt_3)\, v \\ &= v(\overline{x}) \in (\mathsf{specialize}'\, pdt_3\, v) \\ &= \mathsf{let}\, M = [[\![\overline{t}]\!]_{v[\overline{z}\mapsto\overline{d}]} \mid v[\overline{z}\mapsto\overline{d}], i \vDash_\sigma \varphi_1, \overline{d} \in \mathbb{D}^{|\overline{z}|}]\ \mathsf{in} \\ &\quad\ v(\overline{x}) \in \omega(M) \wedge |\overline{y}| > 0 \Longrightarrow M \neq [\,] \\ &= v, i \vDash_\sigma \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi_1 \\ &= v, i \vDash_\sigma \varphi.\end{aligned}$$

By Lemma 8, we get

Lemma 3. *Let* $\varphi = \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi_1$ *be monitorable and* $\overline{z} = \mathsf{fv}(\varphi_1) \setminus \overline{y}$*. Let* pdt_1 *be well-formed with respect to* $\mathsf{bv}(\varphi_1)$ *and adapted to some well-formed label sequence* $\mathsf{map}\,\mathsf{lbl\_of\_term}\,\overline{y}\cdot\overline{\ell}$ *for some* $\overline{\ell}$*. Assume that for any valuation* v*,* $\mathsf{specialize}\, pdt_1\, v = \textsc{Sat}_{\varphi_1}(v, i, \sigma)$*. Let* $pdt = \mathsf{aggregate}\,\overline{x}\,\overline{t}\,\overline{y}\,\overline{z}\, pdt_1$*. Then* $\mathsf{specialize}\, pdt\, v = \textsc{Sat}_{\varphi}(v, i, \sigma)$*.*

We sketch the proof of the following standard correctness theorem:

Theorem 4. *Let* Φ *be a closed MFOTL formula without* let *bindings that is monitorable as per Definition 4. Let* $\sigma = \langle(\tau, D)_{1\le i\le|\sigma|}\rangle$*. Then the sequence defined by*

$$\begin{aligned}\varphi_{-1} &= \mathsf{init}\,\Phi \\ (es_i, \varphi_i) &= \mathsf{eval}\,\varphi_{i-1}\,\sigma\, i \qquad\qquad i > 0\end{aligned}$$

is such that for any $i \ge 0$*, for* (ts, tp, pdt) *in* es_i *and for any valuation* v*, we have* $\tau_{tp} = ts$ *and* $\mathsf{specialize}'\, pdt\, v = (\textbf{\textit{if}}\ v, tp \vDash_\sigma \Phi\ \textbf{\textit{then}}\ \top\ \textbf{\textit{else}}\ \bot)$*.*

Proof (sketch). Denote

$$\begin{aligned}P(buf, \sigma = \langle(\tau, D)_{1\le i\le|\sigma|}\rangle, \Phi) := \forall(ts, tp, pdt) \in buf.\ &\tau_{tp} = ts \\ &\wedge \mathsf{specialize}'\, pdt\, v = (\mathbf{if}\ v, tp \vDash_\sigma \Phi\ \mathbf{then}\ \top\ \mathbf{else}\ \bot).\end{aligned}$$

Our algorithm fulfills the following invariant I_i for all i:
(I_i) All of the following hold:

1. $P(es_i, \sigma, \Phi)$
2. For any subformula $\mathsf{MAnd}\, \varphi_1\, \varphi_2\, (buf_1, buf_2)\, \overline{\ell}$ of φ and corresponding subformula $\Phi_1 \wedge \Phi_2$ of Φ, for $j \in \{1, 2\}$, $P(buf_j, \sigma, \Phi_i)$.
3. For any subformula $\mathsf{MSince}\, \varphi_1\, I\, \varphi_2\, (buf_1, buf_2)\, aux\, \overline{\ell}$ and corresponding subformula $\Phi_1\, \mathsf{S}_I\, \Phi_2$ of Φ, for $j \in \{1, 2\}$, $P(buf_j, \sigma, \Phi_i)$.

   Moreover, for any valuation v,

$$(\mathsf{specialize}'\ aux\ v).\mathsf{beta\_alphas\_in} = [\tau_i - \delta \mid \delta \in I \wedge v, i \vDash_\sigma \Phi_1\, \mathsf{S}_{[\delta,\delta]}\, \Phi_2]$$
$$(\mathsf{specialize}'\ aux\ v).\mathsf{beta\_alphas\_out} = [\tau_i - \delta \mid \delta \in [0, \min\, I) \wedge v, i \vDash_\sigma \Phi_1\, \mathsf{S}_{[\delta,\delta]}\, \Phi_2].$$

4. For any subformula $\mathsf{MUntil}\, \varphi_1\, I\, \varphi_2\, (buf_1, buf_2)\, tstps\, aux\, \overline{\ell}$ of φ and corresponding subformula $\Phi_1\, \mathsf{U}_I\, \Phi_2$ of Φ, for $j \in \{1, 2\}$, $P(buf_j, \sigma, \Phi_i)$.

   Moreover, if $|tstps| > 1$, then for all $(ts, tp) \in tstps.\ \tau_{tp} = ts$ and for any valuation v and $(ts, tp) = \mathsf{fst}\, tstps$, we have

$$(\mathsf{specialize}'\ aux\ v).\mathsf{beta\_in} = [(\tau_{i'}, i') \mid \tau_{i'} - ts \in I \wedge v, i' \vDash_\sigma \Phi_2]$$
$$(\mathsf{specialize}'\ aux\ v).\mathsf{beta\_out} = [(\tau_{i'}, i') \mid \tau_{i'} - ts \in [0, \min\, I) \wedge v, i' \vDash_\sigma \Phi_2]$$
$$(\mathsf{specialize}'\ aux\ v).\mathsf{n\_alpha\_in} = [(\tau_{i'}, i') \mid \tau_{i'} - ts \in I \wedge \neg(v, i' \vDash_\sigma \Phi_1)]$$
$$(\mathsf{specialize}'\ aux\ v).\mathsf{n\_alpha\_out} = [(\tau_{i'}, i') \mid \tau_{i'} - ts \in [0, \min\, I) \wedge \neg(v, i' \vDash_\sigma \Phi_1)].$$

These invariants are standard and similar to those used in previous work [9,4,32,25]. The algorithm itself follows a similar top-down approach as, e.g., VeriMon [4], producing a verdict for formula φ at timepoint i only when enough timepoints *after* i have been read to complete evaluate the truth value of φ at i for any valuation. The truth value of temporal operators is computed in a forward manner using standard unrolling formulae. The PDTs of subformulae are combined using the apply functions, which commute with specialize and specialize′ (see Algorithm 1). Lemma 3 provides the additional correctness arguments for our novel extended aggregations.

The conclusion follows from the invariant, Theorem 4, and Lemma 8.

A.3 Monitoring MFOTL with let bindings

We first extend our definition of monitorability to support let bindings:

Definition 12. *The fact that x does not appear in any function argument of φ, denoted $\mathsf{NF}(\varphi, x)$, is defined as follows:*

$$\mathsf{NF}'(\varphi, x, m) := \begin{cases} \mathsf{NF}'(\varphi_1, x, m) \cup \mathsf{NF}'(\varphi_2, x, m) \\ \qquad \text{if } \varphi = \varphi_1 \land \varphi_2 \text{ or } \varphi_1 \mathsf{S}_I \varphi_2 \text{ or } \varphi_1 \mathsf{U}_I \varphi_2 \\ \mathsf{NF}'(\varphi_1, x, m) \\ \qquad \text{if } \varphi = \exists z.\ \varphi_1 \text{ or } \neg\varphi_1 \\ \mathsf{NF}'(\varphi_2, x, m[e \mapsto (\varphi_1, \overline{x})]) \\ \qquad \text{if } \varphi = \mathsf{let}\, e(\overline{x}) = \varphi_1 \mathsf{~in~} \varphi_2 \\ \mathsf{NF}'(\varphi_1, \overline{x}_i, m) \\ \qquad \text{if } \varphi = e(\overline{t}), m(e) = (\varphi_1, \overline{x}), \exists 1 \le i \le |\overline{t}|.\ x \in \mathsf{fv}(\overline{t}_i) \\ \bot \quad \text{if } \varphi = e(\overline{t}), e \notin \operatorname{dom} m, \exists 1 \le i \le |\overline{t}|.\ x \in \mathsf{fv}(\overline{t}_i) \\ \top \quad \text{otherwise} \end{cases}$$

$$\mathsf{NF}(\varphi, x) := \mathsf{NF}'(\varphi, x, \emptyset)$$

Definition 13. *An MFOTL formula φ where all event names are either bound or in $\mathbb{E}$ is monitorable iff both of the following conditions hold:*

1. *For any quantified subformulae $Qx.\ \psi$ of φ, $Q \in \{\forall, \exists\}$ in the scope of bound predicates $e_1(\overline{t}_1) = \varphi_1, \ldots, e_k(\overline{t}_k) = \varphi_k$ (introduced in the order $e_1, \ldots, e_k$ above $Qx.\ \psi$), either $\Gamma_k \vdash \psi : PG^+_E(x)$ for some E, or $\Gamma_k \vdash \psi : PG^-_E(x)$ for some E, or $\mathsf{NF}'(\psi, x, m')$, where $m' = \{e_i \mapsto (\varphi_i, \overline{t}_i) \mid 1 \le i \le k\}$, $\Gamma_0 = \Gamma$, and for all $1 \le i \le k$, $\Gamma_i = \Gamma_{i-1} \cup \{\mathsf{let}_{e,i,p} : E \mid \Gamma_{i-1} \vdash \varphi_i : PG^p_E(\overline{t}_i)\}, \mathsf{let}_e : \bot$.*
2. *For any subformula $\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \psi$ of φ with bound predicates as in the previous case and any $z \in \mathsf{fv}(\psi) \setminus \overline{y}$, we have $\Gamma_k \vdash \psi : PG^+_E(z)$ for some E.*

Finally, we show that if Φ is monitorable as per Definition 13, unrolling let bindings in Φ yields a formula Φ' that is monitorable as per Definition 4. As Theorem 3 guarantees that our monitoring algorithm returns well-formed PDTs after unrolling let, this shows that the procedure that first unrolls let bindings and then uses Algorithm 6 returns well-formed PDTs.

We first formally define unrolling:

$$\mathsf{unroll}(\varphi, m) = \begin{cases} \mathsf{unroll}(\varphi_1, m) \land \mathsf{unroll}(\varphi_2, m) & \text{if } \varphi = \varphi_1 \land \varphi_2 \\ \exists x.\ \mathsf{unroll}(\varphi_1, m) & \text{if } \varphi = \exists x.\ \varphi_1 \\ \neg\mathsf{unroll}(\varphi_1, m) & \text{if } \varphi = \neg\varphi_1 \\ \mathsf{unroll}(\varphi_1, m)\ \mathsf{S}_I\ \mathsf{unroll}(\varphi_2, m) & \text{if } \varphi = \varphi_1 \mathsf{S}_I \varphi_2 \\ \mathsf{unroll}(\varphi_1, m)\ \mathsf{U}_I\ \mathsf{unroll}(\varphi_2, m) & \text{if } \varphi = \varphi_1 \mathsf{U}_I \varphi_2 \\ \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ (\mathsf{unroll}(\varphi_1, m)) & \text{if } \varphi = \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi_1 \\ \mathsf{unroll}(\varphi_2, m[e \mapsto (\mathsf{unroll}(\varphi_1, m), \overline{x})]) & \text{if } \varphi = \mathsf{let}\, e(\overline{x}) = \varphi_1 \mathsf{~in~} \varphi_2 \\ \varphi_1[\overline{t}/\overline{x}] & \text{if } \varphi = e(\overline{t}), m(e) = (\varphi_1, \overline{x}) \\ e(\overline{t}) & \text{if } \varphi = e(\overline{t}), e \notin \operatorname{dom} m \\ t \approx c & \text{if } \varphi = t \approx c \end{cases}$$

Just as we had done for variables, we henceforth assume that there is no shadowing of let bindings, i.e., the names of let bindings have been converted, if necessary, to ensure that each event name is bound at most once.

We prove:

Lemma 10. *If* $\Gamma \vdash \varphi_1 : PG^p_E(\overline{x}_k)$ *and* $\overline{t}_k = x$ *such that* $x \notin \mathsf{bv}(\varphi_1)$*, then* $\Gamma \vdash \varphi_1[\overline{t}/\overline{x}] : PG^p_E(x)$*.*

Proof. By straightforward induction on the PG rules.

Lemma 11. *If* $\mathsf{NF}'(\varphi, x, m)$*, then* $\mathsf{NF}'(\varphi, x, \mathsf{unroll}(\varphi, m))$*.*

Proof. By straightforward induction on φ.

Lemma 12. *If* φ *is monitorable as per Definition 13, then* $\mathsf{unroll}(\varphi, \emptyset)$ *is monitorable as per Definition 4.*

Proof. By structural induction on φ, we first prove:
(P_φ) Let m and Γ such that

1. $\mathrm{dom}\, m = \{e \mid \mathsf{let}_e \in \mathrm{dom}\, \Gamma\}$;
2. For all $e \in \mathrm{dom}\, m$ and $m(e) = (\varphi_1, \overline{x})$, we have $\mathsf{bv}(\varphi_1) \cap (\mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)) = \emptyset$, and for all $1 \leq i \leq |\overline{x}|$, $p' \in \{+, -\}$, if $\mathsf{let}_{e,i,p'} : E' \in \Gamma$ then $\Gamma \vdash \varphi_1 : \mathrm{PG}^{p'}_{E'}(\overline{x}_i)$;
3. $\Gamma \vdash \varphi : \mathrm{PG}^p_E(x)$.

Then $\Gamma \vdash \mathsf{unroll}(\varphi, m) : \mathrm{PG}^p_E(x)$.

- If $\varphi = e(\overline{t})$, then given 3., two PG rules can have been applied: $\mathbb{E}^+_{\mathrm{PG}}$ or $\mathsf{let}_{\mathrm{PG}}$. If $\mathbb{E}^+_{\mathrm{PG}}$ has been applied, then we have $E = \{e\}$, $p = +$, and $1 \leq k \leq |\overline{t}|$ such that $x = \overline{t}_k$, and $\mathsf{let}_e \notin \mathrm{dom}\, \Gamma$. In this case, assumption 1. gives $e \in \mathrm{dom}\, m$ and $\mathsf{unroll}(\varphi, m) = \varphi$, and assumption 3. yields the conclusion. If $\mathsf{let}_{\mathrm{PG}}$ has been applied, then we have $1 \leq k \leq |\overline{t}|$ such that $x = \overline{t}_k$, $\mathsf{let}_e \in \mathrm{dom}\, \Gamma$, and $\Gamma(\mathsf{let}_{e,k,p}) = E$. By assumption 2., we get φ_1 and $\overline{x}$ such that $m(e) = (\varphi_1, \overline{x})$ and $\Gamma \vdash \varphi_1 : \mathrm{PG}^p_E(\overline{x}_k)$ and $x \notin \mathsf{bv}(\varphi_1)$. Furthermore, $\mathsf{unroll}(\varphi, m) = \varphi_1[\overline{t}/\overline{x}]$. Using Lemma 10, we obtain $\Gamma \vdash \varphi_1[\overline{t}/\overline{x}] : \mathrm{PG}^p_E(\overline{t_k})$, i.e., $\Gamma \vdash \mathsf{unroll}(\varphi, m) : \mathrm{PG}^p_E(x)$.
- If $\varphi = t \approx c$, then $\mathsf{unroll}(\varphi, m) = \varphi$ and 3. yields the conclusion.
- If $\varphi = \varphi_1 \wedge \varphi_2$, assume P_{φ_1} and P_{φ_2}. Given 3., three PG rules can have been applied: $\wedge^{\mathrm{L}+}_{\mathrm{PG}}$, $\wedge^{\mathrm{R}+}_{\mathrm{PG}}$, and $\wedge^-_{\mathrm{PG}}$. In the first case, we have $p = +$ and $\Gamma \vdash \varphi_1 : \mathrm{PG}^+_E(x)$. Since $\mathsf{fv}(\varphi_1) \subseteq \mathsf{fv}(\varphi)$ and $\mathsf{bv}(\varphi_1) \subseteq \mathsf{bv}(\varphi)$, we can use 1.–2. and P_{φ_1} to obtain $\Gamma \vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^+_E(x)$. Now, $\mathsf{unroll}(\varphi_1 \wedge \varphi_2, m) = \mathsf{unroll}(\varphi_1, m) \wedge \mathsf{unroll}(\varphi_2, m)$, and hence we apply $\wedge^+_{\mathrm{PG}}$ using $\Gamma \vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^+_E(x)$ to show $\Gamma \vdash \mathsf{unroll}(\varphi_1 \wedge \varphi_2, m) : \mathrm{PG}^+_E(x)$. The proof is similar for $\wedge^{\mathrm{R}+}_{\mathrm{PG}}$ exchanging the role of φ_1 and φ_2. In the third case, we have $p = -$ and $\Gamma \vdash \varphi_1 : \mathrm{PG}^-_{E_1}(x)$, $\Gamma \vdash \varphi_2 : \mathrm{PG}^-_{E_2}(x)$, $E = E_1 \cup E_2$. Since $\mathsf{fv}(\varphi) = \mathsf{fv}(\varphi_1) \cup \mathsf{fv}\varphi_2$ and $\mathsf{bv}(\varphi) = \mathsf{bv}(\varphi_1) \cup \mathsf{bv}\varphi_2$, we can again use 1.–2. with P_{φ_1} and P_{φ_2} to show $\Gamma \vdash \mathsf{unroll}(\varphi_i, m) : \mathrm{PG}^+_{E_i}(x)$, $i \in \{1, 2\}$. We then apply $\wedge^-_{\mathrm{PG}}$ to get $\Gamma \vdash \mathsf{unroll}(\varphi, m) : \mathrm{PG}^+_E(x)$.

– If $\varphi = \exists z.\, \varphi_1$, assume P_{φ_1}. Given 3., only rule $\exists_{\mathrm{PG}}$ can have been applied. We get $x \neq z$ and $\Gamma \vdash \varphi : \mathrm{PG}^p_E(x)$. Since $\mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi) = \mathsf{fv}(\varphi_1) \cup \mathsf{bv}(\varphi_1)$, we can use 1.–2. and P_{φ_1} to obtain $\Gamma \vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^+_E(x)$. Now, $\mathsf{unroll}(\exists z.\ \varphi_1, m) = \exists z.\ \mathsf{unroll}(\varphi_1, m)$, and hence we apply $\exists_{\mathrm{PG}}$ using $\Gamma \vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^+_E(x)$ and $z \neq x$ to show $\Gamma \vdash \mathsf{unroll}(\exists z.\ \varphi_1, m) : \mathrm{PG}^+_E(x)$.
– If $\varphi = \neg\varphi_1$, the proof is similar to the previous case.
– If $\varphi = \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi_1$, assume P_{φ_1}. Given 3., two rules can have been applied: $\mathsf{agg}_{\mathrm{PG},\overline{x}}$ or $\mathsf{agg}_{\mathrm{PG},\overline{y}}$. In the former case, $p = +$, $v \in \overline{x}$ and $\forall u \in \mathsf{fv}(\overline{t}).\ \exists E \subseteq \Gamma^{-1}(\mathbb{C}).\ \Gamma \vdash \varphi_1 : \mathrm{PG}^+_E(u)$. Since $\mathsf{fv}(\varphi_1) \cup \mathsf{bv}(\varphi_1) \subseteq \mathsf{fv}(\varphi_1) \cup \mathsf{bv}(\varphi_1) \cup \overline{x} = \mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)$, we can use 1.–2. and P_{φ_1} to obtain $\Gamma \vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^+_E(u)$ for all $u \in \mathsf{fv}(\overline{t})$. Since $\mathsf{unroll}(\varphi, m) = \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ (\mathsf{unroll}(\varphi_1, m))$, we can apply $\mathrm{PG}^+_{\mathsf{agg},x}$ again to obtain $\Gamma \vdash \mathsf{unroll}\varphi m : \mathrm{PG}^+_E(x)$. The other case is similar.
– If $\varphi = \mathsf{let}\, e(\overline{x}) = \varphi_1\ \mathsf{in}\ \varphi_2$, assume P_{φ_1} and P_{φ_2}. Given 3., only rule let or $\mathsf{let}_{\mathbb{O}}$ can have been applied. We get $\Gamma' \vdash \varphi_2 : \mathrm{PG}^p_E(x)$, $\Gamma' = \Gamma \cup \{\mathsf{let}_{e,i,p} \mapsto E \mid \Gamma \vdash \varphi_1 : \mathrm{PG}^p_E(\overline{x}_i)\}, \mathsf{let}_e : \bot$. Now, $\mathsf{unroll}(\varphi, m) = \mathsf{unroll}(\varphi_2, m[e \mapsto (\overline{x}, \mathsf{unroll}(\varphi_1, m))])$. We will use P_{φ_2} to conclude, using $m' = m[e \mapsto (\overline{x}, \mathsf{unroll}(\varphi_1, m))]$ and Γ' as above. To do this, we need to prove 1.–3. for φ_2, m', and Γ' (henceforth denoted 1.'–3'). Property 1.' follows from assumption 1. and the fact that m' and Γ' extend m and Γ by mapping e and let_e, respectively. For property 2.', we see using property 2. and the fact that $\mathsf{fv}(\varphi_2) \cup \mathsf{bv}(\varphi_2) \subseteq \mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)$ it is enough to prove the desired equivalence for e. That is, we must show that for all $1 \leq i \leq |\overline{x}|$, $p' \in \{+, -\}$, if $\mathsf{let}_{e,i,p'} : E' \in \Gamma'$ then $\Gamma' \vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^{p'}_{E'}(\overline{x}_i)$. Let i, p' as above. By definition of Γ', we have that $\mathsf{let}_{e,i,p'} : E' \in \Gamma'$ implies $\Gamma \vdash \varphi_1 : \mathrm{PG}^{p'}_{E'}(\overline{x}_i)$. If $\Gamma \vdash \varphi_1 : \mathrm{PG}^{p'}_{E'}(\overline{x}_i)$, then using P_{φ_1}, 1.–3., and $\mathsf{fv}(\varphi_2) \cup \mathsf{bv}(\varphi_2) \subseteq \mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)$ we get $\Gamma \vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^{p'}_{E'}(\overline{x}_i)$. By our assumption that each event is bound at most once by let, this implies $\Gamma' \vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^{p'}_{E'}(\overline{x}_i)$ as the additional types for e cannot affect φ_1. For property 3.', we use $\Gamma' \vdash \varphi_2 : \mathrm{PG}^p_E(x)$ and the fact that PG types do not depend on any $\Gamma(e)$ to obtain $\Gamma' \vdash \varphi_2 : \mathrm{PG}^p_E(x)$. This concludes the proof.

Using P_φ for all φ, we now prove by induction on $|\varphi| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'|$ (where $|\varphi|$ is the number of operators of φ), generalizing on φ, m, and Γ:

$(Q_{\varphi,m,\Gamma})$ Assume that:

1. All event names in φ are either bound, in $\mathbb{E}$, or in $\mathrm{dom}\, m$;
2. $\mathrm{dom}\, m = \{e \mid \mathsf{let}_e \in \mathrm{dom}\, \Gamma\}$;
3. For all $e \in \mathrm{dom}\, m$ and $m(e) = (\varphi_1, \overline{x})$, we have $\mathsf{bv}(\varphi_1) \cap (\mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)) = \emptyset$, and for all $1 \leq i \leq |\overline{x}|$, $p' \in \{+, -\}$, if $\mathsf{let}_{e,i,p'} : E' \in \Gamma$ then $\Gamma \vdash \varphi_1 : \mathrm{PG}^{p'}_{E'}(\overline{x}_i)$;
4. (R_φ) for any quantified subformula $Qx.\ \psi$ of φ, $Q \in \{\forall, \exists\}$ in the scope of bound predicates $e_1(\overline{t}_1) = \varphi_1, \ldots, e_k(\overline{t}_k) = \varphi_k$ (introduced in the order $e_1, \ldots, e_k$ above $Qx.\ \psi$), either $\Gamma_k \vdash \psi : \mathrm{PG}^+_E(x)$ for some E, or $\Gamma_k \vdash \psi : \mathrm{PG}^-_E(x)$ for some E, or $\mathsf{NF}'(\psi, x, m')$, where $m' = m[e_i \mapsto (\varphi_i, \overline{t}_i) \mid 1 \leq i \leq$

$k]$, $\Gamma_0 = \Gamma$, and for all $1 \leq i \leq k$, $\Gamma_i = \Gamma_{i-1} \cup \{\mathsf{let}_{e,i,p} \mapsto E \mid \Gamma_{i-1} \vdash \varphi_i : \mathrm{PG}^p_E(\bar{t}_i)\}, \mathsf{let}_e : \bot$.

5. (S_φ) For any subformula $\overline{x} \leftarrow \omega(\bar{t}; \overline{y})\ \psi$ of φ with bound predicates as in the previous case and any $z \in \mathsf{fv}(\psi) \setminus \overline{y}$, we have $\Gamma_k \vdash \psi : \mathrm{PG}^+_E(z)$ for some E.
6. For any $m(e) = (\varphi', \overline{x})$, φ' does not contain any let bindings and is monitorable as per Definition 4.

Then $\mathsf{unroll}(\varphi, m)$ is monitorable as per Definition 4, i.e.

1'. For any quantified subformula $Qx.\ \psi$ of $\mathsf{unroll}(\varphi, m)$, $Q \in \{\forall, \exists\}$, either $\vdash \psi : \mathrm{PG}^+_E(x)$ for some E, or $\vdash \psi : \mathrm{PG}^-_E(x)$ for some E, or $\mathsf{NF}'(\psi, x, m)$.
2'. For any subformula $\overline{x} \leftarrow \omega(\bar{t}; \overline{y})\ \psi$ of $\mathsf{unroll}(\varphi, m)$ and any $z \in \mathsf{fv}(\psi) \setminus \overline{y}$, we have $\vdash \psi : \mathrm{PG}^+_E(z)$ for some E.

The property Q_φ implies our lemma, since if all event names are either bound or in $\mathbb{E}$ once can always set $m = \emptyset$ and $\Gamma = \emptyset$ to satisfy 1.–3., 6.

Let us prove $Q_{\varphi,m,\Gamma}$, assuming that $Q_{\varphi',m',\Gamma'}$ holds for all φ', m', Γ' such that $|\varphi'| + \sum_{m'(e)=(\varphi',\overline{x})} |\varphi'| < |\varphi| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'|$.

- If $\varphi = e(\bar{t})$, $e \in \operatorname{dom} m$, $m(e) = (\varphi_1, \overline{x})$, then $\mathsf{unroll}(\varphi, m) = \varphi_1[\bar{t}/\overline{x}]$. By 5., formula φ_1 does not contain any let and is monitorable. Since substituting free variables does not affect monitorability, then $\varphi_1[\bar{t}/\overline{x}]$ is monitorable.
- If $\varphi = e(\bar{t})$, $e \notin \operatorname{dom} m$, then $\mathsf{unroll}(\varphi, m) = e(\bar{t})$, which is trivially monitorable.
- If $\varphi = t \approx c$, then $\mathsf{unroll}(\varphi, m) = t \approx c$, which is trivially monitorable.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $\mathsf{unroll}(\varphi, m) = \mathsf{unroll}(\varphi_1, m) \wedge \mathsf{unroll}(\varphi_2, m)$. Clearly, $|\varphi_1| + \sum_{m'(e)=(\varphi',\overline{x})} |\varphi'| < |\varphi| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'|$ and $|\varphi_1| + \sum_{m'(e)=(\varphi',\overline{x})} |\varphi'| < |\varphi| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'|$. Hence, $Q_{\varphi_1,m,\Gamma}$ and $Q_{\varphi_2,m,\Gamma}$ hold. One then checks that 1.–6. still hold for (φ_1, m, Γ) and (φ_2, m, Γ), since φ_1 and φ_2 are subformulae of φ. Hence, both $\mathsf{unroll}(\varphi_1, m)$ and $\mathsf{unroll}(\varphi_2, m)$ are monitorable as per Definition 4, and $\mathsf{unroll}(\varphi_1, m) \wedge \mathsf{unroll}(\varphi_2, m)$ is monitorable.
- The proof is similar for $\varphi = \varphi_1\ \mathsf{S}_I\ \varphi_2$ and $\varphi = \varphi_1\ \mathsf{U}_I\ \varphi_2$.
- If $\varphi = \neg\varphi_1$, then $\mathsf{unroll}(\varphi, m) = \neg\mathsf{unroll}(\varphi_1, m)$. We have $|\varphi_1| + \sum_{m'(e)=(\varphi',\overline{x})} |\varphi'| < |\varphi| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'|$. Hence, $Q_{\varphi_1,m,\Gamma}$ holds. One then checks that 1.–6. still hold for (φ_1, m, Γ), since φ_1 is a subformula of φ. Hence, $\mathsf{unroll}(\varphi_1, m)$ and $\mathsf{unroll}(\varphi_2, m)$ is monitorable as per Definition 4, and $\neg\mathsf{unroll}(\varphi_1, m)$ is monitorable.
- If $\varphi = \exists x.\ \varphi_1$, then $\mathsf{unroll}(\varphi, m) = \exists x.\ \mathsf{unroll}(\varphi_1, m)$. We have $|\varphi_1| + \sum_{m'(e)=(\varphi',\overline{x})} |\varphi'| < |\varphi| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'|$. Hence, $Q_{\varphi_1,m,\Gamma}$ holds. One then checks that 1.–6. still hold for (φ_1, m, Γ), since φ_1 is a subformula of φ. Hence, $\mathsf{unroll}(\varphi_1, m)$ and $\mathsf{unroll}(\varphi_2, m)$ is monitorable as per Definition 4, and $\mathsf{unroll}(\varphi_1, m)$ is monitorable. To show that $\exists x.\ \mathsf{unroll}(\varphi_1, m)$ is monitorable, we must additionally prove that either $\vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^+_E(x)$, $\vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^-_E(x)$, or x does not appear inside any function argument in $\mathsf{unroll}(\varphi_1, m)$. By 4., we know that either $\Gamma \vdash \varphi_1 : \mathrm{PG}^+_E(x)$, or $\Gamma \vdash \varphi_1 : \mathrm{PG}^+_E(x)$, or x does not appear inside any function argument in φ_1, Hence, $\Gamma \vdash \varphi_1 : \mathrm{PG}^p_E(x)$ implies $\vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^p_E(x)$ by our lemma. If $\mathsf{NF}'(\varphi_1, x, m)$, then $\mathsf{NF}'(\mathsf{unroll}(\varphi_1, m), x, m)$ by Lemma 11.

– If $\varphi = \mathsf{let}\, e(\overline{x}) = \varphi_1 \mathsf{in} \varphi_2$, then $\mathsf{unroll}(\varphi, m) = \mathsf{unroll}(\varphi_2, m[e \mapsto (\mathsf{unroll}(\varphi_1, m), \overline{x})])$. Let $m' = m[e \mapsto (\mathsf{unroll}(\varphi_1, m), \overline{x})]$, $\Gamma' = \Gamma \cup \{\mathsf{let}_{e,i,p} : E \mid \Gamma \vdash \varphi_1 : \mathrm{PG}^p_E(\overline{x}_i)\}, \mathsf{let}_e : \bot$. We have $|\varphi_2| + \sum_{m'(e)=(\varphi',\overline{x})} |\varphi'| = |\varphi_1| + |\varphi_2| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'| = |\varphi| - 1 + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'| < |\varphi| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'|$, and hence $Q_{\varphi_2,m',\Gamma'}$ holds. If we can check 1.–6. for $Q_{\varphi_2,m',\Gamma'}$, our conclusion follows since $\mathsf{unroll}(\varphi, m) = \mathsf{unroll}(\varphi_2, m')$. Property 1. holds by assumption 1. and the fact that the only additional free event in φ_2 is e, which is in $\mathrm{dom}\, m'$. Property 2. holds by definition of m' and Γ'. Property 3. holds by assumption 3., the fact that $\mathsf{fv}(\varphi_2) \cup \mathsf{bv}(\varphi_2) \subseteq \mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)$, and the definition of Γ'. For property 4., observe that there is now one less bound predicate in any quantified subformula of φ_2 with respect to the corresponding subformula of φ. However, the new Γ_i in R_{φ_2} (henceforth denoted Γ'_i) are such that $\Gamma'_i = \Gamma_{i+1}$ for all i, since $\Gamma'_0 = \Gamma' = \Gamma \cup \{\mathsf{let}_{e,i,p} : E \mid \Gamma \vdash \varphi_1 : \mathrm{PG}^p_E(\overline{x}_i)\}, \mathsf{let}_e : \bot = \Gamma_1$. Hence, the latest Γ'_i, say, $\Gamma'_{k'}$, is equal to the previous latest Γ_i, Γ_k. Similarly, the new m' (henceforth denoted m'') is such that $m'' = m'$. Given a quantified subformula $Qx.\psi$ of φ_2, then by R_φ either $\Gamma_k = \Gamma'_{k'} \vdash \psi : \mathrm{PG}^p_E(x)$, which concludes this case, or $\mathsf{NF}'(\psi, x, m'') = \mathsf{NF}'(\psi, x, m')$, which concludes too. For property 5., the proof is as for property 4., using assumption 5. For property 6., observe that m' only adds a formula $\mathsf{unroll}(\varphi_1, m)$ to m, which by construction of unroll does not contain a let binding. Moreover, we have $|\varphi_1| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'| < |\varphi| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'|$, and hence $Q_{\varphi_1,m,\Gamma}$ holds. As before, we show that $\mathsf{unroll}(\varphi, m)$ is monitorable, yielding the conclusion.
– If $\varphi = \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi_1$, then $\mathsf{unroll}(\varphi, m) = \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ (\mathsf{unroll}(\varphi_1, m))$. We have $|\varphi_1| + \sum_{m'(e)=(\varphi',\overline{x})} |\varphi'| < |\varphi| + \sum_{m(e)=(\varphi',\overline{x})} |\varphi'|$. Hence, $Q_{\varphi_1,m,\Gamma}$ holds. One then checks that 1.–6. still hold for (φ_1, m, Γ), since φ_1 is a subformula of φ. Hence, $\mathsf{unroll}(\varphi_1, m)$ is monitorable as per Definition 4. To show that $\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ (\mathsf{unroll}(\varphi_1, m))$ is monitorable, we must additionally prove that for all $z \in \mathsf{fv}(\mathsf{unroll}(\varphi_1, m)) \setminus \overline{y}$, we have $\vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^+_E(z)$. By 5., we know that $\Gamma \vdash \varphi_1 : \mathrm{PG}^+_E(z)$ for all $z \in \mathsf{fv}(\mathsf{unroll}(\varphi_1, m)) \setminus \overline{y}$. Hence, $\Gamma \vdash \varphi_1 : \mathrm{PG}^+_E(x)$ implies $\vdash \mathsf{unroll}(\varphi_1, m) : \mathrm{PG}^+_E(x)$ by our lemma, which concludes the proof.

Similar to previous work [45], we can show that

Lemma 13. *Let v, i, σ, and φ such that all events in φ are bound or in $\mathbb{E}$. Then $v, i \vDash_\sigma \varphi \iff v, i \vDash_\sigma \mathsf{unroll}(\varphi, \emptyset)$.*

From this, Lemma 12 and Theorem 4, we get:

Theorem 5. *Let Φ be a closed MFOTL formula that is monitorable as per Definition 13. Let $\sigma = \langle (\tau, D)_{1 \le i \le |\sigma|} \rangle$. Then the sequence defined by*

$$\begin{aligned} \varphi_{-1} &= \mathsf{init}\,(\mathsf{unroll}(\Phi, \emptyset)) \\ (es_i, \varphi_i) &= \mathsf{eval}\, \varphi_{i-1}\, \sigma\, i \qquad\qquad i > 0 \end{aligned}$$

is such that for any $i \ge 0$, for (ts, tp, pdt) in es_i and for any valuation v, we have $\tau_{tp} = ts$ and $\mathsf{specialize}\, pdt\, v = (\textbf{\textit{if}}\, v, tp \vDash_\sigma \Phi\ \textbf{\textit{then}}\ \top\ \textbf{\textit{else}}\ \bot)$.

A.4 Enforcing EMFOTL with function applications

The full, extended set of EMFOTL typing rules is shown in Figure 16. It types functions to elements of the type lattice in Figure 7. Note the presence of new subtypes $\mathbb{C}_0$ and $\mathbb{S}_0$ of $\mathbb{C}_s$ and $\mathbb{S}_s$ that denote the fact that the respective formula can be caused or suppressed without any new event being caused (typically, by only *suppressing* events). In the rules, the symbol $\mathbb{C}_\alpha$ ($\mathbb{S}_\alpha$, resp.) stands for any of $\mathbb{C}$, $\mathbb{C}_0$, $\mathbb{C}_s$, or $\mathbb{C}_n$ (for any of $\mathbb{S}$, $\mathbb{S}_0$, $\mathbb{S}_s$, or $\mathbb{S}_n$, resp.).

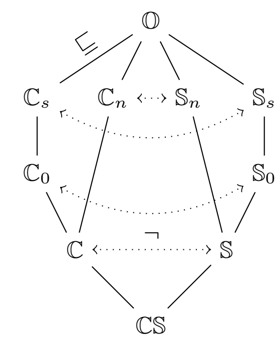


Fig. 17: Extended type lattice

Lemma 1. *$\mathsf{cl}(F, X)$ is finite for a finite set of stable functions F and a finite X.*

Proof. Let $D = \max(X \cup \bigcup_{f\in F} C_f)$ and $d \in \mathsf{cl}^i(F, X)$ for $i \geq 0$. By induction on i and the stability of the functions in F, we can show that $d \preceq D$. Since $\preceq$ is well-founded, then $Y = \{d \in \mathbb{D} \mid d \preceq D\}$ is finite and $\mathsf{cl}(F, X) \subseteq Y$ is finite.

Lemma 2. *Let $\overline{D} \in \mathbb{DB}^\omega$, $k \geq 1$, and disjoint $\mathbb{C}_s, \mathbb{C}_n \subseteq \mathbb{C}$ such that $\forall i \geq 2$,*

$$D_i - D_{i-1} \subseteq \{e(d_1, ..., d_{a(e)}) \mid e \in \mathbb{C} \wedge \forall i\, \exists f \in \mathsf{cl}(\mathbb{F}_s, D_{i-1}), \overline{d'} \in \mathsf{AD}_{D_i,\overline{\mathbb{C}_n}}(\varphi)^{a(f)}.\, d_i = \hat{f}(\overline{d'})\}$$
$$\cup \{e(d_1, ..., d_{a(e)}) \mid e \in \mathbb{C}_s \wedge \forall i\, \exists f \in \mathsf{cl}^k(\mathbb{F}, D_{i-1}), \overline{d'} \in \mathsf{AD}_{D_i,\overline{\mathbb{C}_n}}(\varphi)^{a(f)}.\, d_i = \hat{f}(\overline{d'})\},$$

where $\mathsf{AD}_{D_i,E}(\varphi) := \mathsf{AD}_{\langle(0,D_i)\rangle,E}(\varphi)$, then $\overline{D}$ is eventually constant.

Proof. By induction, each event $e(\overline{d}) \in D_i$ is such that each d_i is either in $X = \mathsf{cl}(\mathbb{F}_s, \mathsf{AD}_{D_0,\mathbb{E}}(\varphi))$ (if $e \in \mathbb{C}_s$), or in $Y = \mathsf{cl}^k(\mathbb{F}, X)$ (if $e \in \mathbb{C}_n$). By the definition of the set $\mathbb{F}_s$, both X and Y are finite.

Given our modified monitoring algorithm, the correctness proofs in [25] are still applicable since the main loop of the enforcement algorithm is unchanged. Only the termination lemma [25, Lemma 11] needs to be modified:

Lemma 14. *When $\Gamma \vdash \varphi : \mathbb{C}$, for all p, σ, X, τ, v, b, any call to $\mathsf{enf}^p_{\tau,b}(\varphi, \sigma, X, v)$ terminates.*

Proof. In the following, we consider the full pseudocode of the enforcement algorithm given by Hublet et al. [25]. This pseudocode differs from the simplified presentation in Algorithm 5 by enforcing all operators as they appear in the basic syntactic description of MFOTL (rather than $\longrightarrow$, $\forall$, $\blacklozenge$, etc.)

By structural induction on φ. As in [25], the only non-trivial cases are those involving a fix point computation: causation of $\wedge$ and suppression of $\exists$, aggregations, $\mathsf{S}_I^{\mathrm{LR}}$, and $\mathsf{S}_I^{\mathrm{LR}}$.

In all three cases, we observe that at each iteration of the fp function, $|D_S| + |D_C| + |X|$ grows strictly. If this quantity stops growing, then the loop is escaped and the algorithm terminates. Let $\sigma = \langle(\tau, D')_{1\leq i\leq|\sigma|}\rangle$ and $\Delta := \bigcup_{i=1}^{|\sigma|} D'_i$ be the set of all events occurring in σ. By contradiction, assume that some fix point computation in enf never terminates. For all i, denote by D_i the set $\{0\} \cup \mathsf{cl}^{\delta(\varphi)}(\Omega, \Delta) \cup \mathsf{const}(\varphi) \cup D_C$ at the end of the ith iteration. We now show that the sequence $\overline{D}$ satisfies the conditions of Lemma 2. Let $i \geq 2$ and $e(\overline{d}) \in D_i - D_{i-1}$. Then $e(\overline{d})$ has been caused in the ith iteration of fp. Let D_{Si} and D_{Ci} be the sets D_S and D_C at the beginning of this iteration. By systematic inspection of enf and the typing rules, we know that $e \in \mathbb{C}$ and either (i) $\Gamma(e) \in \mathbb{C}_s$ and $\overline{d}$ is obtained by applying only functions in $\mathbb{F}_s$ to some $\{v(x) \mid x \in \overline{x}\}$ where each $x \in \overline{x}$ is such that there exists $E \subseteq \mathbb{C}$ with $\Gamma(E) \subseteq \overline{\mathbb{C}_n}$ and $\Gamma(x) = \mathrm{PG}_E^+$; or (ii) $\Gamma(e) \in \mathbb{C}_n$ and $\overline{d}$ is obtained by applying any number of functions to some $\{v(x) \mid x \in \overline{x}\}$ where each $x \in \overline{x}$ is such that there exists $E \subseteq \mathbb{C}$ with $\Gamma(E) \subseteq \overline{\mathbb{C}_n}$ and $\Gamma(x) = \mathrm{PG}_E^+$. In both cases, for each $x \in \overline{x}$, the judgement $x : \mathrm{PG}_E^+$ can only have been introduced into Γ by the application of $\exists^{\mathbb{S}}$ or $\exists^{\mathbb{C}}$, If $\exists^{\mathbb{S}}$ was applied, then $\vdash \varphi' : \mathrm{PG}_E^+(x)$ for some φ'. If $\exists^{\mathbb{C}}$ was applied, then $E = \emptyset$ and $v(x) = 0$. By Lemma 4, we get $v(x) \in \mathsf{AD}^*_{\sigma_{..|\sigma|-1}\cdot(\tau_{|\sigma|}, D_{|\sigma|}\cup D_C\setminus D_S), E}(\varphi') \subseteq \mathsf{AD}^*_{\sigma_{..|\sigma|-1}\cdot(\tau_{|\sigma|}, D_{|\sigma|}\cup D_C\setminus D_S), E}(\varphi) \subseteq \mathsf{AD}^*_{\sigma_{..i-1}\cdot(\tau_{|\sigma|}, D_{|\sigma|}\cup D_C\setminus D_S), \overline{\mathbb{C}_n}}(\varphi)$. Hence, in (i), we get that each d_i is equal to $f(\overline{d'})$ where $\overline{d'} \in \mathsf{AD}_{D_{i-1}, \overline{\mathbb{C}_n}}(\varphi)$ and $f \in \mathsf{cl}(\mathbb{F}_s, D_{i-1})$, while in (ii), we get that each d_i is equal to $f(\overline{d'})$ where $\overline{d'} \in \mathsf{AD}_{D_{i-1}, \overline{\mathbb{C}_n}}(\varphi)$ and $f \in \mathsf{cl}^k(\mathbb{F}, D_{i-1})$ where k is the largest number of nested function calls in any term of φ. This is exactly the conditions in Lemma 2. Hence, we get that $\overline{D}$ is eventually constant from some iteration j. For the execution to continue indefinitely, either D_S or X must grow beyond iteration j. But X can only contain finitely many (say, m) future obligations determined by the syntax of φ (see [25]) and D_S is always a subset of existing events, i.e., a subset of D_j, which is finite. Hence, after at most $j + m + |D_j|$ iterations, the quantity $|D_S| + |D_C| + |X|$ must stop growing and the algorithm terminates.

A.5 Enforcing EMFOTL with aggregations

The following lemma shows the soundness of our approach to suppressing aggregations:

Lemma 15. *Let φ, $\overline{y}$ such that $|\overline{y}| > 0$, and $\overline{z} = z_1, \ldots, z_k = \mathsf{fv}(\varphi) \setminus \overline{y}$. For all v, i, and σ, we have*

$$v, i \vDash_\sigma \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi \Longrightarrow v, i \vDash_\sigma \exists z_1, \ldots, z_k.\ \varphi.$$

Proof. Let v such that $v, i \vDash_\sigma \overline{x} \leftarrow \omega(\overline{t}; \overline{y})\, \varphi$, $M = \left[[\![t]\!]_{v[\overline{z} \mapsto \overline{d}]} \mid v[\overline{z} \mapsto \overline{d}], i \vDash_\sigma \varphi, \overline{d} \in \mathbb{D}^{|\overline{z}|} \right]$, and $M \neq [\ ]$. We obtain v such that $\overline{d} \in \mathbb{D}^{|\overline{z}|}$ and $v[\overline{z} \mapsto \overline{d}], i \vDash_\sigma \varphi$. Hence, $v, i \vDash_\sigma \exists z_1, \ldots, z_k.\ \varphi$.

Observe that rule $\mathsf{agg}^{\mathbb{S}}$ is applicable iff k instances of $\exists^{\mathbb{S}}$ for $z_1, \ldots, z_k$ are applicable. Hence, the correctness theorem [25, Theorem 1] can be straightforwardly adapted to support aggregations.

A.6 Enforcing EMFOTL with let bindings

Similarly to monitoring, our enforcement algorithm unrolls let bindings before enforcing the formula. We only need to show:

Lemma 16. *If φ is enforceable, then $\mathsf{unroll}(\varphi, \emptyset)$ is enforceable.*

Proof. More generally, we prove by induction on $\Gamma \vdash \varphi : \tau$:

(P_φ) Let m, Γ, and $\tau \in \{\mathbb{C}, \mathbb{C}_0, \mathbb{C}_n, \mathbb{C}_s, \mathbb{S}, \mathbb{S}_0, \mathbb{S}_n, \mathbb{S}_s\}$ such that

1. $\operatorname{dom} m = \{e \mid \mathsf{let}_e \in \operatorname{dom} \Gamma\}$;
2. For all $e \in \operatorname{dom} m$ and $m(e) = (\varphi_1, \overline{x})$, we have $\mathsf{bv}(\varphi_1) \cap (\mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)) = \emptyset$, and for all $1 \leq i \leq |\overline{x}|$, $p' \in \{+, -\}$, if $\mathsf{let}_{e,i,p'} : E' \in \Gamma$ then $\Gamma \vdash \varphi_1 : \mathrm{PG}^{p'}_{E'}(\overline{x}_i)$ and if $e : \tau' \in \Gamma$ then $\Gamma \vdash \varphi_1 : \tau'$;
3. $\Gamma \vdash \varphi : \tau$.

Then $\Gamma \vdash \mathsf{unroll}(\varphi, m) : \tau$.

Setting $m = \emptyset$, $\tau = \mathbb{C}$, this proves the desired property.

- Rule cast: In this case, $\Gamma \vdash \varphi : \tau'$ and $\tau \sqsubseteq \tau'$. Then, by our induction hypothesis, we get $\Gamma \vdash \mathsf{unroll}(\varphi, m) : \tau'$. Applying rule cast again, we get $\Gamma \vdash \mathsf{unroll}(\varphi, m) : \tau$.
- Rules $\top^{\mathbb{C}}$, $\top^{\mathbb{S}}$: Trivial.
- Rule $\mathbb{E}^{\mathbb{C}_s}$: In this case, $e \in \mathbb{C} \vee \mathsf{let}_e \in \operatorname{dom} \Gamma$, $\Gamma(e) = \mathbb{C}_s$, $\forall x \in \bigcup_{i=1}^k \mathsf{fv}(t_i).\ \exists E \subseteq \Gamma^{-1}(\overline{\mathbb{C}_n})$. $\Gamma(x) = \mathrm{PG}^+_E$, and $\varphi = e(\overline{t})$, $\tau = \Gamma(e)$.
  If $e \in \mathbb{C}$, then $\mathsf{unroll}(\varphi, m) = \varphi$ and the conclusion follows. If $\mathsf{let}_e \in \operatorname{dom} \Gamma$, then $\mathsf{unroll}(\varphi, m) = \varphi_1[\overline{t}/\overline{x}]$ where $m(e) = (\varphi_1, \overline{x})$. By 2., we get $\Gamma \vdash \varphi_1 : \mathbb{C}_s$. Now, observe that our assumptions on $\bigcup_{i=1}^k \mathsf{fv}(t_i)$ and $\bigcup_{i=1}^k \mathsf{fn}(t_i)$ guarantee that even after substituting $\overline{t}$ into $\overline{x}$ in φ_1, all $\mathbb{E}^{\mathbb{C}_s}$ rules used in $\Gamma \vdash \varphi_1 : \mathbb{C}_s$ remain applicable. The PG rules for newly introduced variables (in quantifiers or aggregations) are unaffected since there is no shadowing. As a consequence, $\Gamma \vdash \varphi_1[\overline{t}/\overline{x}] : \mathbb{C}_s$, and hence $\Gamma \vdash \mathsf{unroll}(\varphi, m) : \mathbb{C}_s$.
- Rules $\mathbb{E}^{\mathbb{C}_n}$, $\mathbb{E}^{\mathbb{S}_0}$, $\mathbb{E}^{\mathbb{S}_n}$, $\mathbb{E}^{\mathbb{C}_n}$: Similar to the previous case.
- Rule $\neg^{\mathbb{C}}$: In this case, $\varphi = \neg\varphi_1$ and $\Gamma \vdash \varphi : \mathsf{S}_\alpha$. Since $\mathsf{fv}(\varphi) = \mathsf{fv}(\varphi_1)$ and $\mathsf{bv}(\varphi) = \mathsf{bv}(\varphi_1)$, our induction hypothesis yields $\Gamma \vdash \mathsf{unroll}(\varphi_1, m) : \mathbb{S}_\alpha$. Now, $\mathsf{unroll}(\varphi, m) = \neg\mathsf{unroll}(\varphi_1, m)$, hence we can use rule $\neg^{\mathbb{C}}$ to show $\Gamma \vdash \varphi : \mathbb{C}_\alpha$.
- Rules $\neg^{\mathbb{S}}$, $\exists^{\mathbb{C}}$, $\wedge^{\mathsf{SL}}$, $\wedge^{\mathsf{SR}}$, $\mathsf{S}^{\mathbb{C}}$, $\mathsf{S}^{\mathbb{S}\mathsf{L}}$, $\mathsf{U}^{\mathbb{S}}$, $\mathsf{U}^{\mathbb{C}\mathsf{R}}$, $\bigcirc^{\mathbb{C}}$, $\bigcirc^{\mathbb{S}}$: Similar to the previous case.

- Rule $\exists^{\mathbb{S}}$: In this case, $\varphi = \exists x.\ \varphi_1$, $\Gamma, x : \mathrm{PG}^{+}_{E} \vdash \varphi : \mathsf{S}_\alpha$, and $\Gamma \vdash \varphi : \mathrm{PG}^{+}_{E}(x)$. Let $\Gamma' = \Gamma, x : \mathrm{PG}^{+}_{E}$. Cleary, by our assumptions and the fact that $\mathsf{fv}(\varphi_1) \cup \mathsf{bv}(\varphi_1) \subseteq \mathsf{bv}(\varphi) \cup \mathsf{fv}(\varphi)$, the induction hypothesis is applicable to φ_1, Γ', and m. By the sublemma in Lemma 12, we additionally get $\Gamma \vdash \mathsf{unroll}(\varphi, m) : \mathrm{PG}^{+}_{E}(x)$. We obtain $\Gamma \vdash \mathsf{unroll}(\varphi_1, m) : \mathbb{S}_\alpha$ and apply $\exists^{\mathbb{S}}$ again to get $\Gamma \vdash \varphi : \mathbb{S}_\alpha$.
- Rule $\mathsf{agg}^{\mathbb{S}}$: Similar to the previous case.
- Rule $\wedge^{\mathbb{C}}$: In this case, $\varphi = \varphi_1 \wedge \varphi_2$, $\Gamma \vdash \varphi_1 : \mathbb{C}_\alpha$, and $\Gamma \vdash \varphi_2 : \mathbb{C}_\alpha$. Since $\mathsf{fv}(\varphi_1) \cup \mathsf{bv}(\varphi_1) \subseteq \mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)$ and $\mathsf{fv}(\varphi_2) \cup \mathsf{bv}(\varphi_2) \subseteq \mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)$, our induction hypothesis yields $\Gamma \vdash \mathsf{unroll}(\varphi_1, m) : \mathbb{C}_\alpha$ and $\Gamma \vdash \mathsf{unroll}(\varphi_2, m) : \mathbb{C}_\alpha$. Now, $\mathsf{unroll}(\varphi, m) = \mathsf{unroll}(\varphi_1, m) \wedge \mathsf{unroll}(\varphi_2, m)$, hence we can use rule $\wedge^{\mathbb{C}}$ to show $\Gamma \vdash \varphi : \mathbb{C}_\alpha$.
- Rules $\mathsf{S}^{\mathbb{S}\mathrm{LR}}$, $\mathsf{U}^{\mathbb{C}\mathrm{LR}}$: Similar to the previous case.
- Rule let: In this case, $\varphi = \mathsf{let}\, e(\overline{x}) = \varphi_1\, \mathsf{in}\, \varphi_2$, $\Gamma \vdash \varphi_1 : \tau_1$, $\Gamma', e : \tau_1 \vdash \varphi_2 : \tau_2$, where $\Gamma' = \Gamma' = \Gamma \cup \{\mathsf{let}_{e,i,p} : E \mid \Gamma \vdash \varphi_1 : \mathrm{PG}^{p}_{E}(x_i)\}, \mathsf{let}_e : \bot$. Let $m' = m[e \mapsto (\mathsf{unroll}(\varphi_1, m), \overline{x})]$. Since $\mathsf{fv}(\varphi_1) \cup \mathsf{bv}(\varphi_1) \subseteq \mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)$, our induction hypothesis on φ_1 applied with Γ and m yields $\Gamma \vdash \mathsf{unroll}(\varphi, m) : \tau_1$. Similarly, since $\mathsf{fv}(\varphi_2) \cup \mathsf{bv}(\varphi_2) \subseteq \mathsf{fv}(\varphi) \cup \mathsf{bv}(\varphi)$ and our induction hypothesis on φ_2 applied with $\Gamma' \cup \{e \mapsto \tau_1\}$ and m' yields $\Gamma' \vdash \mathsf{unroll}(\varphi, m') : \tau_2$. Now, $\mathsf{unroll}(\varphi, m) = \mathsf{unroll}(\varphi_2, m')$. Since Γ' difers from Γ only by the typing of e, let_e, and $\mathsf{let}_{e,i,p}$, which do not occur in $\mathsf{unroll}(\varphi_2, m')$, we conclude that $\Gamma \vdash \mathsf{unroll}(\varphi, m) : \tau_2$.
- Rule $\mathsf{let}_{\mathbb{O}}$: Similar to the previous case.

A.7 Wrapping up

Combining the results from the previous sections, we have:

Theorem 1. *Let φ be a closed formula with function applications, aggregations, and* let *bindings in our extended EMFOTL fragment. Let* fo *denote the set of future obligations,* enf' *the modified* enf *function, and* $\mathsf{unroll}(\varphi) := \mathsf{unroll}(\varphi, \emptyset)$. *Then* $\mathcal{E}_\varphi = (\mathcal{P}(\mathsf{fo}), \{(\mathsf{unroll}(\varphi), \emptyset, +)\}, \mathsf{enf}')$ *is sound with respect to* $\mathcal{L}(\varphi)$.

Proof. From the soundness theorem [25, Theorem 1] modified by Lemma 14 and Lemma 15, together with Lemma 16. The transformation $[\ ]_p$ in [25] is extended as follows to cover the suppression of aggregations after unrolling:

$$[\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi]_- = \exists z_1, \ldots, z_k.\ [\varphi]_- \qquad \text{where } \overline{z} = \mathsf{fv}(\varphi) \setminus \overline{y}.$$

Similarly to previous work [25, Appendix C], we can further restrict our fragment EMFOTL to a fragment TEMFOTL for which our algorithm provides transparent enforcement. This is done by (i) modifying the typing rules as described in Figure 18, where SRP denotes the set of *strictly relative-past formulae* introduced by Hublet et al. [24], and (ii) removing the rule $\mathsf{agg}^{\mathbb{S}}$. All other rules remain as in Figure 16.

$$\frac{\Gamma \vdash \varphi : \mathbb{S}_\alpha \quad \psi \in \mathsf{SRP}}{\Gamma \vdash \varphi \land \psi : \mathbb{S}_\alpha} \land^{\mathbb{S}\mathrm{L}} \quad \frac{\Gamma \vdash \psi : \mathbb{S}_\alpha \quad \varphi \in \mathsf{SRP}}{\Gamma \vdash \varphi \land \psi : \mathbb{S}_\alpha} \land^{\mathbb{S}\mathrm{R}}$$

$$\frac{0 \in I \quad \Gamma \vdash \psi : \mathbb{C}_\alpha \quad \varphi, \psi \in \mathsf{SRP}}{\Gamma \vdash \varphi \mathbin{\mathsf{S}_I} \psi : \mathbb{C}_\alpha} \mathsf{S}^{\mathbb{C}} \quad \frac{0 \notin I \quad \Gamma \vdash \varphi : \mathbb{S}_\alpha \quad \varphi, \psi \in \mathsf{SRP}}{\Gamma \vdash \varphi \mathbin{\mathsf{S}_I} \psi : \mathbb{S}_\alpha} \mathsf{S}^{\mathbb{S}\mathrm{L}}$$

$$\frac{0 \in I \quad \Gamma \vdash \varphi, \psi : \mathbb{S}_\alpha \quad \varphi, \psi \in \mathsf{SRP}}{\Gamma \vdash \varphi \mathbin{\mathsf{S}_I} \psi : \mathbb{S}_\alpha} \mathsf{S}^{\mathbb{S}\mathrm{LR}} \quad \frac{b \neq \infty \quad \Gamma \vdash \psi : \mathbb{C}_\alpha \quad \varphi \in \mathsf{SRP}}{\Gamma \vdash \varphi \mathbin{\mathsf{U}_{[0,b]}} \psi : \mathbb{C}_\alpha} \mathsf{U}^{\mathbb{C}\mathrm{R}}$$

$$\frac{\Gamma \vdash \psi : \mathbb{S}_\alpha \quad \varphi \in \mathsf{SRP}}{\Gamma \vdash \varphi \mathbin{\mathsf{U}_I} \psi : \mathbb{S}_\alpha} \mathsf{U}^{\mathbb{S}}$$

Fig. 18: Modified extended typing rules for TEMFOTL

Theorem 6. *Let φ be a closed formula with function applications, aggregations, and* let *bindings in our extended TEMFOTL fragment. Then $\mathcal{E}_\varphi$ is sound and transparent with respect to $\mathcal{L}(\varphi)$.*

Proof (sketch). By induction on φ, we first prove that $\varphi \in \text{TEMFOTL} \implies \mathsf{unroll}(\varphi) \in \text{TEMFOTL}$. Since aggregations are not transparently enforceable and function applications do not affect transparency, the rest of the proof is as in [25, Theorem 2].

B Typing of example formula (grubbs)

$$\begin{aligned}
\mathsf{grubbs} = {} & \mathsf{let}\, \mathsf{badReboot}(s, dc) = \varphi_1 \,\mathsf{in} \\
& \mathsf{let}\, \mathsf{cntReboots}(dc, c) = \varphi_2 \,\mathsf{in} \\
& \square_{[0s,\infty)}(\forall dc.\ \forall l.\ \varphi_3 \longrightarrow \varphi_4) \\
\varphi_1 = {} & \mathsf{reboot}(s, dc) \land \neg(\bullet_{[0s,\infty)}(\neg \mathsf{reboot}(s, dc) \mathbin{\mathsf{S}_{[0s,\infty)}} \mathsf{intendReboot}(s, dc))) \\
\varphi_2 = {} & c \leftarrow \mathtt{CNT}(i; dc)(\blacklozenge_{[0s,1799s]} \mathsf{badReboot}(s, dc) \land \mathsf{tp}(i)) \\
\varphi_3 = {} & (dc, l \leftarrow \mathtt{GRUBBS}(dc, c;)(\mathsf{cntReboots}(dc, c))) \land (l \approx 1) \\
\varphi_4 = {} & \mathsf{alert}(\mathsf{conc}(\mathsf{conc}(\texttt{"Data center "}, \mathsf{string_of_int}(dc)), \\
& \qquad \texttt{" has rebooted too often"})))
\end{aligned}$$

First, define:

$\Gamma_1 \equiv \mathsf{alert} : \mathbb{C}_n, \mathsf{reboot} : \mathbb{O}$

$\Gamma_2 \equiv \Gamma_1, \mathsf{let}_{\mathsf{badReboot}} : \bot, \mathsf{let}_{\mathsf{badReboot},2,+} : \{\mathsf{reboot}\}, \mathsf{badReboot} : \mathbb{O}$

$\Gamma_3 \equiv \Gamma_2, \mathsf{let}_{\mathsf{cntReboots}} : \bot, \mathsf{let}_{\mathsf{cntReboots},1,+} : \{\mathsf{reboot}\}, \mathsf{let}_{\mathsf{cntReboots},2,+} : \{\mathsf{tp}\}, \mathsf{cntReboots} : \mathbb{O}$

$\Gamma_3' \equiv \Gamma_3, dc : \mathrm{PG}^+_{\{\mathsf{reboot}\}}$

$\Gamma_4 \equiv \Gamma_3', l : \mathrm{PG}^+_{\{\mathsf{reboot}\}}$

Then, consider the subproofs $P_4, P_3, P_2^1, P_2^2, P_1$:

$$\dfrac{\dfrac{\mathsf{alert}\in\mathbb{C}\quad \mathit{\Gamma}_4(\mathsf{alert})=\mathbb{C}_n\quad \mathit{\Gamma}_4(x)=\mathrm{PG}^+_{\{\mathsf{reboot}\}}\quad \mathit{\Gamma}_4(\mathsf{reboot})=\mathbb{O}}{\mathit{\Gamma}_4\vdash\varphi_4:\mathbb{C}_n}\;\mathbb{E}^{\mathbb{C}_n}}{P_4}$$

For $v \in \{dc, l\}$:

$$\dfrac{\dfrac{\begin{array}{c}\mathsf{let}_{\mathsf{cntReboots}}\in\mathrm{dom}\ \mathit{\Gamma}_3\\ \mathit{\Gamma}_3(\mathsf{let}_{\mathsf{cntReboots},1,+})=\{\mathsf{reboot}\}\end{array}}{\mathit{\Gamma}_3\vdash\mathsf{cntReboots}(dc,c):\mathrm{PG}^+_{\{\mathsf{reboot}\}}(dc)}\;\mathsf{let}_{\mathrm{PG}}}{P_3'(v)}$$

$$\dfrac{\dfrac{v\in[dc,l]\quad \dfrac{dc\in[dc,l]\quad P_3'(v)\quad \dfrac{\begin{array}{c}\mathsf{let}_{\mathsf{cntReboots}}\in\mathrm{dom}\ \mathit{\Gamma}_3\\ \mathit{\Gamma}_3(\mathsf{let}_{\mathsf{cntReboots},2,+})=\{\mathsf{tp}\}\end{array}}{\mathit{\Gamma}_3\vdash\mathsf{cntReboots}(dc,c):\mathrm{PG}^+_{\{\mathsf{tp}\}}(c)}\;\mathsf{let}_{\mathrm{PG}}}{\mathit{\Gamma}_3\vdash dc,l\leftarrow\mathtt{GRUBBS}(dc,c;)(\mathsf{cntReboots}(dc,c)):\mathrm{PG}^+_{\{\mathsf{reboot},\mathsf{tp}\}}(dc)}\;\mathsf{agg}_{\mathrm{PG},\overline{x}}}{\mathit{\Gamma}_3\vdash\varphi_3:\mathrm{PG}^+_{\{\mathsf{reboot},\mathsf{tp}\}}(v)}\;\wedge^{\mathrm{L}+}_{\mathrm{PG}}}{P_3(v)}$$

$$\dfrac{\dfrac{dc\in[dc]\quad \dfrac{\dfrac{\dfrac{\begin{array}{c}\mathsf{let}_{\mathsf{badReboot}}\in\mathrm{dom}\ \mathit{\Gamma}_2\\ \mathit{\Gamma}_2(\mathsf{let}_{\mathsf{badReboot},2,+})=\{\mathsf{reboot}\}\end{array}}{\mathit{\Gamma}_2\vdash\mathsf{badReboot}(s,dc):\mathrm{PG}^+_{\{\mathsf{reboot}\}}(dc)}\;\mathsf{let}_{\mathrm{PG}}}{\mathit{\Gamma}_2\vdash\mathsf{badReboot}(s,dc)\wedge\mathsf{tp}(i):\mathrm{PG}^+_{\{\mathsf{reboot}\}}(dc)}\;\wedge^{\mathrm{L}+}_{\mathrm{PG}}}{\mathit{\Gamma}_2\vdash\blacklozenge_{[0s,1799s]}\,\mathsf{badReboot}(s,dc)\wedge\mathsf{tp}(i):\mathrm{PG}^+_{\{\mathsf{reboot}\}}(dc)}\;\blacklozenge^+_{\mathrm{PG}}}{\mathit{\Gamma}_2\vdash\varphi_2:\mathrm{PG}^+_{\{\mathsf{reboot}\}}(dc)}\;\mathsf{agg}_{\mathrm{PG},\overline{y}}}{P_2^1}$$

$$\dfrac{\dfrac{c\in[c]\quad \{\mathsf{tp}\}\subseteq\mathit{\Gamma}_2^{-1}(\overline{\mathbb{C}})\quad \dfrac{\dfrac{\dfrac{}{\mathit{\Gamma}_2\vdash\mathsf{tp}(i):\mathrm{PG}^+_{\{\mathsf{tp}\}}(i)}\;\mathbb{E}^+_{\mathrm{PG}}}{\mathit{\Gamma}_2\vdash\mathsf{badReboot}(s,dc)\wedge\mathsf{tp}(i):\mathrm{PG}^+_{\{\mathsf{tp}\}}(i)}\;\wedge^{\mathrm{R}+}_{\mathrm{PG}}}{\mathit{\Gamma}_2\vdash\blacklozenge_{[0s,1799s]}\,\mathsf{badReboot}(s,dc)\wedge\mathsf{tp}(i):\mathrm{PG}^+_{\{\mathsf{tp}\}}(i)}\;\blacklozenge^+_{\mathrm{PG}}}{\mathit{\Gamma}_2\vdash\varphi_2:\mathrm{PG}^+_{\{\mathsf{tp}\}}(c)}\;\mathsf{agg}_{\mathrm{PG},\overline{x}}}{P_2^2}$$

$$\dfrac{\dfrac{\dfrac{}{\mathit{\Gamma}_1\vdash\mathsf{reboot}(s,dc)}\;\mathbb{E}^+_{\mathrm{PG}}}{\mathit{\Gamma}_1\vdash\varphi_1:\mathrm{PG}^+_{\{\mathsf{reboot}\}}(dc)}\;\wedge^{\mathrm{L}+}_{\mathrm{PG}}}{P_1}$$

The final proof is as follows:

$$
\dfrac{\dfrac{\dfrac{P_3(l)}{\Gamma_3' \vdash \varphi_3 : \mathrm{PG}^+_{\{\mathsf{reboot}\}}(l)}}{\Gamma_3' \vdash \varphi_3 \longrightarrow \varphi_4 : \mathrm{PG}^-_{\{\mathsf{reboot}\}}(l)}\longrightarrow^-_{\mathrm{PG}} \qquad \dfrac{\dfrac{\mathbb{C} \sqsubseteq \mathbb{C}_n \quad \dfrac{P_4}{\Gamma_4 \vdash \varphi_4 : \mathbb{C}_n}}{\Gamma_4 \vdash \varphi_4 : \mathbb{C}}\,\mathsf{cast}}{\Gamma_4 \vdash \varphi_3 \longrightarrow \varphi_4 : \mathbb{C}}\longrightarrow^{\mathbb{CR}}}{\dfrac{\Gamma_3' \vdash \forall l.\ \varphi_3 \longrightarrow \varphi_4 : \mathbb{C}}{P'}}\,\forall^{\mathbb{C}}
$$

$$
\dfrac{P_1 \qquad \dfrac{P_2^1 \quad P_2^2 \quad \dfrac{\dfrac{P' \quad \dfrac{\dfrac{P_3(dc)}{\Gamma_3 \vdash \varphi_3 : \mathrm{PG}^+_{\{\mathsf{reboot}\}}(dc)}}{\Gamma_3 \vdash \varphi_3 \longrightarrow \varphi_4 : \mathrm{PG}^-_{\{\mathsf{reboot}\}}(dc)}\longrightarrow^-_{\mathrm{PG}}}{\Gamma_3 \vdash \forall dc.\ \forall l.\ \varphi_3 \longrightarrow \varphi_4 : \mathbb{C}}\,\forall^{\mathbb{C}}}{\Gamma_3 \vdash \Box_{[0s,\infty)}\, \forall dc.\ \forall l.\ \varphi_3 \longrightarrow \varphi_4 : \mathbb{C}}\,\Box^{\mathbb{C}}}{\Gamma_2 \vdash \mathsf{let}\ \mathsf{cntReboots}(dc, c) = \varphi_2\ \mathsf{in}\ \ldots : \mathbb{C}}\,\mathsf{let}_{\mathbb{O}}}{\Gamma_1 \vdash \mathsf{grubbs} : \mathbb{C}}\,\mathsf{let}_{\mathbb{O}}
$$

C Relevant event names and future obligations

The set $\mathsf{RFO}(\varphi) := \mathsf{RFO}^+(\varphi)$ of relevant future obligations is computed as follows after unrolling let bindings [25]:

$$
\begin{aligned}
\mathsf{RFO}^p(\neg\varphi_1) &= \mathsf{RFO}^{-p}(\varphi_1)\\
\mathsf{RFO}^+(\varphi_1 \wedge \varphi_2) &= \mathsf{RFO}^+(\varphi_1) \cup \mathsf{RFO}^+(\varphi_2)\\
\mathsf{RFO}^-(\varphi_1 \wedge^{\mathbb{SL}} \varphi_2) &= \mathsf{RFO}^-(\varphi_1)\\
\mathsf{RFO}^-(\varphi_1 \wedge^{\mathbb{SR}} \varphi_2) &= \mathsf{RFO}^-(\varphi_2)\\
\mathsf{RFO}^p(\exists x.\ \varphi_1) &= \mathsf{RFO}^p(\varphi_1)\\
\mathsf{RFO}^p(\bigcirc_I \varphi_1) &= \{(\lambda\tau.\ (\neg\mathtt{TP})\ \mathsf{U}_{I-(\tau-ts)}\ (\mathtt{TP} \wedge \varphi_1), v, p) \mid ts, v\}\\
\mathsf{RFO}^+(\varphi_1\ \mathsf{S}_I\ \varphi_2) &= \mathsf{RFO}^+(\varphi_2)\\
\mathsf{RFO}^-(\varphi_1\ \mathsf{S}_I^{\mathbb{SL}}\ \varphi_2) &= \mathsf{RFO}^-(\varphi_1)\\
\mathsf{RFO}^-(\varphi_1\ \mathsf{S}_I^{\mathbb{SR}}\ \varphi_2) &= \mathsf{RFO}^-(\varphi_2)\\
\mathsf{RFO}^+(\varphi_1\ \mathsf{U}_I^{\mathbb{C}\mathrm{LR}}\ \varphi_2) &= \mathsf{RFO}^+(\varphi_1) \cup \mathsf{RFO}^+(\varphi_2) \cup \{\lambda\tau.\ (\mathtt{TP} \to \varphi_1)\ \mathsf{U}_{I-(\tau-ts)}\ (\mathtt{TP} \wedge \varphi_2), v, +) \mid ts, v\}\\
\mathsf{RFO}^+(\varphi_1\ \mathsf{U}_I^{\mathbb{C}\mathrm{R}}\ \varphi_2) &= \mathsf{RFO}^-(\varphi_2) \cup \{\lambda\tau.\ (\mathtt{TP} \to \varphi_1)\ \mathsf{U}_{I-(\tau-ts)}\ (\mathtt{TP} \wedge \varphi_2), v, +) \mid ts, v\}\\
\mathsf{RFO}^-(\varphi_1\ \mathsf{U}_I\ \varphi_2) &= \mathsf{RFO}^-(\varphi_2) \cup \{\lambda\tau.\ (\mathtt{TP} \to \varphi_1)\ \mathsf{U}_{I-(\tau-ts)}\ (\mathtt{TP} \wedge \varphi_2), v, -) \mid ts, v\}\\
\mathsf{RFO}^-(\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi) &= \mathsf{RFO}^-(\exists v_1, \ldots, v_k.\ \varphi) \quad \text{where } \mathsf{fv}(\varphi) \setminus \overline{y} = \{v_1, \ldots, v_k\}
\end{aligned}
$$

The set $\mathsf{RE}(\varphi)$ of relevant event names comprises of all event names that occur in φ after unrolling let bindings.

D Benchmark formulae

D.1 GDPR

$$\begin{aligned}
\mathsf{consent} = {} & \Box(\forall data, dataid, dsid.\ \mathsf{use}(data, dataid, dsid) \\
& \longrightarrow (\blacklozenge\ \mathsf{legal_grounds}(dsid, data)) \\
& \quad \lor (\neg \mathsf{ds_revoke}(dsid, data)\ \mathsf{S}\ \mathsf{ds_consent}(dsid, data))) \\
\mathsf{deletion} = {} & \Box(\forall data, dataid, dsid.\ \mathsf{ds_deletion_request}(data, dataid, dsid) \\
& \longrightarrow \lozenge_{[0,30]}\ \mathsf{delete}(data, dataid, dsid)) \\
\mathsf{information} = {} & \Box(\forall data, dataid, dsid.\ \mathsf{collect}(data, dataid, dsid) \\
& \longrightarrow (\bigcirc\ \mathsf{inform}(dsid) \lor \blacklozenge\ \mathsf{inform}(dsid))) \\
\mathsf{lawfulness} = {} & \Box(\forall data, dataid, dsid.\ \mathsf{use}(data, dataid, dsid) \\
& \longrightarrow \blacklozenge(\mathsf{ds_consent}(dsid, data) \lor \mathsf{legal_grounds}(dsid, data))) \\
\mathsf{sharing} = {} & \Box(\forall data, dataid, dsid, processorid. \\
& (\mathsf{ds_deletion_request}(data, dataid, dsid) \\
& \quad \land \blacklozenge\ \mathsf{share_with}(processorid, dataid)) \\
& \longrightarrow \lozenge_{[0,30]}\ \mathsf{notify_proc}(processorid, dataid)) \\
\mathsf{gdpr} = {} & \mathsf{consent} \land \mathsf{delete} \land \mathsf{information} \land \mathsf{sharing}
\end{aligned}$$

D.2 GDPR$^{\text{FUN}}$

$$\begin{aligned}
\mathsf{consent} = {} & \Box(\forall data, dataid, dsid.\ \mathsf{use}(data, dataid, dsid) \\
& \longrightarrow (\blacklozenge\ \mathsf{legal_grounds}(dsid, data)) \\
& \quad \lor (\mathsf{eq}(\mathsf{owner}(data, dataid), dsid) \approx 1 \\
& \qquad \land \mathsf{has_consent}(dsid, data) \approx 1)) \\
\mathsf{management} = {} & \Box(\forall data, dsid. \\
& (\mathsf{ds_consent}(dsid, data) \\
& \longrightarrow \mathsf{call_function}(\texttt{"register_consent"}, \\
& \qquad \mathsf{register_consent}(dsid, data))) \\
& \land (\mathsf{ds_revoke}(dsid, data) \\
& \longrightarrow \mathsf{call_function}(\texttt{"revoke_consent"}, \\
& \qquad \mathsf{revoke_consent}(dsid, data))) \\
\mathsf{deletion} = {} & \Box(\forall data, dataid. \\
& \mathsf{ds_deletion_request}(data, dataid, \mathsf{owner}(data, dataid)) \\
& \longrightarrow \lozenge_{[0,30]}\ \mathsf{delete}(data, dataid, \mathsf{owner}(data, dataid))) \\
\mathsf{information} = {} & \Box(\forall data, dataid, dsid. \\
& \mathsf{collect}(data, dataid, dsid) \\
& \longrightarrow \mathsf{call_function}(\texttt{"register_owner"}, \\
& \quad \mathsf{register_owner}(data, dataid, dsid)) \\
& \quad \land (\bigcirc\ \mathsf{inform}(dsid) \lor \blacklozenge\ \mathsf{inform}(dsid))) \\
\mathsf{sharing} = {} & \Box(\forall data, dataid, processorid. \\
& (\mathsf{ds_deletion_request}(data, dataid, \mathsf{owner}(data, dataid)) \\
& \land \blacklozenge\ \mathsf{share_with}(processorid, dataid)) \\
& \longrightarrow \lozenge_{[0,30]}\ \mathsf{notify_proc}(processorid, dataid)) \\
\mathsf{gdpr} = {} & \mathsf{consent} \land \mathsf{management} \land \mathsf{deletion} \land \mathsf{information} \land \mathsf{sharing}
\end{aligned}$$

Python:

```python
owners = {}
consent = set()

def has_consent(dsid, data):
    return (dsid, data) in consent

def register_consent(dsid, data):
    consent.add((dsid, data))
    return 1

def revoke_consent(dsid, data):
    global consent
    if (dsid, data) in consent:
        consent.remove((dsid, data))
    return 1

def register_owner(data, dataid, dsid):
    owners[(data, dataid)] = dsid
    return 1

def owner(data, dataid):
    return owners.get((data, dataid), "None")
```

D.3 NOKIA

$$
\begin{aligned}
\mathsf{del\text{-}1\text{-}2} = {} & \Box(\forall user, data.\ \mathsf{delete}(user, \texttt{"db1"}, data) \land \mathsf{eq}(data, \texttt{"[unknown]"}) \approx 0 \\
& \quad \longrightarrow ((\blacklozenge_{[0,1s)} \lozenge_{[0,30h)} (\exists user2.\ \mathsf{delete}(user2, \texttt{"db2"}, data))) \\
& \qquad \lor ((\lozenge_{[0,1s)} \blacklozenge_{[0,30h)} (\exists user2.\ \mathsf{insert}(user2, \texttt{"db1"}, data))) \\
& \qquad\quad \land (\blacksquare_{[0,30h)} \Box_{[0,30h)} (\neg(\exists user2.\ \mathsf{delete}(user2, \texttt{"db3"}, data))))))) \\
\mathsf{del\text{-}2\text{-}3} = {} & \Box(\forall user, data.\ \mathsf{delete}(user, \texttt{"db2"}, data) \land \mathsf{eq}(data, \texttt{"[unknown]"}) \approx 0 \\
& \quad \longrightarrow \blacklozenge_{[0,1s)} \lozenge_{[0,60s]} (\exists user2.\ \mathsf{delete}(user2, \texttt{"db3"}, data))) \\
\mathsf{del\text{-}3\text{-}2} = {} & \Box(\forall user, data.\ \mathsf{delete}(user, \texttt{"db3"}, data) \land \mathsf{eq}(data, \texttt{"[unknown]"}) \approx 0 \\
& \quad \longrightarrow \blacklozenge_{[0,60s)} \lozenge_{[0,1s)} (\exists user2.\ \mathsf{delete}(user2, \texttt{"db2"}, data))) \\
\mathsf{delete} = {} & \Box(\forall user, data.\ \mathsf{delete}(user, \texttt{"db2"}, data) \longrightarrow user \approx \texttt{"script"}) \\
\mathsf{ins\text{-}1\text{-}2} = {} & \Box(\forall user, data.\ \mathsf{insert}(user, \texttt{"db1"}, data) \land \mathsf{eq}(data, \texttt{"[unknown]"}) \approx 0 \\
& \quad \longrightarrow \blacklozenge_{[0,1s)} \lozenge_{[0,30h]} (\exists user2.\ \mathsf{insert}(user2, \texttt{"db2"}, data) \\
& \qquad\qquad\qquad\qquad\qquad \lor \mathsf{delete}(user2, \texttt{"db1"}, data))) \\
\mathsf{ins\text{-}2\text{-}3} = {} & \Box(\forall user, data.\ \mathsf{insert}(user, \texttt{"db2"}, data) \land \mathsf{eq}(data, \texttt{"[unknown]"}) \approx 0 \\
& \quad \longrightarrow \blacklozenge_{[0,1s)} \lozenge_{[0,60s]} (\exists user2.\ \mathsf{insert}(user2, \texttt{"db3"}, data))) \\
\mathsf{ins\text{-}3\text{-}2} = {} & \Box(\forall user, data.\ \mathsf{insert}(user, \texttt{"db3"}, data) \land \mathsf{eq}(data, \texttt{"[unknown]"}) \\
& \quad \longrightarrow \blacklozenge_{[0,60s)} \lozenge_{[0,1s]} (\exists user2.\ \mathsf{insert}(user2, \texttt{"db2"}, data))) \\
\mathsf{insert} = {} & \Box(\forall user, data. \mathsf{insert}(user, \texttt{"db2"}, data) \longrightarrow user \approx \texttt{"script"}) \\
\mathsf{script1} = {} & \mathsf{let\ any\_operation}(script, db, data)\text{-} = \\
& \quad \mathsf{select}(script, db, data) \lor \mathsf{insert}(script, db, data) \\
& \quad \lor \mathsf{delete}(script, db, data) \lor \mathsf{update}(script, db, data)\ \mathsf{in} \\
& \mathsf{let\ running}(script) = \\
& \quad (\neg \blacklozenge_{[0,1s)} \lozenge_{[0,1s)} \mathsf{end}(script))\ \mathsf{S}\ (\blacklozenge_{[0,1s)} \lozenge_{[0,1s)} \mathsf{start}(script))\ \mathsf{in} \\
& \Box(\forall db, data.\ \mathsf{any\_operation}(\texttt{"script"}, db, data) \\
& \quad \longrightarrow (\mathsf{running}(\texttt{"script"}) \lor (\blacklozenge_{[0,1s)} \lozenge_{[0,1s)} \mathsf{end}(\texttt{"script"})))) \\
\mathsf{select} = {} & \Box(\forall user, data.\ \mathsf{select}(user, \texttt{"db2"}, data) \\
& \longrightarrow user \approx \texttt{"script"} \lor user \approx \texttt{"triggers"} \\
\mathsf{update} = {} & \Box(\forall user, data.\ \neg \mathsf{update}(user, \texttt{"db2"}, data))
\end{aligned}
$$

D.4 IC

$$
\begin{array}{l}
\mathsf{validation} = \mathsf{let\ node\_added\_to\_subnet}(node\_id, node\_addr, subnet) = \\
\quad\quad \mathsf{registry\_\_node\_added\_to\_subnet}(node\_id, node\_addr, subnet)\ \mathsf{in} \\
\quad \mathsf{let\ node\_removed\_from\_subnet}(node\_id, node\_addr) = \\
\quad\quad \mathsf{registry\_\_node\_removed\_from\_subnet}(node\_id, node\_addr)\ \mathsf{in} \\
\quad \mathsf{let\ in\_subnet}(node\_id, node\_addr, subnet) = \\
\quad\quad \blacklozenge_{[0s,\infty)}\ \mathsf{originally\_in\_subnet}(node\_id, node\_addr, subnet) \\
\quad\quad \land \neg(\blacklozenge_{[0s,\infty)}\ \mathsf{node\_removed\_from\_subnet}(node\_id, node\_addr)) \\
\quad\quad\quad \lor \neg\mathsf{node\_removed\_from\_subnet}(node\_id, node\_addr) \\
\quad\quad\quad\quad \mathsf{S}_{[0s,\infty)}\ \mathsf{node\_added\_to\_subnet}(node\_id, node\_addr, subnet)\ \mathsf{in} \\
\quad \mathsf{let\ subnet\_size}(subnet\_id, n) = \\
\quad\quad n \leftarrow \mathtt{CNT}(node\_id; subnet\_id) \\
\quad\quad\quad (\exists node\_addr.\ \mathsf{in\_subnet}(node\_id, node\_addr, subnet\_id))\ \mathsf{in} \\
\quad \mathsf{let\ block\_added}(node\_id, subnet\_id, block, t\_add) = \\
\quad\quad \mathsf{validated\_BlockProposal\_Added}(node\_id, subnet\_id, block) \\
\quad\quad \land (\exists node\_addr.\ \mathsf{in\_subnet}(node\_id, node\_addr, subnet\_id)) \\
\quad\quad \land \mathsf{ts}(t\_add)\ \mathsf{in} \\
\quad \mathsf{let\ validated}(block, subnet\_id, t\_add) = \\
\quad\quad \exists n\_validated.\ \exists n\_subnet.\ (\\
\quad\quad\quad n\_validated \leftarrow \mathtt{CNT}(valid\_node; block, subnet\_id, t\_add) \\
\quad\quad\quad\quad (\blacklozenge_{[0s,\infty)}\ \mathsf{block\_added}(valid\_node, subnet\_id, block, t\_add) \\
\quad\quad\quad\quad \lor (\exists add\_node.\ \exists node\_addr. \\
\quad\quad\quad\quad\quad \blacklozenge_{[0s,\infty)}\ \mathsf{block\_added}(add\_node, subnet\_id, block, t\_add) \\
\quad\quad\quad\quad\quad \land \mathsf{validated\_BlockProposal\_Moved}(valid\_node, subnet\_id, block) \\
\quad\quad\quad\quad\quad \land \mathsf{in\_subnet}(valid\_node, node\_addr, subnet\_id)))) \\
\quad\quad\quad \land \mathsf{subnet\_size}(subnet\_id, n\_subnet) \\
\quad\quad\quad \land (\mathsf{gt}(\mathsf{float\_of\_int}(n\_validated), \\
\quad\quad\quad\quad \mathsf{fdiv}(\mathsf{fmul}(2., \mathsf{float\_of\_int}(n\_subnet)), 3.)) \approx 1)\ \mathsf{in} \\
\quad \mathsf{let\ time\_per\_block}(block, subnet\_id, time) = \\
\quad\quad \exists t\_add.\ \exists t\_validated.\ (\\
\quad\quad\quad \mathsf{validated}(block, subnet\_id, t\_add) \\
\quad\quad\quad \land \neg(\CIRCLE_{[0s,\infty)}\ \blacklozenge_{[0s,\infty)}\ \mathsf{validated}(block, subnet\_id, t\_add)) \\
\quad\quad\quad \land \mathsf{ts}(t\_validated); \\
\quad\quad\quad time \leftarrow \mathsf{sub}(t\_validated, t\_add))\ \mathsf{in} \\
\quad \mathsf{let\ subnet\_type\_assoc}(subnet\_id, subnet\_type) = \\
\quad\quad \mathsf{original\_subnet\_type}(subnet\_id, subnet\_type) \\
\quad\quad \lor \mathsf{registry\_\_subnet\_created}(subnet\_id, subnet\_type) \\
\quad\quad \lor \mathsf{registry\_\_subnet\_updated}(subnet\_id, subnet\_type)\ \mathsf{in} \\
\quad \mathsf{let\ subnet\_type\_map}(subnet\_id, subnet\_type) = \\
\quad\quad \neg(\exists subnet\_type2.\ \mathsf{subnet\_type\_assoc}(subnet\_id, subnet\_type2)) \\
\quad\quad \mathsf{S}_{[0s,\infty)}\ \mathsf{subnet\_type\_assoc}(subnet\_id, subnet\_type)\ \mathsf{in} \\
\quad \forall block.\ \forall subnet\_id.\ \forall time. \\
\quad\quad \mathsf{time\_per\_block}(block, subnet\_id, time) \\
\quad\quad\quad \land (\mathsf{subnet\_type\_map}(subnet\_id, \texttt{"System"}) \land (\mathsf{gt}(time, 3000) \approx 1) \\
\quad\quad \lor (\mathsf{subnet\_type\_map}(subnet\_id, \texttt{"Application"}) \\
\quad\quad\quad\quad \lor \mathsf{subnet\_type\_map}(subnet\_id, \texttt{"VerifiedApplication"})) \\
\quad\quad\quad \land (\mathsf{gt}(time, 1000) \approx 1)) \\
\quad\quad \longrightarrow \mathsf{alert\_validation\_latency}(block, subnet\_id, time)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{clean\_logs} = \Box_{[0s,\infty)} (\forall node\_id.\ \forall node\_addr.\ \forall internal\_host\_id.\ \forall subnet\_id. \\
\qquad \forall component.\ \forall level.\ \forall message. \\
\quad (\text{let } \mathsf{in\_ic}(node\_id, node\_addr)\text{-} = \\
\qquad \blacklozenge_{[0s,\infty)}\ \mathsf{originally\_in\_ic}(node\_id, node\_addr) \\
\qquad\quad \land \neg(\blacklozenge_{[0s,\infty)}\ \mathsf{registry\_\_node\_removed\_from\_ic}(node\_id, node\_addr)) \\
\qquad \lor \neg \mathsf{registry\_\_node\_removed\_from\_ic}(node\_id, node\_addr) \\
\qquad\quad \mathsf{S}_{[0s,\infty)}\ \mathsf{registry\_\_node\_added\_to\_ic}(node\_id, node\_addr) \text{ in} \\
\quad \text{let } \mathsf{error\_level}(level) = level \approx \texttt{"CRITICAL"} \lor level \approx \texttt{"ERROR"} \text{ in} \\
\quad \neg(\mathsf{in\_ic}(node\_id, node\_addr) \\
\qquad \land \mathsf{log}(internal\_host\_id, node\_id, subnet\_id, component, level, message) \\
\qquad \land \mathsf{error\_level}(level))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{finalization} = \Box_{[0s,\infty)} (\forall node2.\ \forall hash2.\ \forall addr2.\ \forall subnet. \\
\qquad \forall height.\ \forall replica\_version. \\
\quad (\text{let } \mathsf{in\_ic}(node\_id, node\_addr) = \\
\qquad \blacklozenge_{[0s,\infty)}\ \mathsf{originally\_in\_ic}(node\_id, node\_addr) \\
\qquad\quad \land \neg(\blacklozenge_{[0s,\infty)}\ \mathsf{registry\_\_node\_removed\_from\_ic}(node\_id, node\_addr)) \\
\qquad \lor \neg \mathsf{registry\_\_node\_removed\_from\_ic}(node\_id, node\_addr) \\
\qquad\quad \mathsf{S}_{[0s,\infty)}\ \mathsf{registry\_\_node\_added\_to\_ic}(node\_id, node\_addr) \text{ in} \\
\quad \mathsf{finalized}(node2, subnet, height, hash2, replica\_version) \\
\qquad \land \mathsf{in\_ic}(node2, addr2) \\
\quad \longrightarrow \neg(\exists node1.\ \exists hash1.\ \exists addr1. \\
\qquad \blacklozenge_{[0s,\infty)}\ \mathsf{finalized}(node1, subnet, height, hash1, replica\_version) \\
\qquad\quad \land \mathsf{in\_ic}(node1, addr1) \land \neg(\mathsf{eq}(hash1, hash2) \approx 1))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{divergence} = \text{let } \mathsf{node\_added\_to\_subnet}(node\_id, node\_addr, subnet) = \\
\qquad \mathsf{registry\_\_node\_added\_to\_subnet}(node\_id, node\_addr, subnet) \text{ in} \\
\quad \text{let } \mathsf{node\_removed\_from\_subnet}(node\_id, node\_addr) = \\
\qquad \mathsf{registry\_\_node\_removed\_from\_subnet}(node\_id, node\_addr) \text{ in} \\
\quad \text{let } \mathsf{in\_subnet}(node\_id, node\_addr, subnet) = \\
\qquad \blacklozenge_{[0s,\infty)}\ \mathsf{originally\_in\_subnet}(node\_id, node\_addr, subnet) \\
\qquad\quad \land \neg(\blacklozenge_{[0s,\infty)}\ \mathsf{node\_removed\_from\_subnet}(node\_id, node\_addr)) \\
\qquad \lor \neg \mathsf{node\_removed\_from\_subnet}(node\_id, node\_addr) \\
\qquad\quad \mathsf{S}_{[0s,\infty)}\ \mathsf{node\_added\_to\_subnet}(node\_id, node\_addr, subnet) \text{ in} \\
\quad \forall node.\ \forall node\_addr.\ \forall subnet.\ \forall height. \\
\qquad \mathsf{end\_test}() \land \mathsf{in\_subnet}(node, node\_addr, subnet) \\
\qquad\quad \land \blacklozenge_{[0s,\infty)}\ \mathsf{replica\_diverged}(node, subnet, height) \\
\qquad \longrightarrow \mathsf{CUP\_share\_proposed}(node, subnet)
\end{array}
$$

$$
\begin{aligned}
\mathsf{height} = {} & \mathsf{let\ node\_added\_to\_subnet}(node\_id, node\_addr, subnet) = \\
& \quad \mathsf{registry\_\_node\_added\_to\_subnet}(node\_id, node\_addr, subnet)\ \mathsf{in} \\
& \mathsf{let\ node\_removed\_from\_subnet}(node\_id, node\_addr) = \\
& \quad \mathsf{registry\_\_node\_removed\_from\_subnet}(node\_id, node\_addr)\ \mathsf{in} \\
& \mathsf{let\ in\_subnet}(node\_id, node\_addr, subnet) = \\
& \quad \blacklozenge_{[0s,\infty)}\ \mathsf{originally\_in\_subnet}(node\_id, node\_addr, subnet) \\
& \qquad \wedge \neg(\blacklozenge_{[0s,\infty)}\ \mathsf{node\_removed\_from\_subnet}(node\_id, node\_addr)) \\
& \quad \vee \neg\mathsf{node\_removed\_from\_subnet}(node\_id, node\_addr) \\
& \qquad \mathsf{S}_{[0s,\infty)}\ \mathsf{node\_added\_to\_subnet}(node\_id, node\_addr, subnet)\ \mathsf{in} \\
& \mathsf{let\ subnet\_increasing}(subnet) = \\
& \quad \exists node1.\ \exists node2.\ \exists addr1.\ \exists addr2. \\
& \qquad \mathsf{in\_subnet}(node1, addr1, subnet) \wedge \mathsf{in\_subnet}(node2, addr2, subnet) \\
& \qquad \wedge (\mathsf{eq}(node1, node2) \approx 1) \\
& \qquad \wedge \neg(\neg\mathsf{p2p\_\_node\_removed}(node1, subnet, node2) \\
& \qquad\quad \mathsf{S}_{[0s,\infty)}\ \mathsf{p2p\_\_node\_added}(node1, subnet, node2))\ \mathsf{in} \\
& \mathsf{let\ subnet\_decreasing}(subnet) = \\
& \quad \exists node1.\ \exists addr1.\ \exists node2.\ \exists addr2.\ \exists subneta. \\
& \qquad \mathsf{in\_subnet}(node1, addr1, subnet) \\
& \qquad \wedge (\neg\mathsf{p2p\_\_node\_removed}(node1, subnet, node2) \\
& \qquad\quad \mathsf{S}_{[0s,\infty)}\ \mathsf{p2p\_\_node\_added}(node1, subnet, node2) \\
& \qquad\quad \vee \blacklozenge_{[0s,\infty)}\ \mathsf{originally\_in\_subnet}(node2, addr2, subnet) \\
& \qquad\qquad \wedge \neg(\blacklozenge_{[0s,\infty)}\ \mathsf{p2p\_\_node\_removed}(node1, subnet, node2)) \\
& \qquad\quad \wedge \neg(\exists subneta.\ \blacklozenge_{[0s,\infty)}\ \mathsf{p2p\_\_node\_added}(node1, subneta, node2))) \\
& \qquad \wedge \neg\mathsf{in\_subnet}(node2, subneta, subnet)\ \mathsf{in} \\
& \mathsf{let\ subnet\_is\_changing}(subnet) = \\
& \quad \mathsf{subnet\_increasing}(subnet) \vee \mathsf{subnet\_decreasing}(subnet)\ \mathsf{in} \\
& \mathsf{let\ fin}(node, subnet, height, hash, replica\_version)\text{-} = \\
& \quad \mathsf{finalized}(node, subnet, height, hash, replica\_version) \\
& \quad \wedge \neg(\bullet_{[0s,\infty)}\ \blacklozenge_{[0s,\infty)}(\\
& \qquad \exists nodea.\ \mathsf{finalized}(nodea, subnet, height, hash, replica\_version)))\ \mathsf{in} \\
& \forall subnet.\ \forall n1.\ \forall height1.\ \forall hash1.\ \forall replica\_version.\ \forall n2.\ \forall height2.\ \forall hash2. \\
& \quad \neg((\neg\mathsf{subnet\_is\_changing}(subnet) \\
& \qquad \mathsf{S}_{[81s,\infty)}\ \mathsf{fin}(n1, subnet, height1, hash1, replica\_version)) \\
& \quad \wedge \mathsf{fin}(n2, subnet, height2, hash2, replica\_version) \\
& \quad \wedge (\mathsf{eq}(height2, \mathsf{add}(height1, 1)) \approx 1))
\end{aligned}
$$

$$
\begin{array}{l}
\mathsf{logging} = \mathsf{let}\ \mathsf{node\_added\_to\_subnet}(node\_id, subnet) = \\
\quad \exists node\_addr.\ \mathsf{originally\_in\_subnet}(node\_id, node\_addr, subnet) \\
\qquad \lor\ \mathsf{registry\_\_node\_added\_to\_subnet}(node\_id, node\_addr, subnet)\ \mathsf{in} \\
\mathsf{let}\ \mathsf{node\_removed\_from\_subnet}(node\_id) = \\
\quad \exists node\_addr.\ \mathsf{registry\_\_node\_removed\_from\_subnet}(node\_id, node\_addr)\ \mathsf{in} \\
\mathsf{let}\ \mathsf{in\_subnet}(node\_id, subnet) = \\
\quad \neg \mathsf{node\_removed\_from\_subnet}(node\_id) \\
\qquad \mathsf{S}_{[0s,\infty)}\ \mathsf{node\_added\_to\_subnet}(node\_id, subnet)\ \mathsf{in} \\
\mathsf{let}\ \mathsf{is\_proper\_tp}() = \blacklozenge_{[1s,\infty)} \bot\ \mathsf{in} \\
\mathsf{let}\ \mathsf{relevant\_node}(node\_id, subnet) = \\
\quad \mathsf{in\_subnet}(node\_id, subnet)\ \mathsf{S}_{[10m+0s,\infty)}\ \mathsf{in\_subnet}(node\_id, subnet) \\
\quad \land\ \mathsf{is\_proper\_tp}()\ \mathsf{in} \\
\mathsf{let}\ \mathsf{relevant\_log}(node\_id, subnet, level, message, i) = \\
\quad \exists host\_id.\ \exists component. \\
\qquad \mathsf{log}(host\_id, node\_id, subnet, component, level, message) \\
\qquad \land\ (\mathsf{match}(component, \texttt{"orchestrator::ic\_execution\_environment::"}) \approx 1) \\
\qquad \land\ \neg(node\_id \approx \texttt{""}) \land \mathsf{tp}(i)\ \mathsf{in} \\
\mathsf{let}\ \mathsf{msg\_count}(node\_id, subnet, count) = \\
\quad count \leftarrow \texttt{SUM}(c; node\_id, subnet) \\
\qquad ((c \leftarrow \texttt{CNT}(i; node\_id, subnet) \\
\qquad\quad (\blacklozenge_{[0s,10m)}\ \mathsf{relevant\_log}(node\_id, subnet, level, message, i))) \\
\qquad \land\ \mathsf{relevant\_node}(node\_id, subnet) \\
\quad \lor\ \mathsf{relevant\_node}(node\_id, subnet) \land (c \approx 0))\ \mathsf{in} \\
\mathsf{let}\ \mathsf{typical\_behavior}(subnet, median) = \\
\quad (median \leftarrow \texttt{MED}(count; subnet)(\mathsf{msg\_count}(node\_id, subnet, count))) \\
\quad \land\ (\exists n.\ (n \leftarrow \texttt{CNT}(node\_id; subnet)(\mathsf{relevant\_node}(node\_id, subnet))) \\
\quad \land\ (\mathsf{geq}(n, 3) \approx 1))\ \mathsf{in} \\
\mathsf{let}\ \mathsf{typical\_behaviors}(subnet, median) = \\
\quad \blacklozenge_{[0s,10m)}\ \mathsf{typical\_behavior}(subnet, median)\ \mathsf{in} \\
\mathsf{let}\ \mathsf{compute}(subnet, node\_id, count, min, max) = \\
\quad \neg(\Diamond_{[0s,10m)}\ \mathsf{end\_test}()) \land \mathsf{msg\_count}(node\_id, subnet, count) \\
\quad \land\ (min \leftarrow \texttt{MIN}(m; subnet)(\mathsf{typical\_behaviors}(subnet, m))) \\
\quad \land\ (max \leftarrow \texttt{MAX}(m; subnet)(\mathsf{typical\_behaviors}(subnet, m)))\ \mathsf{in} \\
\mathsf{let}\ \mathsf{exceeds}(subnet, node\_id, count, min, max) = \\
\quad \mathsf{compute}(subnet, node\_id, count, min, max) \\
\quad \land\ (\mathsf{gt}(\mathsf{float\_of\_int}(count), \mathsf{fmul}(\mathsf{float\_of\_int}(max), 1.1)) \approx 1) \\
\qquad \lor\ \mathsf{compute}(subnet, node\_id, count, min, max) \\
\qquad\quad \land\ (\mathsf{lt}(\mathsf{float\_of\_int}(count), \mathsf{fmul}(\mathsf{float\_of\_int}(min), 0.9)) \approx 1)\ \mathsf{in} \\
\forall subnet.\ \forall node\_id.\ \forall count.\ \forall min.\ \forall max. \\
\quad \mathsf{exceeds}(subnet, node\_id, count, min, max) \\
\qquad \land\ \neg(\CIRCLE_{[0s,10m)}(\exists a.\ \exists b.\ \exists c.\ \mathsf{exceeds}(subnet, node\_id, a, b, c))) \\
\quad \longrightarrow \mathsf{alert\_continuous\_violations}(subnet, node\_id, count, min, max)
\end{array}
$$

$$
\begin{aligned}
\mathsf{reboot} = {} & \mathsf{let}\ \mathsf{in\_ic}(\mathit{node\_id}, \mathit{node\_addr}) = \\
& \quad \blacklozenge_{[0s,\infty)}\ \mathsf{originally\_in\_ic}(\mathit{node\_id}, \mathit{node\_addr}) \\
& \qquad \land \neg(\blacklozenge_{[0s,\infty)}\ \mathsf{registry\_\_node\_removed\_from\_ic}(\mathit{node\_id}, \mathit{node\_addr})) \\
& \quad \lor \neg \mathsf{registry\_\_node\_removed\_from\_ic}(\mathit{node\_id}, \mathit{node\_addr}) \\
& \qquad \mathsf{S}_{[0s,\infty)}\ \mathsf{registry\_\_node\_added\_to\_ic}(\mathit{node\_id}, \mathit{node\_addr})\ \mathsf{in} \\
& \mathsf{let}\ \mathsf{true\_reboot}(\mathit{ip\_addr}, \mathit{data\_center}) = \\
& \quad \exists \mathit{node\_id}.\ \mathsf{in\_ic}(\mathit{node\_id}, \mathit{ip\_addr}) \land \mathsf{reboot}(\mathit{ip\_addr}, \mathit{data\_center}) \\
& \qquad \land \CIRCLE_{[0s,\infty)}\ \blacklozenge_{[0s,\infty)}\ \mathsf{reboot}(\mathit{ip\_addr}, \mathit{data\_center})\ \mathsf{in} \\
& \mathsf{let}\ \mathsf{unintended\_reboot}(\mathit{ip\_addr}, \mathit{data\_center}) = \\
& \quad \mathsf{true\_reboot}(\mathit{ip\_addr}, \mathit{data\_center}) \\
& \quad \land \neg(\CIRCLE_{[0s,\infty)}(\neg \mathsf{reboot}(\mathit{ip\_addr}, \mathit{data\_center}) \\
& \qquad\qquad \mathsf{S}_{[0s,\infty)}\ \mathsf{reboot\_intent}(\mathit{ip\_addr}, \mathit{data\_center})))\ \mathsf{in} \\
& \mathsf{let}\ \mathsf{num\_reboots}(\mathit{data\_center}, n) = \\
& \quad \blacklozenge_{[0s,30m)}(\exists \mathit{ip\_addr}.\ \mathsf{unintended\_reboot}(\mathit{ip\_addr}, \mathit{data\_center})) \\
& \qquad \land (n \leftarrow \mathtt{CNT}(i; \mathit{data\_center}) \\
& \qquad\quad (\blacklozenge_{[0s,30m)}\ \mathsf{unintended\_reboot}(\mathit{ip\_addr}, \mathit{data\_center}) \land \mathsf{tp}(i)))\ \mathsf{in} \\
& \square_{[0s,\infty)}(\forall \mathit{data\_center}.\ \forall n.\ \mathsf{num\_reboots}(\mathit{data\_center}, n) \land (\mathsf{gt}(n, 2) \approx 1) \\
& \longrightarrow \mathsf{alert\_reboots}(\mathit{data\_center}, n))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{unauthorized} = {} & \mathsf{let}\ \mathsf{unauthorized\_connection\_attempt}(\mathit{local\_addr}, \mathit{peer\_addr}) = \\
& \quad \mathsf{ControlPlane\_\_spawn\_accept\_task\_\_tls\_server\_handshake\_failed}(\\
& \qquad \mathit{local\_addr}, \mathit{peer\_addr})\ \mathsf{in} \\
& \mathsf{let}\ \mathsf{node\_added\_to\_subnet}(\mathit{node\_id}, \mathit{node\_addr}, \mathit{subnet})\text{-} = \\
& \quad \mathsf{registry\_\_node\_added\_to\_subnet}(\mathit{node\_id}, \mathit{node\_addr}, \mathit{subnet})\ \mathsf{in} \\
& \mathsf{let}\ \mathsf{node\_removed\_from\_subnet}(\mathit{node\_id}, \mathit{node\_addr})\text{+} = \\
& \quad \mathsf{registry\_\_node\_removed\_from\_subnet}(\mathit{node\_id}, \mathit{node\_addr})\ \mathsf{in} \\
& \mathsf{let}\ \mathsf{in\_subnet}(\mathit{node\_id}, \mathit{node\_addr}, \mathit{subnet})\text{-} = \\
& \quad \blacklozenge_{[0s,\infty)}\ \mathsf{originally\_in\_subnet}(\mathit{node\_id}, \mathit{node\_addr}, \mathit{subnet}) \\
& \qquad \land \neg(\blacklozenge_{[0s,\infty)}\ \mathsf{node\_removed\_from\_subnet}(\mathit{node\_id}, \mathit{node\_addr})) \\
& \quad \lor \neg \mathsf{node\_removed\_from\_subnet}(\mathit{node\_id}, \mathit{node\_addr}) \\
& \qquad \mathsf{S}_{[0s,\infty)}\ \mathsf{node\_added\_to\_subnet}(\mathit{node\_id}, \mathit{node\_addr}, \mathit{subnet})\ \mathsf{in} \\
& \forall \mathit{dest\_addr}.\ \forall \mathit{sender\_addr}.\ \forall \mathit{dest\_id}.\ \forall \mathit{subnet}. \\
& \quad \mathsf{unauthorized\_connection\_attempt}(\mathit{dest\_addr}, \mathit{sender\_addr}) \\
& \quad \land \mathsf{in\_subnet}(\mathit{dest\_id}, \mathit{dest\_addr}, \mathit{subnet}) \\
& \quad \longrightarrow (\exists \mathit{sender\_id}.\ \exists \mathit{subneta}. \\
& \qquad \mathsf{in\_subnet}(\mathit{sender\_id}, \mathit{sender\_addr}, \mathit{subneta}) \\
& \qquad \land (\mathsf{eq}(\mathit{subneta}, \mathit{subnet}) \approx 1) \\
& \qquad \land \blacklozenge_{[0s,15m+0s]}\ \mathsf{in\_subnet}(\mathit{sender\_id}, \mathit{sender\_addr}, \mathit{subnet}))
\end{aligned}
$$

D.5 AGG

$$
\begin{aligned}
\mathsf{p1} = {} & \Box_{[0s,\infty)}(\forall u.\ \forall s.\ \forall a.\ \mathsf{withdraw}(u, a) \\
& \wedge (s \leftarrow \mathtt{SUM}(a; u)(\blacklozenge_{[0s,30s]}\ \mathsf{withdraw}(u, a) \wedge \mathsf{tp}(t))) \\
& \longrightarrow \mathsf{leq}(s, 10000.) \approx 1) \\
\mathsf{p2} = {} & \Box_{[0s,\infty)}(\forall u.\ \forall s.\ \forall a.\ \mathsf{withdraw}(u, a) \\
& \wedge (s \leftarrow \mathtt{SUM}(a; u)(\blacklozenge_{[0s,30s]}\ \mathsf{withdraw}(u, a) \wedge \mathsf{tp}(t))) \\
& \wedge (\neg \mathsf{limit\_off}(u)\ \mathsf{S}_{[0s,\infty)}\ \mathsf{limit\_on}(u)) \\
& \longrightarrow \mathsf{leq}(s, 10000.) \approx 1) \\
\mathsf{p3} = {} & \Box_{[0s,\infty)}(\forall u.\ \forall s.\ \forall a.\ \forall l.\ \mathsf{withdraw}(u, a) \\
& \wedge (s \leftarrow \mathtt{SUM}(a; u)(\blacklozenge_{[0s,30s]}\ \mathsf{withdraw}(u, a) \wedge \mathsf{tp}(t))) \\
& \wedge (\neg(\exists l2.\ \mathsf{limit}(u, l2))\ \mathsf{S}_{[0s,\infty)}\ \mathsf{limit}(u, l)) \\
& \longrightarrow \mathsf{leq}(s, l) \approx 1) \\
\mathsf{p4} = {} & \Box_{[0s,\infty)}(\forall u.\ \forall s.\ \forall m.\ \forall a.\ \mathsf{withdraw}(u, a) \\
& \wedge (s \leftarrow \mathtt{AVG}(a; u)(\blacklozenge_{[0s,90s]}\ \mathsf{withdraw}(u, a) \wedge \mathsf{tp}(t))) \\
& \wedge (m \leftarrow \mathtt{MAX}(a; u)(\blacklozenge_{[0s,7s]}\ \mathsf{withdraw}(u, a) \wedge \mathsf{tp}(t))) \\
& \longrightarrow \mathsf{leq}(m, \mathsf{fmul}(2., s)) \approx 1) \\
\mathsf{p5} = {} & \Box_{[0s,\infty)}(\forall s.\ \forall u.\ \forall a.\ \mathsf{withdraw}(u, a) \\
& \wedge (s \leftarrow \mathtt{AVG}(c; u)(c \leftarrow \mathtt{CNT}(t; u; \blacklozenge_{[0s,30s]}\ \mathsf{withdraw}(u, a) \wedge \mathsf{tp}(t)))) \\
& \longrightarrow \mathsf{leq}(s, 150) \approx 1) \\
\mathsf{p6} = {} & \Box_{[0s,\infty)}(\forall u.\ \forall c.\ \forall a.\ \mathsf{withdraw}(u, a) \\
& \wedge (c \leftarrow \mathtt{CNT}(k; u)((v \leftarrow \mathtt{AVG}(a; u)(\blacklozenge_{[0s,30s]}\ \mathsf{withdraw}(u, a) \wedge \mathsf{tp}(t))) \\
& \wedge \mathsf{withdraw}(u, p) \wedge \mathsf{tp}(k) \wedge (\mathsf{lt}(\mathsf{fmul}(2., v), p) \approx 1))) \\
& \longrightarrow \mathsf{leq}(c, 5) \approx 1)
\end{aligned}
$$

D.6 CLUSTER

$$
\begin{aligned}
\mathsf{dbscan} = {} & \mathsf{let}\ \mathsf{unintended\_reboot}(s, dc) = \\
& \quad \mathsf{reboot}(s, dc) \\
& \quad \wedge \neg(\bullet_{[0s,\infty)}(\neg \mathsf{reboot}(s, dc)\ \mathsf{S}_{[0s,\infty)}\ \mathsf{intended\_reboot}(s, dc)))\ \mathsf{in} \\
& \mathsf{let}\ \mathsf{cnt\_reboots}(dc, c) = \\
& \quad c \leftarrow \mathtt{CNT}(i; dc)(\blacklozenge_{[0s,1799s]}\ \mathsf{unintended\_reboot}(s, dc) \wedge \mathsf{tp}(i))\ \mathsf{in} \\
& \Box_{[0s,\infty)}(\forall dc.\ \forall l. \\
& \quad (dc, l \leftarrow \mathtt{DBSCAN}(dc, c;)(\mathsf{cnt\_reboots}(dc, c))) \wedge (l \approx 1) \\
& \quad \longrightarrow \mathsf{alert}(\mathsf{conc}(\mathsf{conc}(\texttt{"Data center "}, \mathsf{string\_of\_int}(dc)), \\
& \qquad \texttt{" has rebooted too often"}))) \\
\mathsf{grubbs} = {} & \mathsf{let}\ \mathsf{unintended\_reboot}(s, dc) = \\
& \quad \mathsf{reboot}(s, dc) \\
& \quad \wedge \neg(\bullet_{[0s,\infty)}(\neg \mathsf{reboot}(s, dc)\ \mathsf{S}_{[0s,\infty)}\ \mathsf{intended\_reboot}(s, dc)))\ \mathsf{in} \\
& \mathsf{let}\ \mathsf{cnt\_reboots}(dc, c) = \\
& \quad c \leftarrow \mathtt{CNT}(i; dc)(\blacklozenge_{[0s,1799s]}\ \mathsf{unintended\_reboot}(s, dc) \wedge \mathsf{tp}(i))\ \mathsf{in} \\
& \Box_{[0s,\infty)}(\forall dc.\ \forall l. \\
& \quad (dc, l \leftarrow \mathtt{GRUBBS}(dc, c;)(\mathsf{cnt\_reboots}(dc, c))) \wedge (l \approx 1) \\
& \quad \longrightarrow \mathsf{alert}(\mathsf{conc}(\mathsf{conc}(\texttt{"Data center "}, \mathsf{string\_of\_int}(dc)), \\
& \qquad \texttt{" has rebooted too often"})))
\end{aligned}
$$

Python:

```python
import numpy as np
from scipy import stats
from sklearn.cluster import DBSCAN as D

def GRUBBS(data):
    values = np.array([v for k, v in data])
    keys = [k for k, v in data]

    n = len(values)

    if n == 0:
        return []
    elif n == 1:
        return [(keys[0], 0)]

    mean = np.mean(values)
    std = np.std(values, ddof=1)

    G = np.abs(values - mean) / std

    t_crit = stats.t.ppf(1 - 0.05 / (2 * n), n - 2)
    G_crit = ((n - 1) / np.sqrt(n)) * \
        np.sqrt(t_crit**2 / (n - 2 + t_crit**2))

    is_outlier = G > G_crit

    result = [(k, int(outlier))
              for k, outlier in zip(keys, is_outlier)]

    return result

def DBSCAN(data):
    values = np.array([v for k, v in data])
    keys = [k for k, v in data]

    n = len(values)

    if n == 0:
        return []
    elif n == 1:
        return [(keys[0], 0)]

    X = values.reshape(-1, 1)

    dbscan = D(eps=0.5, min_samples=2)
    labels = dbscan.fit_predict(X)

    is_outlier = labels == -1

    result = [(k, int(outlier))
              for k, outlier in zip(keys, is_outlier)]

    return result
```