Scaling Up Proactive Enforcement

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, 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 [41] 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 [4], 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 [24].

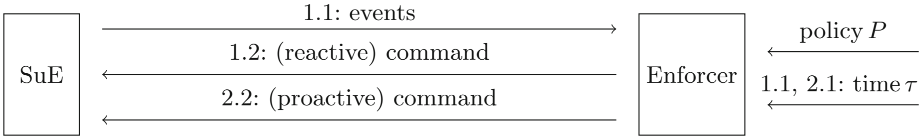


Fig. 1: Communication diagram for enforcement. R-step: 1.1, 1.2; P-step: 2.1, 2.2

R. Piskac and Z. Rakamarić (Eds.): CAV 2025, LNCS 15933, pp. 370–392, 2025.
https://doi.org/10.1007/978-3-031-98682-6_19

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 [18]:"

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

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 (^); *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 [4,23,24] 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 [24]. 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 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 [23,24], 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 our technical report [25]. EnfGuard is open source and is publicly available on GitHub [26].

Related Work. Reactive enforcement was introduced by Schneider et al. using security automata [41,13] that terminate the SuE to prevent violations. Subsequent research supported the suppression [9,17] and causation [31] of individual events by buffering SuE events before making decisions. This (unrealistic) buffering capability was later dropped [34], and other capabilities, such as delaying events [37,14] and SuE code inspection [38], were considered.

Many enforcers use (timed) automata either as a policy language [15,16] or as the translation target for logics such as MITL [36,40]. Controller synthesis tools for LTL [27,12,43], Timed CTL [10,35], and MTL [30,22] also generate enforcers.

Very few works enforce *first-order temporal* policies: Hallé and Villemaire [19] give an enforcer for LTL-FO$^+$, a first-order variant of future-only LTL. Hublet et al. [23] 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. [4] describe a proactive enforcer for finite automata and dynamic condition response graphs [21], which is a propositional formalism. Hublet et al. [24] 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 [24].

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     $\textbf{case } o \textbf{ of } \mathsf{PCom}(D_{\mathbb{C}}) \Rightarrow (\tau, D_{\mathbb{C}}) \cdot \mathsf{run}(s', \sigma \cdot (\tau, D_{\mathbb{C}}), \sigma', \tau + 1)$
4                $\mid \mathsf{NoCom} \Rightarrow \mathsf{run}(s', \sigma, \sigma', \tau + 1)$
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     $\textbf{case } o \textbf{ of } \mathsf{RCom}(D_{\mathbb{C}}, D_{\mathbb{S}}) \Rightarrow (\tau, D') \cdot \mathsf{run}(s', \sigma \cdot (\tau, D'), \sigma'', \tau + 1)$
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) [8,11] is an expressive logic for specifying trace properties. In this paper, we extend MFOTL with function applications in terms, aggregations [7], and non-recursive let bindings [44]. Our MFOTL syntax is defined by the following grammar (extensions highlighted):

$$\begin{aligned} t &::= c \mid x \mid \boxed{f(t,\ldots,t)} \\ \varphi &::= e(t,\ldots,t) \mid t \approx c \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists x.\ \varphi \mid \bigcirc_I \varphi \mid \bullet_I \varphi \mid \varphi\ \mathsf{U}_I\ \varphi \mid \varphi\ \mathsf{S}_I\ \varphi \\ &\quad \mid \boxed{x,\ldots,x \leftarrow \omega(t,\ldots,t;x,\ldots,x)\ \varphi} \mid \boxed{\mathsf{let}\, e(x,\ldots,x) = \varphi\ \mathsf{in}\ \varphi}\,. \end{aligned}$$

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 $\top := p \vee \neg p$, $\bot := \neg\top$, $\varphi \longrightarrow \psi := \neg\varphi \vee \psi$, 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.

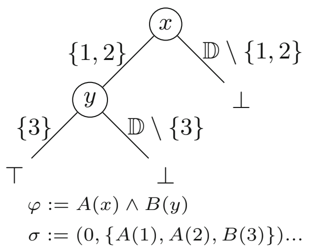
2.3 Partitioned decision trees

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 [33], Lima et al. [32] introduce partitioned decision trees (PDTs) to compactly represent sets of valuations. PDTs

$$
\begin{array}{lll}
v,i \vDash t \approx c & \text{iff } [\![\, t \,]\!]_v = c \quad \mid & v,i \vDash e(t_1,\dots,t_{a(e)}) \text{ iff } e([\![t_1]\!]_v,\dots,[\![t_{a(e)}]\!]_v) \in D_i \\
v,i \vDash \exists x.\ \varphi & \text{iff } v[x \mapsto d], i \vDash \varphi \text{ for some } d \in \mathbb{D} & \mid\ v,i \vDash \neg\varphi \quad \text{iff } v,i \nvDash \varphi \\
v,i \vDash \bigcirc_I \varphi & \text{iff } v,i+1 \vDash \varphi \text{ and } \tau_{i+1} - \tau_i \in I & \mid\ v,i \vDash \varphi \wedge \psi \text{ iff } v,i \vDash \varphi \text{ and } v,i \vDash \psi \\
v,i \vDash \CIRCLE_I \varphi & \text{iff } i > 0 \text{ and } v,i-1 \vDash \varphi \text{ and } \tau_i - \tau_{i-1} \in I & \\
v,i \vDash \varphi \,\mathsf{U}_I\, \psi & \multicolumn{2}{l}{\text{iff } v,j \vDash \psi \text{ for some } j \geq i \text{ with } \tau_j - \tau_i \in I \text{ and } v,k \vDash \varphi \text{ for all } i \leq k < j} \\
v,i \vDash \varphi \,\mathsf{S}_I\, \psi & \multicolumn{2}{l}{\text{iff } v,j \vDash \psi \text{ for some } j \leq i \text{ with } \tau_i - \tau_j \in I \text{ and } v,k \vDash \varphi \text{ for all } j < k \leq i}
\end{array}
$$

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


x
$\{1,2\}$
$\mathbb{D} \setminus \{1,2\}$
y
$\bot$
$\{3\}$
$\mathbb{D} \setminus \{3\}$
$\top$
$\bot$
$\varphi := A(x) \wedge B(y)$
$\sigma := (0, \{A(1), A(2), B(3)\})\dots$

(a) PDT of φ's satisfactions on σ

```
1 let find $d$ $parts$ = case $parts$ of
2       $[(D, pdt)] \Rightarrow (D, pdt)$
3     | $(D, pdt) :: parts \Rightarrow$
4         if $d \in D$ then $(D, pdt)$ else find $d$ $parts$
5 let specialize $pdt\, v$ = case $pdt$ of Leaf $\ell \Rightarrow \ell$
6     | Node $x\, parts \Rightarrow$
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 $\sigma := (0, \{A(1), A(2), B(3)\})\dots$ 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. [32] 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 [8], PDT-based algorithms lift many of the restrictions on the supported MFOTL fragment imposed in previous work [8,39], 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 [23], yet there are syntactic fragments that guarantee enforceability.

Hublet et al. [24, 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. $\mathbb{S}$).

Our technical report [25, Appendix A] gives the complete set of typing rules.

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

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

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

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

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

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 fp $(\sigma \cdot (\tau, D), X, f)$ =
    $(D_C, D_S) \leftarrow (\emptyset, \emptyset)$;      $r \leftarrow$ 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 [24, Algorithm 2]

The EMFOTL enforcement algorithm [24, 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 $\mathbb{N} \to$ EMFOTL, v a valuation, and $p \in \{+, -\}$. At every time-point i with timestamp ts, the algorithm enforces $\Phi = \bigwedge_{(\xi,v,+)} \xi(ts)[v] \wedge \bigwedge_{(\xi,v,-)} \neg\xi(ts)[v]$ 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 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 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 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 our technical report [25, 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 $\mathsf{eq}(x, y) := \textbf{if } x = y \textbf{ then } 1$ **else** 0 used to compare two variables, and some $f \in \mathbb{F}$, consider the formulae

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

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 $B(x) \land B(y) \land \neg(\mathsf{eq}(x, y) \approx 1)$ 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, $B(x) \land B(y) \land \neg(\mathsf{eq}(x, y) \approx 1)$ 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

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

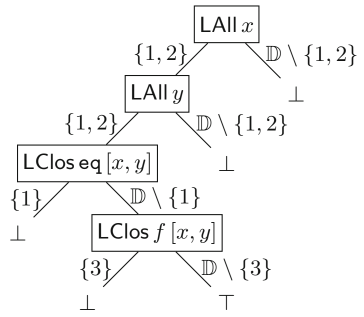
containing either free variables (LVar), existentially quantified variables (LEx), universally quantified variables (LAll), or closures with a function name and a list of terms (LClos). An example of an extended PDT is shown in Figure 6a.

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 our technical report [25, Appendix A], we extend Lima et al.'s [32] algorithm to use the new PDTs and show that it monitors all formulae covered by Definition 4.



$\varphi := \varphi_1$ from Example 3
$\sigma := (0, \{A(1), A(2), B(3)\})\ldots$

(a) PDT of φ's satisfactions on σ

```
let specialize $pdt$ $v$ = case $pdt$ of Leaf $\ell \Rightarrow \ell$
    | Node (LVar $x$) $parts$ $\Rightarrow$
        let (_, $pdt'$) = find $parts$ $(v\ x)$ in
        specialize $pdt'$ $v$
    | Node (LEx $x$) $parts$ $\Rightarrow$
      $\bigvee^{(D,pdt')\in parts}_{|D|<\infty} \bigvee_{d\in D}$ specialize $pdt'$ $v[x \mapsto d]$
      $\vee \bigvee^{(D,pdt')\in parts}_{|D|=\infty}$ specialize $pdt'$ $v$
    | Node (LAll $x$) $parts$ $\Rightarrow$
      $\bigwedge^{(D,pdt')\in parts}_{|D|<\infty} \bigwedge_{d\in D}$ specialize $pdt'$ $v[x \mapsto d]$
      $\wedge \bigwedge^{(D,pdt')\in parts}_{|D|=\infty}$ specialize $pdt'$ $v$
    | LClos $f$ $\bar{t}$ $\Rightarrow$ specialize (find $parts$ $[\![f(\bar{t})]\!]_v$) $v$
```

(b) Specialization of extended PDTs

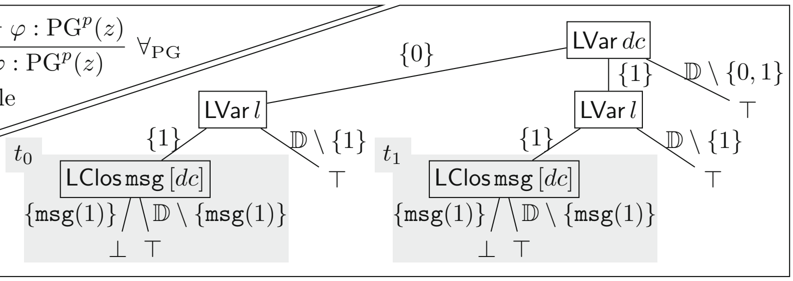
Fig. 6: Extended PDTs

Example 4. Consider the formula $\varphi_{\mathsf{Grubbs}}$ from Section 1. Let $\varphi'_{\mathsf{Grubbs}} := dc, l \leftarrow \mathtt{GRUBBS}(dc, c;)(\mathsf{cntReboots}(dc, c))) \wedge l \approx 1$ 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.

$$\begin{tikzcd}
 & \mathbb{O} \arrow[dl, dash, "\sqsupseteq"'] \arrow[dr, dash] & \\
\mathbb{C} \arrow[rr, dotted, leftrightarrow, "\neg"'] \arrow[dr, dash] & & \mathbb{S} \arrow[dl, dash] \\
 & \mathbb{CS} &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & \mathbb{O} \arrow[dll, dash, "\sqsupseteq"'] \arrow[dl, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
\mathbb{C}_s \arrow[rrrr, dotted, leftrightarrow, bend right, "\neg"] \arrow[dr, dash] & \mathbb{C}_n \arrow[rr, dotted, leftrightarrow, "\neg"] \arrow[d, dash] & & \mathbb{S}_n \arrow[d, dash] & \mathbb{S}_s \arrow[dl, dash] \\
 & \mathbb{C} \arrow[rr, dotted, leftrightarrow, "\neg"'] \arrow[dr, dash] & & \mathbb{S} \arrow[dl, dash] & \\
 & & \mathbb{CS} & &
\end{tikzcd}$$

Fig. 7: Hublet et al.'s type lattice [24] (left) and our extended type lattice (right)

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

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

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

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

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

Example 5. In $\varphi_{\mathsf{Grubbs}}$, the concatenation function (ˆ) 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 (ˆ) 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 $\Box(\forall x.\ \mathsf{alert}(x) \longrightarrow \mathsf{alert}(x \mathbin{\hat{}} x))$ cannot be typed to $\mathbb{C}$ by causing $\mathsf{alert}(x \mathbin{\hat{}} x)$: 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 (ˆ).

Enforcement. With the additional restrictions that we just introduced and our extended monitor, the enforcement algorithm proposed by Hublet et al. [24, Algorithm 2] can be reused when function applications are introduced. The modified termination and correctness proofs rely on Lemma 2 [25, 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:

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

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 [8] 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 $\overline{z}$.

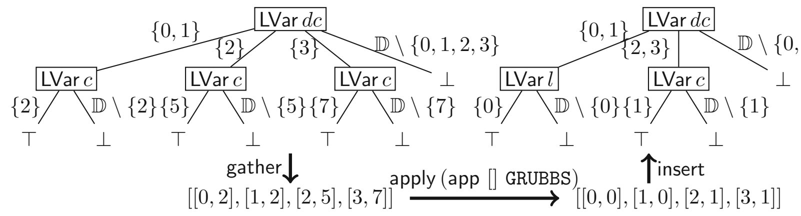
The algorithm proceeds in three steps, exemplified in Figure 10. First, the original PDT with Boolean leaves is transformed into a PDT with nodes in $\{\mathsf{LVar}\ y \mid y \in \overline{y}\}$ and leaves containing the multiset M. This is done using the gather function (l. 7–18) that uses standard concat : $\mathsf{list\,list}\,a \to \mathsf{list}\,a$ and 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 [25, 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* $\mathrm{SAT}_{\varphi}(\bullet, i, \sigma)$. *Then* aggregate $\overline{x}\ \overline{t}\ \overline{y}\ \overline{z}$ *pdt stores* $\mathrm{SAT}_{\overline{x} \leftarrow \omega(\overline{t}; \overline{y})\ \varphi}(\bullet, i, \sigma)$.

```
let distribute $f\, x\, (D, pdt)$ = if $|D| < \infty$ then $[(\{d\}, f\ d\ pdt) \mid d \in D]$ else $[(D, x)]$
let tabulate $\bar{t}\ sv\ vs$ = case $sv$ of $[\,] \Rightarrow [[\![\bar{t}]\!]_v \mid v \in vs]$
    | $(x, D) :: sv'$ where $x \in \mathbb{V} \Rightarrow$ tabulate $\bar{t}\ sv'\ [v[x \mapsto d] \mid d \in D, v \in vs]$
    | $(t, D) :: sv' \Rightarrow$ tabulate $\bar{t}\ sv'\ [v \mid v \in vs, [\![t]\!]_v \in D]$
let gather $sv\ \bar{t}\ \bar{y}\ pdt$ = let $f\, t\, (D, pdt) = (D,$ gather $(sv \cdot (t, D))\ t\ \bar{y}\ pdt)$ in
    case $pdt$ of Leaf $\ell \Rightarrow$ if $\ell = \top$ then Leaf (tabulate $\bar{t}\ sv\ [\emptyset]$) else Leaf $[\,]$
    | Node (LVar $x$) $parts \Rightarrow$ if $x \notin \bar{y}$ then applyn $(\cup)$ (map $(f\ x)$ $parts$) else
        let $g\ d\ pdt$ = gather $\{v[x \mapsto d] \mid v \in vs\}\ \bar{t}\ \bar{y}\ pdt$ in
        Node (LVar $v$) (concat (map (distribute $g$ $[\,]$) $parts$)
    | Node (LEx $x$) $parts \Rightarrow$ applyn $(\cup)$ (map $(f\ x)$ $parts$)
    | Node (LAll $x$) $parts \Rightarrow$ applyn $(\cap)$ (map $(f\ x)$ $parts$)
    | Node (LClos $h\,\bar{t}$ _) $parts \Rightarrow$ applyn $(\cup)$ (map $(h(\bar{t}))$ $parts$)
let agg $\bar{y}\ \omega\ M$ = if $|\bar{y}| > 0 \wedge M = [\,]$ then None else $\omega\, M$
let insert $v\ \bar{x}\ \bar{z}\ pdt$ = case $\bar{z}, pdt$ of
     $x :: \bar{z}'$, _ where $x \in \bar{x} \Rightarrow$ let $D$ = map $(\lambda v.\ v\ x)$ (all_leaves $pdt$) in
        if $D = [\,]$ then Leaf $\bot$
        else Node (LVar $y$, distribute $(\lambda d\ pdt.$ insert $v[x \mapsto d]\ \bar{x}\ \bar{z}'\ pdt)\ \bot\ (D, pdt)$)
    | $y :: \bar{z}'$, Node (LVar $y'$, $parts$) where $y = y' \Rightarrow$
        Node (LVar $y'$, map $(\lambda(D, pdt).\ (D,$ insert $x\ \bar{z}\ pdt)))$ $parts$
    | _ $:: \bar{z}'$, Node _ $\Rightarrow$ insert $v\ \bar{x}\ \bar{z}'\ pdt$
    | _, Leaf (Some $vs$) $\Rightarrow$ if $\exists v' \in vs.\ \forall x \in \mathrm{dom}\, v.\ v\ x = v'\ x$ then $\top$ else $\bot$
    | _, Leaf None $\Rightarrow \bot$
let aggregate $\omega\ \bar{x}\ \bar{t}\ \bar{y}\ \bar{z}\ pdt$ = insert $\emptyset\ \bar{x}\ \bar{z}$ (apply (agg $\bar{y}\ \omega$) (gather $[\,]\ \bar{t}\ \bar{y}\ pdt$))
```

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

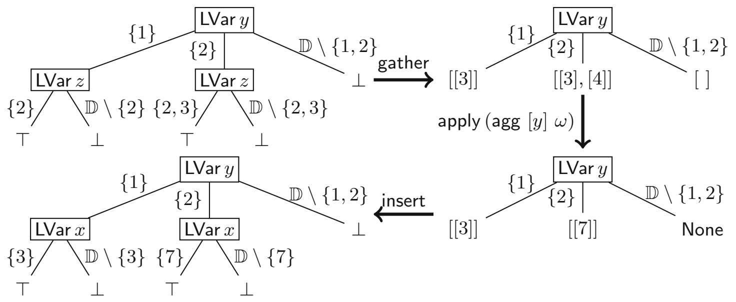


Fig. 10: Formula $x \leftarrow \texttt{SUM}(z+1; y)\ A(y,z)$ with $D = \{A(1,2), A(2,2), A(2,3)\}$

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

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

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

Fig. 11: Additional typing rules for aggregations

$\texttt{SUM}(y;)A(y) \longrightarrow A(x)$). Other past-guardedness rules have the same Γ on the LHS of all of their sequents. The rules in Figure 11 are sound [25, 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^{-}:

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

3.3 let bindings

We adopt the semantics of let bindings introduced by Zingg et al. [44]:

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

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 [25, 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.

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

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

Fig. 12: Additional typing rules for let bindings

Our report [25, Appendix B] gives the full typing of the formula in Section 1.

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 $c \leftarrow \mathtt{CNT}(i; dc)(\blacklozenge_{[0,1800)}(\mathsf{badReboot}(s, dc) \land \mathsf{tp}(i)))$ if badReboot is not in $\mathbb{C}$. Note that dc is past-guarded by reboot in $\mathsf{reboot}(s, dc) \land \neg \bullet(\neg \mathsf{reboot}(s, dc)\ \mathsf{S}\ \mathsf{intendReboot}(s, dc))$. 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 [25, 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 [25, 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 [3,24].
- 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 [6] 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 [42] and 3 platform execution logs [5] having 100–150 events/s.
- AGG: 6 fraud detection formulae [7] 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 (🐍). Our report [25, Appendix D] lists all formulae used.

In this evaluation, we compare EnfGuard to three tools: EnfPoly [23] and WhyEnf [24], the only existing MFOTL enforcement tools, and MonPoly [8],

	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	[3,24]	✓	1	5,631	10^{-4}	72				6	6	6	2	6
GPDR$^{\text{FUN}}$	[3,24]	✓	1	5,631	10^{-4}	108			🐍	6	6			
NOKIA	[28,6]	✓	1	9,458,824	109	44			✓	11	11	11	5	11
IC	[5]	✓	3	634,789	147	179	✓		✓	8	8			8
AGG	[7]		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 [7], let bindings [44], 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 [5]) and difficult to read and maintain. Aggregations strictly increase the policy language's expressiveness [20]: some requirements [5,7] 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_ℓ	max_ℓ	a	avg_{er}	avg_ℓ	max_ℓ	a	avg_{er}	avg_ℓ	max_ℓ	a	avg_{er}	avg_ℓ	max_ℓ
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_ℓ	max_ℓ	a	avg_{er}	avg_ℓ	max_ℓ
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_ℓ	max_ℓ
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.

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., Debois, S., Hildebrandt, T.: Proactive enforcement of provisions and obligations. J. Comput. Secur. **32**(3), 247â€"289 (2024)
5. 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)
6. Basin, D., Harvan, M., Klaedtke, F., Zalinescu, E.: Monitoring data usage in distributed systems. IEEE Transactions on Software Engineering **39**(10), 1403–1426 (2013)
7. 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)
8. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. Journal of the ACM (JACM) **62**(2), 1–45 (2015)
9. Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. In: Workshop on Foundations of Computer Security (FCS). Citeseer (2002)
10. 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)
11. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Transactions on Database Systems (TODS) **20**(2), 149–186 (1995)
12. 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)
13. 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)
14. 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)
15. 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)
16. 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)
17. 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)

18. Grubbs, F.E.: Sample criteria for testing outlying observations. Ann. Math. Statist. **21**(4), 27–58 (1950)
19. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. **5**(2), 192–206 (2012)
20. 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
21. 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)
22. 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)
23. 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)
24. 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)
25. Hublet, F., Lima, L., Basin, D., Krstić, S., Traytel, D.: Scaling-up proactive enforcement: Technical report (2025), https://doi.org/10.5281/zenodo.15501642
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 run-time security policies. International Journal of Information Security **4**, 2–16 (2005)
32. 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)
33. Minato, S.i.: Binary decision diagrams and applications for VLSI CAD, vol. 342. Springer Science & Business Media (1995)
34. 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)
35. 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)

36. 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)
37. 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)
38. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H.: Predictive runtime enforcement. Formal Methods Syst. Des. **51**(1), 154–199 (2017)
39. Raszyk, M., Basin, D., Krstić, S., Traytel, D.: Efficient evaluation of arbitrary relational calculus queries. Logical Methods in Computer Science **19** (2023)
40. 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)
41. Schneider, F.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000)
42. The DFINITY Team: The Internet Computer for geeks. Cryptology ePrint Archive, Paper 2022/087 (2022), https://eprint.iacr.org/2022/087
43. 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)
44. 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)