





Correct and Efficient Policy Monitoring, a Retrospective

David Basin¹, Srđan Krstić¹, Joshua Schneider¹, and Dmitriy Traytel²

¹ Institute of Information Security, Department of Computer Science, ETH Zurich
{basin, srdan.krstic, joshua.schneider}@inf.ethz.ch

² Department of Computer Science, University of Copenhagen
traytel@di.ku.dk

Abstract. The MonPoly project started over a decade ago to build effective tools for monitoring trace properties, including functional correctness, security, and compliance policies. The original MonPoly tool supported monitoring specifications given in metric first-order temporal logic, an expressive specification language. It handled both the online case, where system events are monitored as they occur, and the offline case, monitoring logs. Our tool has evolved over time into a family of tools and supporting infrastructure to make monitoring both scalable and suitable for high assurance applications. We survey this evolution which includes: (1) developing more expressive monitors, e.g., adding aggregation operators, regular expressions, and limited forms of recursion; (2) delimiting efficiently monitorable fragments and designing new monitoring algorithms for them; (3) supporting parallel and distributed monitoring; (4) using theorem proving to verify monitoring algorithms and explore extensions; and (5) carrying out ambitious case studies.

Keywords: runtime verification, monitoring, temporal logic

1 Introduction

Monitoring is a Formal Method for system analysis where one analyzes a system’s behavior as system events occur, or afterwards when reading the events from logs. The objective is to decide whether the system’s observed behavior satisfies a given specification and, if not, to report violations. This problem is general and has wide ranging applications. The events can be at any level of abstraction (machine instructions, operating system calls, I/O events, etc.) and one can apply monitoring to hardware, operating systems, software programs and components, network traffic, etc. Moreover, depending on the problem domain, the specification may state ordering requirements on the events, real-time requirements on when they occur, or requirements on the relationships between data referenced by the events. The challenge then is to design monitors that are general enough to handle many relevant problem domains and to make their decisions efficiently and effectively, even in the presence of high-velocity event streams.

We have been working for over a decade on different aspects of this problem. Parts of our research were project driven, tackling challenges that arose in applying monitoring to different problem domains and making our monitoring tools scale. Other parts were curiosity driven, exploring different monitoring semantics, algorithms, specification languages, parallelization techniques, and even formal

Tool	Logic	Features	References	Sect.
MonPoly	MFOTL ^{RANF} _{Ω, def}	online	[20, 26, 28, 31]	4.1
CppMon	MFOTL ^{RANF} _{Ω, def}	online	[56]	4.1
StaticMon	MFOTL ^{RANF} _{$\Omega, \text{def}, \text{rec}$}	online, pre-compiled	[57, 58]	4.1
HashMon	MFOTL ^{RANF} _{Ω, def}	online, randomized	[91, 92]	4.1
VeriMon	MFODL ^{RANF} _{$\Omega, \text{def}, \text{rec}$}	online, verified	[16, 17, 20, 97]	4.1, 6
MFOTL2RANF	MFOTL \rightarrow MFOTL ^{RANF}	translation	[82, 85]	4.2
MonPoly-Reg	MFOTL	online	[26, 28]	4.3
Aerial	MDL	equivalence verdicts	[12, 35, 69]	4.4
Hydra	MDL	multi-head	[81, 84, 87]	4.4
Vydra	MDL	multi-head, verified	[81, 87]	4.4, 6
Slicing framework	MFOTL ^{RANF}	offline, parallel	[15]	5.1
Slicing framework	MFOTL ^{RANF} _{Ω, def} , QTL [62]	online, parallel	[43, 94, 96]	5.2
POLMON	MTL ^{\downarrow}	unordered input	[32, 34]	5.3
TimelyMon	MFOTL ^{RANF}	unordered input	[88]	5.3

Fig. 1. Our monitors and related tools

verification applied to monitoring itself. We provide here a retrospective on this work, explaining the tools we have built, summarized in Figure 1, and highlighting their advances within the context of the larger field of runtime verification. We hope this will be of value both for those researchers interested in understanding our tools and the problems they address and those wishing to understand some of the challenges in bridging theory and practice in this exciting research area.

Our aim has been to develop foundations and tools to cover the largest possible range of applications. Our starting point has been the expressive specification language of metric first-order temporal logic (MFOTL) built on first-order logic with equality and metric temporal logic (MTL) operators. For some applications MFOTL is still not expressive enough. Hence we have systematically explored *extensions* of MFOTL (Section 2), such as adding aggregation operators ($-\Omega$ in Figure 1), regular expressions from dynamic logic ($-\text{DL}$), and limited forms of recursion ($-\text{rec}$). Unfortunately, monitoring using expressive specification languages is computationally intractable in the standard monitoring setting (Section 3). We have therefore explored ways to mitigate this problem by: delimiting efficiently monitorable *fragments* of MFOTL (Section 4), such as those monitorable using relational data structures ($-\text{RANF}$), and designing monitoring algorithms for them; weakening some of the requirements on monitors, such as how they present their output; and providing support for parallel and distributed monitoring (Section 5).

As monitoring is often used in critical applications where correctness matters, it is important that monitors themselves are correct. Part of our journey has been in using theorem provers to formally verify our monitoring algorithms (Section 6). The verified monitors can be run directly, with some performance slowdown compared to their optimized but unverified brethren. Alternatively they can be used to ascertain the correctness of other monitors using differential testing.

We describe our results here as well as substantial case studies that we carried out to learn where bottlenecks and limitations are in practice (Section 7).

$\sigma, v, i \models \mathbf{p}(\bar{t})$	if $\mathbf{p}(v(\bar{t})) \in D_i^\sigma$	$\sigma, v, i \models \mathbf{tpts}(t_1, t_2)$	if $v(t_1) = i$ and $v(t_2) = \tau_i^\sigma$
$\sigma, v, i \models t_1 \mathcal{R} t_2$	if $v(t_1) \mathcal{R} v(t_2)$	$\sigma, v, i \models \exists x. \alpha$	if $\exists d \in \mathbb{D}. \sigma, v[x \mapsto d], i \models \alpha$
$\sigma, v, i \models \neg \alpha$	if $\sigma, v, i \not\models \alpha$	$\sigma, v, i \models \Downarrow x. \alpha$	if $\sigma, v[x \mapsto R_i^\sigma(r)], i \models \alpha$
$\sigma, v, i \models \alpha \vee \beta$	if $\sigma, v, i \models \alpha$ or $\sigma, v, i \models \beta$		
$\sigma, v, i \models \ominus_I \alpha$	if $i > 0$, $\tau_i^\sigma - \tau_{i-1}^\sigma \in I$, and $\sigma, v, i-1 \models \alpha$		
$\sigma, v, i \models \circ_I \alpha$	if $\tau_{i+1}^\sigma - \tau_i^\sigma \in I$ and $\sigma, v, i+1 \models \alpha$		
$\sigma, v, i \models \alpha \mathbf{S}_I \beta$	if $\exists j \leq i. \sigma, v, j \models \beta$, $\tau_i^\sigma - \tau_j^\sigma \in I$ and $\forall k. \text{if } j < k \leq i \text{ then } \sigma, v, k \models \alpha$		
$\sigma, v, i \models \alpha \mathbf{U}_I \beta$	if $\exists j \geq i. \sigma, v, j \models \beta$, $\tau_j^\sigma - \tau_i^\sigma \in I$ and $\forall k. \text{if } i \leq k < j \text{ then } \sigma, v, k \models \alpha$		
$\sigma, v, i \models \triangleleft_I r$	if $\exists j \leq i. (j, i) \in (r)_v^\sigma$ and $\tau_i^\sigma - \tau_j^\sigma \in I$		
$\sigma, v, i \models \triangleright_I r$	if $\exists j \geq i. (i, j) \in (r)_v^\sigma$ and $\tau_j^\sigma - \tau_i^\sigma \in I$		
$\sigma, v, i \models y \leftarrow \Omega(t; \bar{g}) \alpha$	if $v(y) = \Omega(M)$ and if $M = \{\!\! \ \bar{g} \!\!\}$ then $ \bar{g} = 0$, where $M = \biguplus_{\bar{a} \in \mathbb{D}^{ \bar{g} }} \{\!\! u(t) \mid \sigma, u, i \models \alpha \text{ where } u = v[\bar{z} \mapsto \bar{a}] \!\!\}$ and $\bar{z} = \mathcal{V}(\alpha) \setminus \bar{g}$		
$\sigma, v, i \models \mathbf{def} \mathbf{p}(\bar{x}) := \alpha \text{ in } \beta$	if $\sigma[\mathbf{p} \Rightarrow \lambda j. \llbracket \alpha; \bar{x} \rrbracket_j^\sigma], v, i \models \beta$		
$\sigma, v, i \models \mathbf{rec} \mathbf{p}(\bar{x}) := \alpha \text{ in } \beta$	if $\sigma[\mathbf{p} \Rightarrow \lambda j. \mathbf{fp}_j(\lambda S k. \llbracket \alpha; \bar{x} \rrbracket_k^{\sigma[\mathbf{p} \Rightarrow S]})], v, i \models \beta$		
$(\star)_v^\sigma$	$= \{(i, i+1) \mid i \in \mathbb{N}\}$	$(r+s)_v^\sigma$	$= (r)_v^\sigma \cup (s)_v^\sigma$
$(\alpha?)_v^\sigma$	$= \{(i, i) \mid \sigma, v, i \models \alpha\}$	$(r^*)_v^\sigma$	$= ((r)_v^\sigma)^*$
$\llbracket \alpha; \bar{x} \rrbracket_i^\sigma$	$= \{v(\bar{x}) \mid \sigma, v, i \models \alpha\}$	$(rs)_v^\sigma$	$= \{(i, k) \mid \exists j. (i, j) \in (r)_v^\sigma \text{ and } (j, k) \in (s)_v^\sigma\}$
$\mathbf{fp}_i(F)$	$= F(\lambda j. \text{if } j < i \text{ then } \mathbf{fp}_j(F) \text{ else } \{\}) i$		

Fig. 2. Semantics of MFOTL (gray background) and its extensions

2 The Logic

We present a logic that unifies our tools' specification languages. Presently, no tool supports all presented features, but all features are supported by some tool (see Figure 1). We start with metric first-order temporal logic (MFOTL) [28, 47] and extend it with regular expressions [17, 36], aggregations operators [24], a freeze quantifier [32], and a recursion operator [102]. We refer to the cited publications for detailed explanations and the historical background of each operator.

We fix a set of event names \mathbb{E} and for simplicity assume a single infinite domain of values \mathbb{D} . We consider \mathbb{D} to include integers, strings, and floats, as well as POSIX regular expressions to match strings against (not to be confused with the temporal regular expressions occurring in formulas). The event names $\mathbf{p} \in \mathbb{E}$ have associated arities $\iota(\mathbf{p}) \in \mathbb{N}$. An *event* $\mathbf{p}(d_1, \dots, d_{\iota(\mathbf{p})})$ is an element of $\mathbb{E} \times \mathbb{D}^*$. We further fix infinite sets of variables \mathbb{V} and registers \mathbb{R} such that \mathbb{V} , \mathbb{R} , \mathbb{D} , and \mathbb{E} are all pairwise disjoint. Let \mathbb{I} be the set of nonempty intervals $[a, b) := \{x \in \mathbb{N} \mid a \leq x < b\}$, where $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$, and $a < b$. Terms \mathbb{T} include $\mathbb{V} \cup \mathbb{D}$ and can also be constructed by applying operators defined on \mathbb{D} (e.g., $+$ and \times on integers) to terms. Formulas φ and temporal regular expressions r are defined (mutually) inductively, where t, \mathbf{p}, r, x , and I range over $\mathbb{T}, \mathbb{E}, \mathbb{R}, \mathbb{V}$, and \mathbb{I} , respectively:

$$\begin{aligned} \varphi ::= & \mathbf{p}(\bar{t}) \mid \mathbf{tpts}(t, t) \mid t \mathcal{R} t \mid \neg \varphi \mid \varphi \vee \varphi \mid \exists x. \varphi \mid \ominus_I \varphi \mid \circ_I \varphi \mid \varphi \mathbf{S}_I \varphi \mid \varphi \mathbf{U}_I \varphi \mid \\ & \Downarrow x. \varphi \mid x \leftarrow \Omega(t; \bar{x}) \varphi \mid \mathbf{def} \mathbf{p}(\bar{x}) := \varphi \text{ in } \varphi \mid \mathbf{rec} \mathbf{p}(\bar{x}) := \varphi \text{ in } \varphi \mid \triangleleft_I r \mid \triangleright_I r \\ r ::= & \star \mid \varphi? \mid rr \mid r + r \mid r^* . \end{aligned}$$

Here $\mathcal{R} \in \{=, <, \leq, \stackrel{\mathbb{R}\mathbb{E}}{\Leftarrow}\}$ is a rigid (i.e., non-changing) relation and $\Omega \in \{\text{CNT}, \text{SUM}, \text{AVG}, \text{MIN}, \text{MAX}, \text{MED}\}$ is an aggregation function on multisets, e.g., $\text{CNT} \llbracket 1, 1, 2 \rrbracket = 3$ and $\text{SUM} \llbracket 1, 1, 2 \rrbracket = 4$. We write \bar{a} for a list of zero or more a .

MFOTL formulas are built from operators shown in gray background. Formulas $\mathfrak{p}(\bar{t})$ are called (*atomic*) *predicates*. The special predicate \mathfrak{tpts} refers to the current time(-point and time-stamp). Besides logical operators (\neg, \vee, \exists) and rigid relations (\mathcal{R}), MFOTL has metric past and future temporal operators \ominus (previous), \circ (next), S (since), and U (until), which may be nested freely. Metric temporal logic (MTL) is a fragment of MFOTL with nullary predicates and no quantification.

The addition of \triangleleft (past match) and \triangleright (future match) operators to MTL and MFOTL results in their dynamic variants MDL and MFODL, respectively. These operators use temporal regular expressions constructed from wildcard (\star), test ($?$), concatenation, alternation ($+$), and star ($*$) operations. We also consider formulas with freeze quantification $\downarrow x. \alpha$. In particular, MTL^\downarrow is the extension of MTL with freeze quantifiers. Finally, MFOTL is extended with aggregations $x \leftarrow \Omega(t; \bar{x}) \alpha$ (called MFOTL_Ω), and with non-recursive ($\text{MFOTL}_{\text{def}}$) and recursive ($\text{MFOTL}_{\text{rec}}$) definitions. The latter two are given by formulas of the form $\text{def } \mathfrak{p}(\bar{x}) := \alpha \text{ in } \beta$ and $\text{rec } \mathfrak{p}(\bar{x}) := \alpha \text{ in } \beta$, respectively. We derive additional operators: truth $\top := 0 = 0$, falsehood $\perp := \neg \top$, inequality $t_1 \neq t_2 := \neg(t_1 = t_2)$, conjunction $\alpha \wedge \beta := \neg(\neg\alpha \vee \neg\beta)$, implication $\alpha \rightarrow \beta := \neg\alpha \vee \beta$, current time-point $\mathfrak{tp}(i) := \exists t. \mathfrak{tpts}(i, t)$ and time-stamp $\mathfrak{ts}(t) := \exists i. \mathfrak{tpts}(i, t)$, universal quantification $\forall x. \alpha := \neg(\exists x. \neg\alpha)$, eventually $\diamond_I \alpha := \top \mathsf{U}_I \alpha$, always $\square_I \alpha := \neg \diamond_I \neg \alpha$, once $\diamond_I \alpha := \top \mathsf{S}_I \alpha$, and historically $\square_I \alpha := \neg \diamond_I \neg \alpha$. A formula is *future-bounded* iff all subformulas of the form $\alpha \mathsf{U}_{[a,b]} \beta$ and $\triangleright_{[a,b]} r$ (including derived operators) satisfy $b < \infty$.

Formulas are interpreted over *temporal structures*, which model executions of a monitored system. A temporal structure σ is an infinite sequence $(\tau_i^\sigma, D_i^\sigma, R_i^\sigma)_{i \in \mathbb{N}}$, where $\tau_i^\sigma \in \mathbb{N}$ is a time-stamp, the *database* $D_i^\sigma \in \mathcal{P}(\mathbb{E} \times \mathbb{D}^*)$ is a finite set of events, and the *register map* R_i^σ assigns each register $r \in \mathbb{R}$ a single domain value from \mathbb{D} . Time-stamps must be *monotone* ($\forall i. \tau_i \leq \tau_{i+1}$) and *progressing* ($\forall \tau. \exists i. \tau < \tau_i$). Note that different time-points $i \neq j$ may have the same time-stamp $\tau_i = \tau_j$.

Figure 2 shows the relation $\sigma, v, i \models \varphi$ defining the satisfaction of the formula φ for a temporal structure σ , a valuation v , and a time-point i . The valuation v assigns domain values to φ 's free variables $\mathcal{V}(\varphi)$. Overloading notation, v is also the extension of v to terms \mathbb{T} in the obvious way, e.g., $v(t_1 + t_2) = v(t_1) + v(t_2)$. The valuation $v[x \mapsto d]$ is equal to v except that d is assigned to the variable x . Similarly, trace $\sigma[p \Rightarrow X]$ is equal to σ except that the set of events for predicate \mathfrak{p} from D_i^σ is replaced by $X(i)$ at each time-point i . The rigid relation $x \stackrel{\text{RE}}{\Leftarrow} r$ is satisfied by all strings x matched by the POSIX regular expression r . The other rigid relations behave as expected. Aggregations support grouping using variables \bar{g} and their semantics is defined using multiset union \uplus . The additional operators are intuitive, e.g., unfolding a non-recursive definition (even under temporal operators) results in an equivalent formula. The semantics of recursive definitions is as expected provided all recursive occurrences of \mathfrak{p} in φ are evaluated at past time-points.

For $I \in \mathbb{I}$ and $n \in \mathbb{N}$, let $I - n$ denote the interval $\{x - n \mid x \in I\} \cap \mathbb{N}$ and I^- the set of intervals $\{I - m \mid m \in \mathbb{N}\} \setminus \{\emptyset\}$, which is always finite. We write $\text{SF}(\varphi)$ for the set of φ 's subformulas and define *interval-skewed subformulas* $\text{ISF}(\varphi)$ as

$$\begin{aligned} \text{SF}(\varphi) \cup \{ \alpha \mathsf{S}_J \beta \mid \alpha \mathsf{S}_I \beta \in \text{SF}(\varphi), J \in I^- \} \cup \{ \triangleleft_J r \mid \triangleleft_I r \in \text{SF}(\varphi), J \in I^- \} \\ \cup \{ \alpha \mathsf{U}_J \beta \mid \alpha \mathsf{U}_I \beta \in \text{SF}(\varphi), J \in I^- \} \cup \{ \triangleright_J r \mid \triangleright_I r \in \text{SF}(\varphi), J \in I^- \}. \end{aligned}$$

3 Monitoring Setting

The central problem in monitoring is, given a policy and a trace from a monitored system, to decide whether the trace satisfies the policy. The monitoring problem has many variants that motivate specialized algorithms. For example, one may grant the monitor random access to the trace for efficiency, or require the timely detection of violations for some applications. Here we sketch the most important problem dimensions as well as the setting in which we position our tools. A more detailed taxonomy for runtime verification tools has been developed by Falcone et al. [52] and extensive introductions to the topic by many others [11, 72, 90].

Offline monitors run after the monitored system has terminated and therefore read the complete trace, typically stored as a log file. In contrast, *online* monitors run while the monitored system executes and observe a trace’s prefix up to the present. They typically receive the trace incrementally, as a stream of events, one event at a time. Equivalently, online monitors read the trace with a single one-way reading head that moves forward only, whereas offline monitors have random access to the entire trace. Every online monitor can be used offline by replaying the log file as a stream, but it may be less efficient than a dedicated offline tool. We primarily develop online monitors, yet we propose a multi-head approach that lies in between offline and online monitoring (Section 4.4).

The linear order of events observed by a monitor (be it in a log file or a stream) does not necessarily coincide with the events’ temporal order of occurrence in the monitored system. We speak of a *trace* only when they do coincide; otherwise, we call the monitor’s input *observations*. For example, most distributed systems do not provide traces in this strict sense because it is difficult to reconstruct the true order of events [98]. Our monitors operate on traces by default. We discuss two approaches that handle more general observations in Section 5.3.

All our approaches work with policies of the form $\Box \varphi$ where φ is future-bounded. Such policies describe *safety properties*,³ characterized by *bad prefixes* [5], which are finite traces with all their infinite extensions violating the policy. Our monitors detect all bad prefixes of $\Box \varphi$ by *evaluating* the formula $\neg \varphi$ at every time-point. Their output is monotonic with respect to time-points and consists of exactly those time-points at which $\neg \varphi$ is satisfied in all infinite extensions. A non-empty output indicates that $\Box \varphi$ is violated. Dually, *co-safety* properties [60] of the form $\Diamond \varphi'$ can be monitored by evaluating φ' directly.

A monitor’s output may range from a single bit to detailed proof trees [73]. As described above, to monitor a policy $\Box \forall \bar{x}. \varphi$, our monitors evaluate $\neg \forall \bar{x}. \varphi$ at every time-point. After pushing the negation in and dropping the leading existential quantifiers, they can evaluate $\neg \varphi$, which has free variables. The computed valuations are output together with the corresponding time-points and provide insight into the policy’s violations. The output is never provided for time-points beyond the observed trace prefix. This cannot be avoided in general: a policy $\Box \varphi$ is violated on all traces (and therefore also on all extensions of the empty prefix) iff φ is unsatisfiable, which is undecidable for MFOTL [22].

³ Although not every safety property expressible in MFOTL has this form [47].

4 Restrictions and Algorithms

We describe our algorithms for monitoring fragments of our logic. Restricting the policy language has two advantages. First, without restrictions it may be impossible to build a monitor that satisfies the desired properties. For example, detecting bad prefixes is already undecidable for a much weaker form of quantification than that of MFOTL [37]. Second, algorithms can be tailored to language fragments yielding better performance in exchange for less expressiveness or conciseness.

We focus primarily on fragments that retain MFOTL’s first-order aspects and which can be monitored using finite relations (Section 4.1). A monitor-independent translation makes these fragments more user-friendly by lifting syntactic restrictions (Section 4.2). We also compare the finite relation approach to automatic structures (Section 4.3). While less expressive, propositional languages are attractive because they admit better complexity. Notably, we developed two algorithms that achieve (almost) event-rate independence for MDL (Section 4.4).

4.1 Relational Algebra Normal Form

In database theory, Codd’s theorem [48] states that relational algebra and domain-independent queries expressed using the relational calculus are equally expressive. Relational algebra consists of effectively computable operations on finite relations, whereas the relational calculus is essentially first-order logic. Domain-independence [51] ensures finite query results, independently of the domain that the query’s variables range over. *Relational algebra normal form (RANF)* [1, 55] is a syntactically defined, domain-independent fragment of the relational calculus with a straightforward translation to the algebra.

The policy language fragment supported by MonPoly [27, 31], VeriMon [16, 97], CppMon [56], StaticMon [58], and HashMon [92] is a generalization of RANF from first-order logic to MFOTL. We sometimes call it the *monitorable fragment* [28] (not to be confused with other notions of monitorability [80]). The motivation is the same as for databases: one can translate this fragment directly to operations acting on (streams of) finite relations. In the following, we first describe aspects common to the aforementioned RANF-based tools, thus speaking of a single abstract monitor, before explaining the main differences between the tools.

General Approach. The basic idea is to view the policy formula as a tree whose nodes correspond to relational operations. The monitor processes the input trace incrementally. Every time-point gives rise to a database that supplies the leaves of the tree with relations. Then, the monitor evaluates the tree, bottom up. The relation obtained at the root, which is appended to the in-order output stream, contains the satisfying valuations of the formula. The main difference to the database setting is that some tree nodes, namely those corresponding to temporal operators, are stateful. Future operators are handled by delaying intermediate computations that depend on those operators. Our monitors over-approximate the required delay using the formula’s intervals. Hence they do not detect *minimal* bad prefixes; however, they eventually report a bad prefix when there is one.

Pattern	Constraint	Relational operation
$r(\bar{t})$	$\forall i. t_i \in \mathbb{V} \cup \mathbb{D}$	selection and projection
$\neg \alpha$	$\mathcal{V}(\alpha) = \emptyset$	complement relative to $\{()\}$
$\alpha \wedge \beta$	none	natural join
$\alpha \wedge t_1 \mathcal{R} t_2$	$\mathcal{V}(t_1) \cup \mathcal{V}(t_2) \subseteq \mathcal{V}(\alpha)$	selection
$\alpha \wedge x = t$	$\mathcal{V}(t) \subseteq \mathcal{V}(\alpha), x \notin \mathcal{V}(\alpha)$	generalized projection
$\alpha \wedge \neg \beta$	$\mathcal{V}(\beta) \subseteq \mathcal{V}(\alpha)$	anti-join (generalized difference)
$\alpha \vee \beta$	$\mathcal{V}(\alpha) = \mathcal{V}(\beta)$	union
$\exists x. \alpha$	none	projection
$y \leftarrow \Omega(t; \bar{g}) \alpha$	$\mathcal{V}(t) \cup \bar{g} \subseteq \mathcal{V}(\alpha), y \notin \bar{g}$	group-by and aggregation
$\ominus_I \alpha, \circ_I \alpha$	none	} monitor-specific
$\alpha \mathbf{S}_I \beta, (\neg \alpha) \mathbf{S}_I \beta$	$\mathcal{V}(\alpha) \subseteq \mathcal{V}(\beta)$	
$\alpha \mathbf{U}_I \beta, (\neg \alpha) \mathbf{U}_I \beta$	$\mathcal{V}(\alpha) \subseteq \mathcal{V}(\beta), \text{bounded } I$	

Fig. 3. Relational algebra normal form for a subset of MFOTL

Figure 3 defines the RANF fragment for a subset of our logic; we discuss more advanced operators below for those tools that support them. The first column contains patterns: any formula obtained by instantiating a pattern is in RANF if the instantiations of α and β are in RANF, and the constraints in the second column are satisfied. Formulas can often be rewritten to obtain an equivalent RANF formula, e.g., by applying the distributive law to $\mathbf{p}(x, y) \wedge (\mathbf{q}(x) \vee \mathbf{q}(y))$. However, finding suitable rewrite rules becomes difficult once temporal operators are involved. MonPoly implements a simple but incomplete rewriting procedure. We describe a more general translation in Section 4.2. The third column in Figure 3 describes the relational algebra operation that is used to evaluate formulas matching the pattern. Most are standard [1]. The generalized projection evaluates the term t on each tuple in the input relation to compute the value assigned to x in the output relation. The anti-join generalizes set difference such that the “negative” relation may have a subset of the other relation’s variables. Aggregation operators are computed by first partitioning the relation into groups (if there are grouping variables) and afterwards, for each group, evaluating the term t on the tuples and combining the results using the appropriate aggregation function (e.g., sum, count, or average).

Temporal Operators. The implementation of the temporal operators is specific to the monitoring setting, although temporal-relational algebras have been studied previously [79, 100]. A basic approach, used in MonPoly’s original implementation [26, 28], employs auxiliary relations that are maintained as part of the monitor’s state. For $\ominus_I \alpha$, the auxiliary relation is simply the evaluation result for α at the previous time-point. For $\alpha \mathbf{S}_I \beta$, the auxiliary relation extends the tuples obtained from β with the corresponding time-stamp, which is used to check the interval constraint I . All tuples must satisfy α since the time-point when they were most recently added to the relation. Several optimizations are possible. For example, the special cases \diamond and \diamond benefit from a sliding-window algorithm [30]. A specialized data structure that improves the evaluation time of \mathbf{S} in general was first introduced in VeriMon [17]. The evaluation of future temporal operators is not symmetric to the past operators. Specifically, $\alpha \mathbf{U}_I \beta$ requires an additional auxiliary relation that stores the time-points at which uninterrupted α sequences start.

Example. We explain how a MonPoly-style algorithm monitors the policy $\Box \forall x. \mathbf{p}(x) \rightarrow \Diamond_{[0,3]} \mathbf{q}(x)$. Specifically, it evaluates $\neg\varphi \equiv \mathbf{p}(x) \wedge \neg\chi$, where $\chi \equiv \Diamond_{[0,3]} \mathbf{q}(x)$. All subformulas of $\neg\varphi$ have a single free variable x and their evaluation results are thus all unary relations, i.e., sets. The monitor maintains an additional binary relation S_χ in its state, which is used for the $\Diamond_{[0,3]}$ operator. This relation stores pairs (τ, x) such that the event $\mathbf{q}(x)$ occurred most 3 time units ago, and τ is the most recent time-stamp for the event. For each input $(\tau_i^\sigma, D_i^\sigma, R_i^\sigma)$, corresponding to the time-point i , the monitor proceeds as follows.

- (1) Evaluate $\mathbf{q}(x)$ by computing $E_{\mathbf{q}} = \{x \mid \mathbf{q}(x) \in D_i^\sigma\}$.
- (2) Remove pairs (τ, x) from S_χ where $\tau_i^\sigma - \tau > 3$ or $x \in E_{\mathbf{q}}$, then add (τ_i^σ, x) for all $x \in E_{\mathbf{q}}$. This restores S_χ 's invariant.
- (3) The result of χ is $E_\chi = \{x \mid (\tau, x) \in S_\chi\}$.
- (4) Evaluate $\mathbf{p}(x)$ by computing $E_{\mathbf{p}} = \{x \mid \mathbf{p}(x) \in D_i^\sigma\}$.
- (5) Compute $E_{\neg\varphi} = E_{\mathbf{p}} \setminus E_\chi$ as a special case of anti-join to evaluate $\mathbf{p}(x) \wedge \neg\chi$.
- (6) The formula $\neg\varphi$ is satisfied (i.e., the policy is violated) at time-point i iff $E_{\neg\varphi}$ is non-empty. In this case, the monitor outputs $E_{\neg\varphi}$.

Tool-specific Details and Extensions. The main difference between MonPoly's and VeriMon's algorithms is the scheduling in the presence of delays due to future operators. VeriMon uses buffers attached to every binary operator to "align" the relations computed for the two operands. It evaluates the operator whenever a pair of relations (for the same time-point) is available. The operands are evaluated independently and eagerly. In contrast, MonPoly's scheduling is asymmetric: the second operand is evaluated only once the first has yielded a result, which requires buffering for the atomic predicates. The two strategies differ in their memory usage, which is incomparable because the buffered relations' size depends on the formula.

Extensions compatible with the RANF approach include the "dynamic" operators \triangleleft and \triangleright , as well as the `def` and `rec` constructs. The operators \triangleleft and \triangleright generalize `S` and `U` to regular expressions. We have implemented them in VeriMon using Antimirov's partial derivatives [6]. The syntactic restrictions that guarantee finite relations are subtle, and we refer to the corresponding paper [17] for details. To evaluate `def p(x) := alpha in beta`, our monitors evaluate α eagerly first and buffer any results to be used in the subsequent evaluation of β . For `rec`, which is currently supported by VeriMon and StaticMon, we exploit that only the valuations of \mathbf{p} at past time-points are relevant when evaluating α . Hence no fixpoint computation is required. The tools syntactically check that every use of \mathbf{p} in α is *guarded* by a strict past operator, such as \ominus or \Diamond_I , where I does not include zero. This guarantees that the monitor can eventually evaluate every time-point [102].

The monitors mentioned so far use immutable tree data structures to represent finite relations. CppMon [56] reimplements VeriMon's algorithm in C++ using mutable hash tables. StaticMon develops this idea further by using C++ template metaprogramming [2] to generate an optimized monitor program tailored for each formula. It outperforms MonPoly, VeriMon, and CppMon on many benchmarks [58]. HashMon [92] reuses MonPoly's evaluation algorithm. In addition, HashMon can automatically replace large domain values, such as long strings, by short, randomized hash values to reduce the monitor's memory usage.

4.2 Translation to RANF

The restrictions imposed by the RANF on negations and the subformula's free variables may hamper concise, intuitive formalizations. While one can often rewrite a formula manually into an equivalent RANF representation, this increases the risk of formalization errors. For example, it is difficult to rewrite the formula $\mathfrak{p}(x) \wedge (\mathfrak{q}(x, y) \mathfrak{S} r(y))$, which is not in RANF because of the subformula $\mathfrak{q}(x, y) \mathfrak{S} r(y)$. However, this formula is actually domain-independent as $\mathfrak{p}(x)$ and $r(y)$ jointly provide an upper bound on the set of satisfying valuations.

Raszyk proposed an automatic translation for *arbitrary* relational calculus queries [83] and MFOTL formulas [85] into RANF. The translation, implemented in the tool MFOTL2RANF, introduces an additional free variable x_∞ . If $x_\infty = 1$ in any satisfying valuation, the set of satisfying valuations for the given trace prefix is infinite and no further guarantees are made. This case cannot occur if the original formula is domain-independent. Otherwise, the satisfying valuations without x_∞ correspond precisely to those of the original formula. For the above example, we get (with minor simplifications)

$$\begin{aligned} & \left(\mathfrak{p}(x) \wedge (\mathfrak{q}(x, y) \mathfrak{S} ((\diamond \mathfrak{q}(x, y)) \wedge r(y))) \right) \vee \\ & \mathfrak{p}(x) \wedge (\mathfrak{q}(x, y) \mathfrak{S} (\mathfrak{q}(x, y) \wedge \ominus ((\circ \mathfrak{q}(x, y)) \wedge (\neg \diamond \mathfrak{q}(x, y)) \wedge r(y)))) \vee \\ & \mathfrak{p}(x) \wedge r(y) \wedge \neg \diamond \mathfrak{q}(x, y) \wedge x_\infty = 0. \end{aligned}$$

The translation detects the domain-independence and it sets x_∞ to zero. The three disjuncts correspond to a case distinction over possible origins of relevant values for x in the evaluation of the subformula $\mathfrak{q}(x, y) \mathfrak{S} r(y)$: either there is a $\mathfrak{q}(x, y)$ event concurrent with or before $r(y)$, or the earliest occurrence of $\mathfrak{q}(x, y)$ is strictly within the span of \mathfrak{S} , or there is no such occurrence and $\mathfrak{p}(x)$ serves as the bound.

Adding the automatic translation to our monitors is ongoing work. It is an open question whether and how the translation can be extended to cover additional features of MFOTL and our extensions, such as inequalities, aggregations, and rec.

4.3 Automatic Structures

An alternative approach that lifts the restrictions of RANF is to replace finite relations with *automatic structures* [42, 68]. These structures represent each relation as a finite-state automaton that recognizes those (suitably encoded) tuples that are in the relation. The main advantage is that automatic structures are closed under all Boolean operations, including negation and projection. Moreover, they can represent and operate on (a subset of the) infinite relations.

Binary decision diagrams (BDDs) are an efficient implementation of automatic structures. They are used in an alternative implementation of MonPoly's algorithm called MonPoly-Reg [26, 28], as well as Havelund et al.'s DejaVu tool [61, 62]. The two monitors mainly differ in the encoding of tuples representing valuations. MonPoly-Reg, which supports only integer values, uses the MONA library [64], whose automata natively read multiple variables in parallel. DejaVu translates values to bitstrings, which it then concatenates into tuples.

The use of automatic structures has drawbacks. All operators in terms (e.g., $+$) and all rigid relations (e.g., \leq), must be expressible as regular languages. This generally limits the scope to Presburger arithmetic and none of the above tools handle aggregations. Moreover, the time and memory used by the BDD operations depend on the internal variable ordering and can be unpredictable [102].

4.4 Propositional Monitoring

A monitor’s time and memory performance depends on the sizes of its inputs, i.e., the formula and trace. The latter is typically larger than the former by orders of magnitude. Therefore, one ideally uses *trace-length independent* monitors, whose memory complexity is independent of the trace size. Moreover, for monitors that support real-time constraints, it is desirable that the memory complexity be independent of the trace’s event rate, i.e., the number of events per unit of time. Traces arising in practice have a bound on their event rate, although the bound may be unknown in advance. We henceforth focus on such *(event-rate) bounded traces*.

Both event-rate independent (ERI) and trace-length independent (TLI) monitoring algorithms are not attainable for first-order specifications. For example, monitoring $\diamond p(x)$ requires, in the worst case, memory proportional to the entire trace prefix seen by the monitor. In contrast, TLI monitoring algorithms for the propositional fragments of MFOTL (like MTL) have been proposed in the past. These, however, deviate from our monitoring setting (Section 3): they either do not support future operators [29, 33, 63], only produce a single Boolean verdict for a formula at the trace’s first time-point [53, 99], or access the trace in an offline manner [89]. The challenge is to develop an online ERI algorithm that supports both future operators and produces verdicts for every time-point. Our monitors achieve this by operating in a slightly modified monitoring setting: they either output out-of-order equivalence verdicts, or use multiple reading heads.

Equivalence Verdicts. Our Aerial tool [12, 35] solves the above challenge by outputting verdicts differently. In addition to the standard Boolean verdicts, it outputs equivalence verdicts of the form $j \equiv i$ stating that the verdict at time-point j is identical to the verdict at an earlier time-point $i < j$, although both verdicts are currently unknown. This makes Aerial’s output non-monotonic with respect to time-points and requires slightly more effort to understand. To output equivalence verdicts, the algorithm must refer to natural numbers encoding time-points, which requires logarithmic space as time-points increase with the trace length. Aerial refers to time-points using an offset within a block of consecutive time-points labeled with the same time-stamp. It therefore requires logarithmic space in the event rate, since the size of such a block is bounded by the event rate. Due to this logarithmic dependence, Aerial is an *almost* ERI algorithm.

As an example, consider the policy $\Box (p \rightarrow \diamond_{[0,3]} q)$ similar to the one from Section 4.1, only propositional and with a future $\diamond_{[0,3]}$ operator. The equation

$$\sigma, v, i \models \diamond_I \alpha \quad \text{iff} \quad \sigma, v, i \models \alpha \quad \text{or} \quad \sigma, v, i + 1 \models \diamond_{I-(\tau_{i+1}-\tau_i)} \alpha$$

reduces the satisfaction of $\diamond_I \alpha$ to a disjunction of the satisfaction of α at the same time-point i and the satisfaction of $\diamond_{I-(\tau_{i+1}-\tau_i)} \alpha$ at the next time-point

$i + 1$. The algorithm can immediately compute the satisfaction at the current time-point i , but it must wait for the one at the next time-point. This also means that after processing time-point i , the algorithm cannot store a Boolean verdict for the formula $\diamond_I \alpha$ in its state. Instead, it stores a dependency in the form of a pointer to the part of its state referring to $\diamond_{I-(\tau_{i+1}-\tau_i)} \alpha$, which becomes available after processing time-point $i + 1$. More generally, for every (interval-skewed) subformula (recall $\text{ISF}(\cdot)$ from Section 2), the algorithm stores a Boolean combination of such dependencies in the form of symbolic Boolean expressions. By processing the subsequent time-points, the algorithm may resolve some expressions to Boolean values and output them as verdicts. Crucially, if the algorithm detects two semantically equivalent Boolean expressions for the top-level formula at different time-points, it outputs an equivalence verdict and removes one of the two expressions from its state. As the number of semantically different Boolean expressions only depends on the formula, so does the space needed to store them. Aerial extends this idea to MDL operators \triangleleft and \triangleright using partial derivatives [6].

Multi-head Monitoring. Our Hydra tool [84,87] implements an ERI algorithm that supports both past and future operators, but, unlike Aerial, produces *Boolean* verdicts in time-point order. It achieves this by using multiple independent and unidirectional reading heads. If an event is needed for subsequent analysis after it was read, a standard online monitor must keep it in its memory. The idea of using multiple heads is to avoid this memory usage and rely on one of the reading heads to read the event again. The way Hydra reads its input trace makes it neither an online nor an offline monitor. An online monitor does not require the trace to be persistent, whereas Hydra requires this for the part of the trace between its first and last reading head. In contrast, an offline monitor has a reading head without movement constraints, while all of Hydra’s reading heads are unidirectional.

Conceptually, a multi-head monitor for an MTL formula φ is built recursively from multi-head (sub-)monitors, one for each direct subformula of φ . The total number of reading heads equals the number of atomic predicates in φ . The algorithm recursively *steps* the monitors in a loop, where a step either advances one reading head or propagates a cached verdict from a sub-monitor to its parent. Once every sub-monitor for φ has produced a verdict, the algorithm computes as many verdicts for φ as possible (which may be none) based on the MTL semantics of φ ’s top-level operator. For example, Hydra monitors the policy $\square (\mathbf{p} \rightarrow \diamond_{[0,3]} \mathbf{q})$ using two reading heads for \mathbf{p} and \mathbf{q} . The $\diamond_{[0,3]}$ operator stores a run-length encoded list of integers. In the list, zeros representing time-points are interleaved with the (positive) time-stamp differences between them. The list encodes a sequence of time-points, spanning a time-stamp difference of at most 3, at which \mathbf{q} is *not* satisfied. It is updated using the verdicts from the head for \mathbf{q} , which runs ahead. Specifically, whenever the head reports a \mathbf{q} event, all time-points in the list become positive verdicts for the $\diamond_{[0,3]}$ operator (after checking the interval constraint). Finally, the \rightarrow operator combines the verdicts returned by its sub-monitors.

Hydra generalizes [86] the idea of multi-head monitoring to MDL operators \triangleleft and \triangleright using a number of heads exponential in the formula’s size [87]. Both Aerial and Hydra outperform MonPoly on their specialized fragment.

5 Parallelization

The algorithms described in Section 4 all execute sequentially. The only way to make them faster (beyond clever optimizations) is to use a faster processor, which clearly has its limits. It is thus natural to ask how one can parallelize the existing algorithms or develop new parallel ones, which is not straightforward as the linear nature of traces results in a bias towards sequential processing. Theoretical results are promising: Kuhtz and Finkbeiner [70] and later Bundala and Ouaknine [44] showed that LTL and MTL monitoring over finite traces is in the highly parallelizable circuit complexity class NC. However, these results do not generalize to first-order policies, where the complexity rises to PSPACE-complete [37], which likely rules out fast parallel algorithms. We must therefore resort to a best effort strategy.

We discuss task-parallel and data-parallel approaches. With task parallelism, independent operations within the monitor are executed in parallel. Data parallelism partitions the data instead, i.e., the trace. While parallel monitoring, distributed monitoring, and monitoring of distributed systems all have different requirements, there are some connections that we discuss in Section 5.3.

5.1 Scalable Offline Monitoring

Offline monitors are more easily parallelized than online ones because all data is available from the start. We summarize results on applying the MapReduce framework [49] to offline monitoring. MapReduce is suitable for computations on large sets of data items. There are two phases. In the first phase, each data item is *mapped* (transformed) individually and the results are each assigned a key, which is used for grouping. In the second phase, each group is independently *reduced* to a result. Clearly, both phases are parallelizable provided the groups are not too large.

Barre et al. [10] evaluate LTL formulas bottom up using one round of MapReduce for each layer of the formula tree. They combine task and data parallelism: the map phase operates on all time-points and operators in the current layer, whereas the reduce phase is organized by operators only. Follow-up work generalized this approach to MTL with aggregations [41] and a more fine-grained partitioning of temporal operators based on their interval constraints [40].

We developed a slicing framework [14, 15] based on MapReduce and data parallelism. The map phase applies *slicers* to partition the input trace into a finite collection of slices, which are again traces. Each slice is associated with a *restriction*, a subset of the valuations and time-stamps that the monitor may output for a given policy formula. In the reduce phase, MonPoly is used as a *submonitor* that evaluates the formula on each slice. Its outputs are intersected with the corresponding restriction and combined. For correctness, it is important that the slices are sound and complete with respect to their restrictions. Slices need not be (and, in general, are not) disjoint. However, by choosing the slicers carefully, each slice has significantly fewer events than the input trace, such that each parallel invocation of MonPoly runs for a shorter amount of time and uses less memory.

There are two fundamental types of slicers, which may be composed. A *data slicer* selects events as a function of the events' parameters (domain values). It

is parametrized by one of the formula’s free variables x and a subset S_x of the domain. Any event that may be involved in a satisfying valuation v of the formula, such that $v(x) \in S_x$, is included in the slice. This is determined using a static over-approximation. For example, for the formula $p(x) \wedge \neg(\exists y. q(x, y) \vee q(y, x))$, the slice $S_x = \{3\}$ receives the events $p(3)$, $q(3, c)$, and $q(c, 3)$ for all c .

In contrast, the *time slicer* considers the events’ time-stamps. The basic idea is to split the trace into contiguous chunks. However, if the formula uses temporal operators, they cannot be evaluated correctly near the chunk boundaries. Therefore, there must be a sufficient overlap between adjacent chunks. This overlap can again be computed statically in advance based on the formula’s *relative intervals*. The relative interval of $p \wedge \diamond_{[1,2]} (q \vee \diamond_{[7,7]} r)$ is $[-2, 6]$, for instance, as any relevant event is contained within that interval relative to the current time-stamp.

5.2 Scalable Online Monitoring

MapReduce was designed for offline (batch) processing and it is not directly suitable for low-latency online monitors. In contrast, the *data stream* model of computation is tailored to continuous queries over rapidly changing data [9], which online monitoring can be seen as an instance of. Data stream management systems (DSMS) are generic platforms that provide high-level abstractions, while taking care of common issues in large-scale data stream processing such as scheduling, distributed execution, and fault tolerance [4, 46, 101].

Our online slicing framework [94, 96] transplants the data slicing approach onto the Apache Flink DSMS [4], which parallelizes stream processing using multiprocessing or a distributed cluster. Data slicing is more useful for online monitoring than time slicing as the reduction in the maximum number of events across all slices is often higher over short periods of time. The main criterion for choosing Flink was its support for distributed snapshots [45], which enables fault-tolerance. If a machine in a distributed monitoring cluster fails, for example, the monitor can restart from the latest snapshot, which reduces the latency until it catches up with the event stream. To this end, we implemented a custom operator for Flink’s data flow that can extract MonPoly’s state.

We improved the slicing framework along several dimensions. The *joint data slicer* takes all free variables of the policy formula into account. Parametrized by a *slicing strategy* (an assignment of valuations over the free variables to slice identifiers), it computes for every event a subset of target slices that the event must be included in. We extended the joint data slicer with support for **def** and **rec** and identified a policy fragment for which the submonitors’ outputs need not be filtered against the slicing strategy [93], which is otherwise required by (joint) data slicers.

Moreover, we developed an automatic slicing strategy selection for the joint data slicer. It adapts the *hypercube* algorithm for the parallel processing of relational joins [3, 38] to our setting. The algorithm is so called because it partitions the domain of every variable separately, such that each slice corresponds to a hypercube of the product space over all variables. The number of splits per variable is optimized to minimize event duplication. This requires specific statistics of the event stream, such as the relative frequency of the different event names.

These statistics may change substantially over time in a long-running stream and so may the corresponding optimal strategy. To change the strategy at runtime, we developed a state splitting and merge interface for MonPoly [95], since the submonitors’ states must be kept consistent with the current slicing strategy.

Both the offline and the online framework are *black box* approaches because they rely on a standard, non-parallelized monitoring tool. This is convenient because all tool optimizations are readily available and the implementation can be changed relatively easily. For example, we have used not only MonPoly but also DejaVu with our online framework. However, it is known specifically for joins that redistributing data in multiple rounds may improve performance [38]. This corresponds to exchanging data between individual operators in the monitor’s execution. We describe such a parallel *white box* monitor in Section 5.3.

5.3 Monitoring Distributed Systems

We focus on centralized specifications, which take a holistic view of a distributed system and abstract away from its structure [52]. Many temporal logics for centralized specifications—including MFOTL—assume that a total order is defined over all events. Yet it is often difficult to determine the true order of events generated by different components in an asynchronous distributed systems. The uncertainty about the ordering can be reduced, but not eliminated, using logical clocks such as vector clocks [75]. A global physical (real-time) clock may be approximated by employing synchronization protocols, but it has limits in environments with high event rates [39]. The RV community has developed many approaches that try to circumvent these obstacles [54]. Here we summarize our contributions to this area.

The *interleaving-sufficient fragment* [21,22] is a syntactically defined subset of MFOTL that can be monitored correctly on any interleaving of traces from different sources (e.g., components), meaning that the formula is either satisfied or violated on any interleaving. The only assumption is that a global clock with a possibly low resolution creates the time-stamps across all traces. The *collapse-sufficient fragment* is contained in the interleaving-sufficient fragment and consists of formulas that can be monitored correctly on the collapse, where all events with the same time-stamp are combined into a single time-point. Such fragments are useful because determining whether any (or all) interleavings satisfy a propositional formula is already (co-)NP-complete [22].

We used the collapse-sufficient fragment with the online slicing framework to parallelize slicing itself [19]. In the original framework, the slicer can become a bottleneck if the event rate (number of events per second) is too high. The idea is to slice the streams from each source in parallel and then merge the incoming slices at each submonitor. If the monitored policy is collapse-sufficient, it suffices to sort and group the events by their time-stamp using a (small) buffer. However, this approach still requires a low-resolution global clock. For a propositional fragment, we have shown that monitoring is possible even if the clock has bounded error [23].

When the monitor is operating in a distributed setting, messages sent may get lost or arrive out-of-order and components may even crash. Basin et al. [32,34] developed a monitoring algorithm that uses a three valued Kleene logic to soundly

operate in the presence of knowledge gaps; the third value \perp stands for “unknown”. Reasoning is monotonic with respect to a partial order on truth values where \mathbf{t} and \mathbf{f} are incomparable, and both are greater than \perp . Hence verdicts, once emitted, are never retracted, even when knowledge gaps are filled as events come in, out-of-order. The out-of-order monitor (POLIMON) supports the language MTL^\downarrow , which is MTL augmented with freeze quantifiers, where \downarrow is a quantifier that extracts data values from registers and bind these values to logical variables.

A second, recently developed monitor for the out-of-order-setting is TimelyMon [88]. TimelyMon supports the RANF fragment of MFOTL with proper quantification and is thus more expressive than POLIMON, which is limited to freeze quantification. TimelyMon can receive individual events (not databases) in any order, but expects them to be labeled with the correct time-points and time-stamps (as defined by the temporal structure). It outputs assignments out-of-order, which allows it to signal policy violations much earlier than MonPoly and VeriMon, whose verdicts are delayed by the future interval bounds. Technology-wise, TimelyMon is implemented in the Timely Dataflow DSMS [76] and thus constitutes a white box implementation of a data-parallel online monitor (Section 5.2). Initial experiments confirm TimelyMon’s good scalability with increasing numbers of workers, which is simply a parameter in Timely Dataflow.

6 Verification

Monitors use complex, optimized algorithms to efficiently support expressive specification languages. These algorithms’ correctness is rarely obvious. Even worse, pen-and-paper proofs of correctness usually reason about idealized pseudocode. These proofs can be faulty, and so can be the translation from pseudocode to code. Over the years, we have found and fixed various errors in MonPoly’s code.

VeriMon [16, 17, 97, 102] was conceived out of our frustration with this build-break-fix cycle. Our original goal was to create a simplified version of MonPoly with strong correctness guarantees, machine-checked using the interactive theorem prover Isabelle/HOL [78]. To this end, we have formalized MFOTL’s syntax and semantics, the simplified monitor’s specification, and its correctness statement. We then defined invariants on the monitor’s state and proved in Isabelle that they are preserved by the monitor’s steps and imply the correctness statement. Finally, using Isabelle’s code generator [59], we extracted executable OCaml code from our formalization. The resulting functional program, augmented with MonPoly’s unverified formula and log parsers and user interface is what we call VeriMon.

Is VeriMon more trustworthy than MonPoly? Isabelle will not accept a vague or incomplete argument: all reasoning passes through Isabelle’s kernel, which is a trustworthy guardian. Since Isabelle accepts the proof of VeriMon’s correctness statement (expressing that the monitor’s output complies with the specified MFOTL semantics), errors can only happen in reused, unverified parts of MonPoly’s code or in the formal specification of MFOTL. The actual monitoring algorithm, arguably the most complex part of a monitor, is error-free. To further increase trustworthiness, we are working on verifying the unverified code used by

VeriMon and validating MFOTL’s specification by manual inspection, asserting that it faithfully represents what we intend to model.

VeriMon proved useful beyond its trustworthiness. Verifying a monitor has significantly improved our own understanding of the matter and provided us with a platform for experimentation and growth. VeriMon has become the incubator for first-order monitoring research. For example, new constructs like temporal regular expressions [17] and recursive definitions [102], previously unseen in any first-order monitor supporting future temporal operators, were introduced in VeriMon. We have also optimized VeriMon’s simplified algorithms, sometimes inspired by optimizations used in MonPoly, other times going beyond, e.g., by incorporating multi-way joins from databases [17]. Moreover, we developed entirely new components, such as a type system and a type inference algorithm. In all cases, we took care, with Isabelle’s help, to maintain or extend the correctness proof.

Eventually, the new features started migrating to other, unverified monitors, which still outclass VeriMon in efficiency. We have also used VeriMon as a reliable testing oracle. Differential testing on random inputs revealed several previously unknown implementation errors in MonPoly and helped us to localize them [17].

VeriMon is not our only verified monitor. The multi-head monitor Hydra (Section 4.4) has a verified counterpart, Vydra, and the online slicing framework’s core (Section 5.2) is also formally verified. We firmly believe that theorem proving is a must when the goal is to develop and implement complex algorithms one can trust.

7 Applications

Research in monitoring strongly benefits from the plethora of immediate applications and the close interplay between theory and practice. Theoretical advances in monitoring lead to performance improvements in terms of memory use, execution time, parallelism, and even metatheoretic guarantees about what monitoring achieves. Conversely, applications provide insights on which features are useful in practice and whether tools scale in realistic settings. Although the scope of applications for monitoring is wide, our focus has been on problems in security, data protection, and protocols for distributed systems.

7.1 Security and Anomaly Detection

Security policies regulate which actions may and must not happen within a system. The vast majority of these policies constitute safety properties (the exceptions are typically information flow policies, which are hyperproperties). Hence, they are excellent candidates for monitoring since policy violations are detectable on finite traces. Also relevant for security is that monitors can be used to detect anomalous behavior, for example for intrusion detection. Such applications benefit from statistical computations as offered by our logic’s aggregations.

A prototypical security policy has the form $\Box \forall \bar{x}. \text{action}(\bar{x}) \rightarrow \text{authorized}(\bar{x})$. Namely, every occurrence of some *action* must be authorized. The \bar{x} are parameters associated with the action, e.g., attributes of the user responsible for the action, the resource(s) used, or the environment. Moreover, *authorized*

specifies that authorization is present or, alternatively, is a formula specifying the conditions for authorization. As a concrete example

$$\Box \forall u, a, o. \text{exec}(u, a, o) \rightarrow \exists r. \text{UA}(u, r) \wedge \text{PA}(r, a, o)$$

might formalize access restrictions in a system implementing an access control mechanism based on some variant of Role-based Access Control (RBAC). It states that whenever a user u carries out an action a on an object o , then the user is assigned to the role r under the user assignment relation UA and moreover the role r is granted the privilege to carry out action a on the object o , under the permission assignment relation PA . We provide extensive examples of how security policies can be formalized in MFOTL [25], ranging from simple access control requirements like the above to more complex policies formalizing history-based access control policies like so-called Chinese Wall policies, where access rights change dynamically with each access, and separation-of-duty requirements.

We carried out a large-scale case study with Google, monitoring compliance to access control policies of Google employees using Google’s infrastructure [15]. Policies concerned configurations of accessing computers, time limits on the use of authentication tokens, and restrictions on software used during access. We used off-line monitoring, taking events from a distributed logging infrastructure recording log data on roughly 35,000 computers accessing sensitive resources over a period of two years. The log data contained roughly 77.2 million time-points and 26 billion events, and required 0.4 TB to store in a compressed form. For each policy, we used 1,000 computers for slicing and monitoring. The original MonPoly system was used together with the offline slicing framework (Section 5.1) leveraging Google’s MapReduce infrastructure. Namely, we split the log into 10,000 slices whereby each computer processed 10 slices on average. Overall, processing time was on the order of hours (2–12 hours), with the vast majority of time being spent on monitoring, and it scaled well with the introduction of more computing resources.

In the context of anomaly detection, we developed policies that aim at identifying fraudulent reviews in an e-commerce setting. The first policy, based on an algorithm by Heydari et al. [65], detects outliers in the number of reviews received by a brand. This required encoding a *tumbling window* [46] by combining aggregations and temporal operators. We applied the policy to a dataset of reviews published on Amazon [77] to evaluate HashMon. Hashing the review texts reduced the memory usage by a third [92]. The other policies detect brands whose products receive identical reviews (as determined by the score and, optionally, the text) from the same user. These policies were designed to be challenging for our monitors as it is difficult or impossible to rewrite them in RANF (Section 4.1). The formulas obtained from the MFOTL2RANF tool outperformed other approaches, including MonPoly-Reg, on synthetic data and the Amazon data [85].

7.2 Privacy and Data Protection

We have also carried out case studies on using monitoring to check compliance to privacy and data protection policies. We first used MonPoly in a case study with Nokia [22], which revolved around the use of cell-phone data of participants and en-

sure compliance to privacy policies by auditing logs for proper usage of this data. For example, policies required that data would only be propagated to certain systems, that appropriate anonymization steps would be taken prior to sharing, data requested for deletion would actually be deleted from all appropriate systems, etc.

In more recent work [8], we formalized a substantial part of the GDPR in MFOTL. The GDPR has challenges that go beyond traditional access control. For example, once access is granted, data may only be used when there is a legal basis for the usage or users have granted explicit consent. Users may also restrict how their data is processed at any time and have the right to have their data deleted. Our formalization of such rights provided a basis for using MonPoly to determine GDPR compliance. We carried out a case study on the use of sensitive data by a research foundation concerning how they evaluated and awarded grant applications.

Finally, we have used MonPoly as an enforcement component [67] in a data protection framework called *Taint, Track, and Control* (TTC) [66]. Applications developed in TTC natively use dynamic information-flow control to track the provenance information of every value in their memory and persistent storage. This information includes the identifiers of all user whose data affected the value. Users can formulate their data protection policies in MFOTL, and TTC determines if revealing a value conforms to the policies. In particular, the application’s execution trace (including the provenance information) is monitored by MonPoly and, based on its output, all attempted violations are prevented.

7.3 Distributed Systems

A significant challenge in the Nokia case study mentioned above was that the data was stored in multiple logs collected from components of a distributed system. Hence, even assuming synchronized clocks, there is only a partial order on time-stamped data rather than a total order assumed by MFOTL’s semantics. We tackled this problem by expressing policies with formulas in MFOTL’s collapse-sufficient fragment and monitoring the collapse of the trace (Section 5.3).

We also used MonPoly in a case study to check properties of the Internet Computer (IC) [18]. The IC is a complex distributed system that facilitates governance and execution of Web3 applications and spans over 1,200 nodes worldwide. Web3 applications process data with decentralized ownership (e.g., financial assets). Hence the integrity of their execution must be ensured on devices beyond the asset owner’s control. The IC ensures this by combining an efficient consensus protocol and state machine replication [50]. The efficiency of the IC’s consensus protocol is achieved by grouping all IC nodes into subnets of manageable size. The configuration of the IC (e.g., the assignment of the nodes to subnets) is highly dynamic and the IC possesses numerous other features that are challenging to monitor, such as a long-lived high event-rate execution, a layered software architecture, and continuous evolution. The policies we have formalized range from common symptoms of the IC’s production incidents to properties of the IC’s consensus protocol, including malicious behaviors and infrastructure outages that the protocol must tolerate.

For example, Figure 4 shows our formalization of the `logging-behavior` IC policy. The policy first computes the current assignment of nodes to subnets (pred-

```

def NodeAdded( $n, s$ ) := InSubnet0( $n, s$ )  $\vee$  RegistryAddNodeTo( $n, s$ ) in
def InSubnet( $n, s$ ) :=  $\neg$ RegistryRemoveNodeFrom( $n, s$ )  $\text{S}$  NodeAdded( $n, s$ ) in
def ProperEvent :=  $\diamond_{[1, \infty)}$   $\top$  in
def RelevantNode( $n, s$ ) := InSubnet( $n, s$ )  $\text{S}_{[10 \text{ min}, \infty)}$  (InSubnet( $n, s$ )  $\wedge$  ProperEvent) in
def RelevantLog( $n, s, lvl, msg, i$ ) :=  $\exists host, comp.$ 
  Log( $host, n, s, comp, lvl, msg$ )  $\wedge$   $comp \stackrel{\text{RE}}{=} \text{"orchestrator"} \wedge n \neq "" \wedge \text{tp}(i)$  in
def MsgCount( $n, s, k$ ) :=  $k \leftarrow \text{SUM}(k'; n, s)$ 
  (
    (
      ( $k' \leftarrow \text{CNT}(i; n, s) \diamond_{[0, 10 \text{ min}]}$  RelevantLog( $n, s, lvl, msg, i$ ))  $\wedge$ 
      RelevantNode( $n, s$ ))  $\vee$  (RelevantNode( $n, s$ )  $\wedge k' = 0$ )
    ) in
def TypicalBehavior( $s, m$ ) := ( $m \leftarrow \text{MED}(k; s) \text{MsgCount}(n, s, k)$ )  $\wedge$ 
  ( $\exists k. (k \leftarrow \text{CNT}(n; s) \text{RelevantNode}(n, s)) \wedge k \geq 3$ ) in
def TypicalBehaviors( $s, m$ ) :=
  ( $\diamond_{[0, 10 \text{ min}]}$  TypicalBehavior( $s, m$ ))  $\vee$  ( $\diamond_{[0, 10 \text{ min}]}$  TypicalBehavior( $s, m$ )) in
def BehaviorAndRange( $s, n, k, min, max$ ) :=  $\neg$  ( $\diamond_{[0, 10 \text{ min}]}$  EndTest)  $\wedge$ 
  ( $\exists k'. \text{MsgCount}(n, s, k') \wedge k = \text{int2float}(k')$ )  $\wedge$ 
  ( $min \leftarrow \text{MIN}(m; s) \text{TypicalBehaviors}(s, m)$ )  $\wedge$ 
  ( $max \leftarrow \text{MAX}(m; s) \text{TypicalBehaviors}(s, m)$ ) in
def Exceeds( $s, n, k, min, max$ ) :=
  (BehaviorAndRange( $s, n, k, min, max$ )  $\wedge k > 1.1 \cdot max$ )  $\vee$ 
  (BehaviorAndRange( $s, n, k, min, max$ )  $\wedge k < 0.9 \cdot min$ ) in
Exceeds( $s, n, k, min, max$ )  $\wedge \neg \ominus_{[0, 10 \text{ min}]}$  ( $\exists a, b, c. \text{Exceeds}(s, n, a, b, c)$ )

```

Fig. 4. The Internet Computer’s logging-behavior policy [18]

icate InSubnet) based on the IC’s initial configuration (InSubnet₀) and the nodes that have joined (RegistryAddNodeTo) or left (RegistryRemoveNodeFrom) a subnet. Next, for each subnet the policy compares its nodes’ logging frequencies computed over a 10 minute sliding window (MsgCount) against the median logging frequency over all nodes in the subnet (TypicalBehavior). Only messages containing `orchestrator` in their component name are relevant for the frequency calculation.

The IC’s execution traces were recorded in a detailed JSON format, which required a non-trivial mapping to more abstract events (e.g., RegistryAddNodeTo) with appropriate parameters. This motivated a recent extension of MFOTL and MonPoly with complex data types, like records, variants, and recursive types [74].

8 Conclusions and Open Problems

Monitoring is a fascinating research area given the rich interplay between theory and practice. While the gold standard for system verification is the full verification of implementations using model-checkers and theorem provers, monitoring offers an attractive alternative. Not only is monitoring relatively lightweight and easy to use, it has a larger scope. Namely, one can monitor extremely complex systems, even involving humans and non-technical components, provided one has policies for their behavior. Moreover the verdicts returned are statements about the actual system’s behavior, rather than a mathematical model thereof. Below we discuss some research questions and open problems that have arisen from our work.

Whenever monitoring is used in practice, the question arises how to handle policy violations. We learned from our IC case study (Section 7.3) that engineers value detailed and precise information about violations, as it helps them identify and fix the root cause more quickly. As a first step towards explainable and certifiable monitor verdicts, we have developed a monitor for MTL that outputs minimal proof objects [13, 73]. Can one go farther and design a feedback loop that aids with fault localization by matching such certificates against the monitored system?

Both a monitor’s performance and correctness are critical. VeriMon is frequently outperformed by the unverified tools MonPoly and StaticMon. We believe there are two main reasons for this performance gap: the exclusive use of immutable data structures and the layers of abstractions that were vital for the proofs but cannot be simplified by the compiler. Our long-term goal is to refine VeriMon to a highly efficient, imperative implementation. Despite impressive advances in verified refinement [7, 71], the complex, recursive invariants of VeriMon’s state require new ideas to break this effort down into manageable and composable parts.

Complex policies are often built from abstract concepts that must be made precise for monitoring. For example, in the IC case study from Section 7.3, the predicate `TypicalBehavior` was defined as the median logging frequency of nodes in a subnetwork. One could well imagine that what constitutes typical behavior is something that can be learned, using machine learning, rather than specified a priori. Combining monitoring with machine learning is an exciting topic, with many applications, e.g., in security, anomaly detection, and beyond.

Acknowledgments This paper has four authors, but it reports on a decade of collaboration with numerous other researchers. We would like to explicitly name some of them here. First and foremost, we thank Felix Klaedtke, Martin Raszyk, and Eugen Zălinescu. Felix and Eugen were key contributors during the inception of MonPoly. Martin arrived later but left his mark through his work on Hydra, Vydra, MFOTL2RANF, and VeriMon.

We also thank the past and present monitoring aficionados from our groups at ETH Zürich and the University of Copenhagen: Bhargav Bhatt, Rafael Castro G. Silva, Matús Harvan, François Hublet, Jonathan Julián Huerta y Munive, Leonardo Lima, Srđan Marinović, Samuel Müller, Lennard Reese. In addition, we are grateful to those B.Sc. and M.Sc. students who contributed to our journey: Berkay Aydogdu, Marc Bolliger, Frederik Brix, Thibault Dardinier, Christian Fania, Artur Gigon Almada e Melo, Matthieu Gras, Emma Pind Hansen, Nico Hauser, Lukas Heimes, Andrei Herasimau, Hróbjartur Höskuldsson, Valeria Jannelli, Nicolas Kaletsch, Jeniffer Lima Graf, Emanuele Marsicano, Galina Peycheva, Sarah Plocher, Jonathan Rappl, Pascal Schärli, Dawit Legesse Tirore, Adrian Wortmann, Simon Yuan, Stefan Zemljic, Sheila Zingg, and Remo Zumsteg. We would also like to thank our external collaborators from the past and present: Emma Arfelt, Daniel Bristot de Oliveira, Germano Caronni, Søren Debois, Daniel Stefan Dietiker, Sarah Ereth, Yliès Falcone, Heiko Mantel, Birgit Pfitzmann, Yvonne-Anne Pignolet, Giles Reger, Arshavir Ter-Gabrielyan, as well as the participants of the ARVI COST Action and many (mostly) anonymous reviewers.

Finally, we acknowledge the generous external funding we have received for research on monitoring from the Swiss National Science Foundation (grant 167162 “Big Data Monitoring” and grant 204796 “Model-driven Security & Privacy”), the US Air Force Research Laboratory (grant FA9550-17-1-0306 “Monitoring at Any Cost”), and the Novo Nordisk Foundation (start package grant NNF20OC0063462).

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.
3. Foto N. Afrati and Jeffrey D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.
4. Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.
5. Mack W. Alford, Leslie Lamport, and Geoff P. Mullery. Basic concepts. In M. Paul and H. J. Siegart, editors, *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, volume 190 of *LNCS*, pages 7–43. Springer, 1984.
6. Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.
7. Arvind Arasu, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Aymeric Fromherz, Kesha Hietala, Bryan Parno, and Ravi Ramamurthy. FastVer2: A provably correct monitor for concurrent, key-value stores. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic, editors, *12th ACM SIGPLAN Int. Conf. on Certified Programs and Proofs (CPP 2023)*, pages 30–46. ACM, 2023.
8. Emma Arfelt, David Basin, and Søren Debois. Monitoring the GDPR. In Kazuo Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *24th Eur. Symp. on Research in Computer Security (ESORICS 2019), Part I*, volume 11735 of *LNCS*, pages 681–699. Springer, 2019.
9. Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis, editors, *21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS 2002)*, pages 1–16. ACM, 2002.
10. Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier, and Sylvain Hallé. MapReduce for parallel trace validation of LTL properties. In Shaz Qadeer and Serdar Tasiran, editors, *3rd Int. Conf. on Runtime Verification (RV 2012)*, volume 7687 of *LNCS*, pages 184–198. Springer, 2012.
11. Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification: Introductory and Advanced Topics*, volume 10457 of *LNCS*. Springer, 2018.
12. David Basin, Bhargav Nagaraja Bhatt, Srdan Krstić, and Dmitriy Traytel. Almost event-rate independent monitoring. *Formal Methods Syst. Des.*, 54(3):449–478, 2019.
13. David Basin, Bhargav Nagaraja Bhatt, and Dmitriy Traytel. Optimal proofs for linear temporal logic on lasso words. In Shuvendu K. Lahiri and Chao Wang, editors, *16th Int. Symp. on Automated Technology for Verification and Analysis (ATVA 2018)*, volume 11138 of *LNCS*, pages 37–55. Springer, 2018.
14. David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *5th Int. Conf. on Runtime Verification (RV 2014)*, volume 8734 of *LNCS*, pages 31–47. Springer, 2014.

15. David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring of temporal specifications. *Formal Methods Syst. Des.*, 49(1-2):75–108, 2016.
16. David Basin, Thibault Dardinier, Nico Hauser, Lukas Heimes, Jonathan Julián Huerta y Munive, Nicolas Kaletsch, Srđan Krstić, Emanuele Marsicano, Martin Raszyk, Joshua Schneider, Dawit Legesse Tirore, Dmitriy Traytel, and Sheila Zingg. VeriMon: A formally verified monitoring tool. In Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu, editors, *19th Int. Colloq. on Theoretical Aspects of Computing (ICTAC 2022)*, volume 13572 of *LNCS*, pages 1–6. Springer, 2022.
17. David Basin, Thibault Dardinier, Lukas Heimes, Srđan Krstić, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. A formally verified, optimized monitor for metric first-order dynamic logic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *10th Int. Joint Conf. on Automated Reasoning (IJCAR 2020)*, *Part I*, volume 12166 of *LNCS*, pages 432–453. Springer, 2020.
18. David Basin, Daniel Stefan Dietiker, Srđan Krstić, Yvonne-Anne Pignolet, Martin Raszyk, Joshua Schneider, and Arshavir Ter-Gabrielyan. Monitoring the Internet Computer. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *25th Int. Symp. on Formal Methods (FM 2023)*, volume 14000 of *LNCS*, pages 383–402. Springer, 2023.
19. David Basin, Matthieu Gras, Srđan Krstić, and Joshua Schneider. Scalable online monitoring of distributed systems. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *20th Int. Conf. on Runtime Verification (RV 2020)*, volume 12399 of *LNCS*, pages 197–220. Springer, 2020.
20. David Basin, Matúš Harvan, Felix Klaedtke, Srđan Krstić, Martin Raszyk, Joshua Schneider, Dmitriy Traytel, Eugen Zălinescu, et al. MonPoly and VeriMon. <https://bitbucket.org/jshs/monpoly>.
21. David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. Monitoring usage-control policies in distributed systems. In Carlo Combi, Martin Leucker, and Frank Wolter, editors, *18th Int. Symp. on Temporal Representation and Reasoning (TIME 2011)*, pages 88–95. IEEE, 2011.
22. David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. Monitoring data usage in distributed systems. *IEEE Trans. Software Eng.*, 39(10):1403–1426, 2013.
23. David Basin, Felix Klaedtke, Srđan Marinović, and Eugen Zălinescu. On real-time monitoring with imprecise timestamps. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *5th Int. Conf. on Runtime Verification (RV 2014)*, volume 8734 of *LNCS*, pages 193–198. Springer, 2014.
24. David Basin, Felix Klaedtke, Srđan Marinović, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods Syst. Des.*, 46(3):262–285, 2015.
25. David Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In James B. D. Joshi and Barbara Carminati, editors, *15th ACM Symp. on Access Control Models and Technologies (SACMAT 2010)*, pages 23–34. ACM, 2010.
26. David Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *22nd Int. Conf. on Computer Aided Verification (CAV 2010)*, volume 6174 of *LNCS*, pages 1–18. Springer, 2010.
27. David Basin, Felix Klaedtke, Samuel Müller, and Birgit Pfizmann. Runtime monitoring of metric first-order temporal properties. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *IARCS Annual Conf. on Foundations of*

- Software Technology and Theoretical Computer Science (FSTTCS 2008)*, volume 2 of *LIPICs*, pages 49–60. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2008.
28. David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.
 29. David Basin, Felix Klaedtke, and Eugen Zălinescu. Algorithms for monitoring real-time properties. In Sarfraz Khurshid and Koushik Sen, editors, *2nd Int. Conf. on Runtime Verification (RV 2011), Revised Selected Papers*, volume 7186 of *LNCS*, pages 260–275. Springer, 2011.
 30. David Basin, Felix Klaedtke, and Eugen Zălinescu. Greedily computing associative aggregations on sliding windows. *Inf. Process. Lett.*, 115(2):186–192, 2015.
 31. David Basin, Felix Klaedtke, and Eugen Zălinescu. The MonPoly monitoring tool. In Giles Reger and Klaus Havelund, editors, *Int. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES 2017)*, volume 3 of *Kalpa Publications in Computing*, pages 19–28. EasyChair, 2017.
 32. David Basin, Felix Klaedtke, and Eugen Zălinescu. Runtime verification of temporal properties over out-of-order data streams. In Rupak Majumdar and Viktor Kunčák, editors, *29th Int. Conf. on Computer Aided Verification (CAV 2017), Part I*, volume 10426 of *LNCS*, pages 356–376. Springer, 2017.
 33. David Basin, Felix Klaedtke, and Eugen Zălinescu. Algorithms for monitoring real-time properties. *Acta Informatica*, 55(4):309–338, 2018.
 34. David Basin, Felix Klaedtke, and Eugen Zălinescu. Runtime verification over out-of-order streams. *ACM Trans. Comput. Log.*, 21(1):5:1–5:43, 2020.
 35. David Basin, Srđan Krstić, and Dmitriy Traytel. AERIAL: almost event-rate independent algorithms for monitoring metric regular properties. In Giles Reger and Klaus Havelund, editors, *Int. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES 2017)*, volume 3 of *Kalpa Publications in Computing*, pages 29–36. EasyChair, 2017.
 36. David Basin, Srđan Krstić, and Dmitriy Traytel. Almost event-rate independent monitoring of metric dynamic logic. In Shuvendu K. Lahiri and Giles Reger, editors, *17th Int. Conf. on Runtime Verification (RV 2017)*, volume 10548 of *LNCS*, pages 85–102. Springer, 2017.
 37. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In Axel Legay and Saddek Bensalem, editors, *4th Int. Conf. on Runtime Verification (RV 2013)*, volume 8174 of *LNCS*, pages 59–75. Springer, 2013.
 38. Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017.
 39. Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John C. Linford. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Comput.*, 35(12):595–607, 2009.
 40. Marcello M. Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. Efficient large-scale trace checking using MapReduce. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *38th Int. Conf. on Software Engineering (ICSE 2016)*, pages 888–898. ACM, 2016.
 41. Domenico Bianculli, Carlo Ghezzi, and Srđan Krstić. Trace checking of metric temporal logic with aggregating modalities using MapReduce. In Dimitra Giannakopoulou and Gwen Salaün, editors, *12th Int. Conf. on Software Engineering and Formal Methods (SEFM 2014)*, volume 8702 of *LNCS*, pages 144–158. Springer, 2014.

42. Achim Blumensath and Erich Grädel. Automatic structures. In *15th Annual IEEE Symp. on Logic in Computer Science (LICS 2000)*, pages 51–62. IEEE Computer Society, 2000.
43. Frederik Brix, Christian Fania, Matthieu Gras, Srđan Krstić, and Joshua Schneider. Scalable online monitor. <https://bitbucket.org/krle/scalable-online-monitor>.
44. Daniel Bundala and Joël Ouaknine. On the complexity of temporal-logic path checking. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *41st Int. Colloq. on Automata, Languages, and Programming (ICALP 2014)*, volume 8573 of *LNCS*, pages 86–97. Springer, 2014.
45. Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, 2017.
46. Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams – A new class of data management applications. In *28th VLDB Conf. (VLDB 2002)*, pages 215–226. Morgan Kaufmann, 2002.
47. Jan Chomicki and Damian Niwinski. On the feasibility of checking temporal integrity constraints. *J. Comput. Syst. Sci.*, 51(3):523–535, 1995.
48. E. F. Codd. Relational completeness of data base sublanguages. Technical Report RJ987, IBM Research Laboratory, San Jose, California, 1972.
49. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symp. on Operating System Design and Implementation (OSDI 2004)*, pages 137–150. USENIX Association, 2004.
50. DFINITY Team. The Internet Computer for geeks. Cryptology ePrint Archive, Paper 2022/087, 2022. <https://eprint.iacr.org/2022/087>.
51. Ronald Fagin. Horn clauses and database dependencies. *J. ACM*, 29(4):952–985, 1982.
52. Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 23(2):255–284, 2021.
53. Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods Syst. Des.*, 24(2):101–127, 2004.
54. Adrian Francalanza, Jorge A. Pérez, and César Sánchez. Runtime verification for decentralised and distributed systems. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification: Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 176–210. Springer, 2018.
55. Allen Van Gelder and Rodney W. Topor. Safety and translation of relational calculus queries. *ACM Trans. Database Syst.*, 16(2):235–278, 1991.
56. Matthieu Gras. CPPMon. <https://github.com/matthieugras/cppmon>.
57. Matthieu Gras. StaticMon. <https://github.com/matthieugras/staticmon>.
58. Matthieu Gras. Explicit meets implicit monitoring. Master’s thesis, ETH Zurich, Switzerland, 2022.
59. Florian Haftmann. *Code generation from specifications in higher-order logic*. PhD thesis, Technical University Munich, Germany, 2009.
60. Klaus Havelund and Doron Peled. Runtime verification: From propositional to first-order temporal logic. In Christian Colombo and Martin Leucker, editors, *18th Int. Conf. on Runtime Verification (RV 2018)*, volume 11237 of *LNCS*, pages 90–112. Springer, 2018.

61. Klaus Havelund, Doron Peled, and Dogan Ulus. Dejavu: A monitoring tool for first-order temporal logic. In *3rd Workshop on Monitoring and Testing of Cyber-Physical Systems (MT@CPSWeek 2018)*, pages 12–13. IEEE, 2018.
62. Klaus Havelund, Doron Peled, and Dogan Ulus. First-order temporal logic monitoring with BDDs. *Formal Methods Syst. Des.*, 56(1):1–21, 2020.
63. Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.
64. Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *1st Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1995)*, volume 1019 of *LNCS*, pages 89–110. Springer, 1995.
65. Atefeh Heydari, Mohammadali Tavakoli, and Naomie Salim. Detection of fake opinions using time series. *Expert Syst. Appl.*, 58:83–92, 2016.
66. François Hublet, David Basin, and Srđan Krstić. User-controlled privacy: Taint, track, and control. *Proc. Priv. Enhancing Technol.*, 2024(1), 2024. To appear.
67. François Hublet, David Basin, and Srđan Krstić. Real-time policy enforcement with metric first-order temporal logic. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *27th Eur. Symp. on Research in Computer Security (ESORICS 2022), Part II*, volume 13555 of *LNCS*, pages 211–232. Springer, 2022.
68. Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. In Daniel Leivant, editor, *Int. Workshop on Logic and Computational Complexity (LCC 1994)*, volume 960 of *LNCS*, pages 367–392. Springer, 1994.
69. Srđan Krstić and Dmitriy Traytel. Aerial. <https://bitbucket.org/traytel/aerial>.
70. Lars Kuhlitz and Bernd Finkbeiner. Efficient parallel path checking for linear-time temporal logic with past and bounds. *Log. Methods Comput. Sci.*, 8(4), 2012.
71. Peter Lammich. Refinement of parallel algorithms down to LLVM. In June Andronick and Leonardo de Moura, editors, *13th Int. Conf. on Interactive Theorem Proving (ITP 2022)*, volume 237 of *LIPICs*, pages 24:1–24:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
72. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebraic Methods Program.*, 78(5):293–303, 2009.
73. Leonardo Lima, Andrei Herasimau, Martin Raszyk, Dmitriy Traytel, and Simon Yuan. Explainable online monitoring of metric temporal logic. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *29th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2023), Part II*, volume 13994 of *LNCS*, pages 473–491. Springer, 2023.
74. Jeniffer Lima Graf, Srđan Krstić, and Joshua Schneider. Metric first-order temporal logic with complex data types. In Panagiotis Katsaros and Laura Nenzi, editors, *23rd Int. Conf. on Runtime Verification (RV 2023)*, LNCS. Springer, 2023. To appear.
75. Menna Mostafa and Borzoo Bonakdarpour. Decentralized runtime verification of LTL specifications in distributed systems. In *29th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS 2015)*, pages 494–503. IEEE, 2015.
76. Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In Michael

- Kaminsky and Mike Dahlin, editors, *24th ACM SIGOPS Symp. on Operating Systems Principles (SOSP 2013)*, pages 439–455. ACM, 2013.
77. Jianmo Ni, Jiacheng Li, and Julian J. McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Conf. on Empirical Methods in Natural Language Processing (EMNLP-IJCNLP 2019)*, pages 188–197. Association for Computational Linguistics, 2019. Dataset available at <https://nijianmo.github.io/amazon/index.html>.
 78. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
 79. Mehmet A. Orgun and William W. Wadge. A relational algebra as a query language for temporal DATALOG. In A Min Tjoa and Isidro Ramos, editors, *Int. Conf. on Database and Expert Systems Applications (DEXA 1992)*, pages 276–281. Springer, 1992.
 80. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *14th Int. Symp. on Formal Methods (FM 2006)*, volume 4085 of *LNCS*, pages 573–586. Springer, 2006.
 81. Martin Raszyk. Hydra and Vydra. <https://github.com/mraszyk/hydra>.
 82. Martin Raszyk. MFOTL2RANF. <https://github.com/mraszyk/mfotl2ranf>.
 83. Martin Raszyk. *Efficient, Expressive, and Verified Temporal Query Evaluation*. PhD thesis, ETH Zurich, Switzerland, 2022.
 84. Martin Raszyk, David Basin, Srđan Krstić, and Dmitriy Traytel. Multi-head monitoring of metric temporal logic. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *17th Int. Symp. on Automated Technology for Verification and Analysis (ATVA 2019)*, volume 11781 of *LNCS*, pages 151–170. Springer, 2019.
 85. Martin Raszyk, David Basin, Srđan Krstić, and Dmitriy Traytel. Practical relational calculus query evaluation. In Dan Olteanu and Nils Vortmeier, editors, *25th Int. Conf. on Database Theory (ICDT 2022)*, volume 220 of *LIPICs*, pages 11:1–11:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
 86. Martin Raszyk, David Basin, and Dmitriy Traytel. From nondeterministic to multi-head deterministic finite-state transducers. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th Int. Colloq. on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *LIPICs*, pages 127:1–127:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.
 87. Martin Raszyk, David Basin, and Dmitriy Traytel. Multi-head monitoring of metric dynamic logic. In Dang Van Hung and Oleg Sokolsky, editors, *18th Int. Symp. on Automated Technology for Verification and Analysis (ATVA 2020)*, volume 12302 of *LNCS*, pages 233–250. Springer, 2020.
 88. Lennard Reese, Rafael Castro G. Silva, and Dmitriy Traytel. TimelyMon. <https://git.ku.dk/kfx532/timelymon>.
 89. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Autom. Softw. Eng.*, 12(2):151–197, 2005.
 90. César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srđan Krstić, João M. Lourenço, Dejan Nickovic, Gordon J. Pace, José Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.*, 54(3):279–335, 2019.
 91. Joshua Schneider. HashMon. <https://bitbucket.org/jshs/hashmon>.

92. Joshua Schneider. Randomized first-order monitoring with hashing. In Thao Dang and Volker Stolz, editors, *22nd Int. Conf. on Runtime Verification (RV 2022)*, volume 13498 of *LNCS*, pages 3–24. Springer, 2022.
93. Joshua Schneider. *Scalable and Trustworthy Monitoring*. PhD thesis, ETH Zurich, Switzerland, 2023.
94. Joshua Schneider, David Basin, Frederik Brix, Srđan Krstić, and Dmitriy Traytel. Scalable online first-order monitoring. In Christian Colombo and Martin Leucker, editors, *18th Int. Conf. on Runtime Verification (RV 2018)*, volume 11237 of *LNCS*, pages 353–371. Springer, 2018.
95. Joshua Schneider, David Basin, Frederik Brix, Srđan Krstić, and Dmitriy Traytel. Adaptive online first-order monitoring. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *17th Int. Symp. on Automated Technology for Verification and Analysis (ATVA 2019)*, volume 11781 of *LNCS*, pages 133–150. Springer, 2019.
96. Joshua Schneider, David Basin, Frederik Brix, Srđan Krstić, and Dmitriy Traytel. Scalable online first-order monitoring. *Int. J. Softw. Tools Technol. Transf.*, 23(2):185–208, 2021.
97. Joshua Schneider, David Basin, Srđan Krstić, and Dmitriy Traytel. A formally verified monitor for metric first-order temporal logic. In Bernd Finkbeiner and Leonardo Mariani, editors, *19th Int. Conf. on Runtime Verification (RV 2019)*, volume 11757 of *LNCS*, pages 310–328. Springer, 2019.
98. Scott D. Stoller. Detecting global predicates in distributed systems with clocks. *Distributed Comput.*, 13(2):85–98, 2000.
99. Prasanna Thati and Grigore Roşu. Monitoring algorithms for metric temporal logic specifications. In Klaus Havelund and Grigore Roşu, editors, *4th Workshop on Runtime Verification (RV 2004)*, volume 113 of *Electr. Notes Theor. Comput. Sci.*, pages 145–162. Elsevier, 2004.
100. Alexander Tuzhilin and James Clifford. A temporal relational algebra as basis for temporal relational completeness. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th Int. Conf. on Very Large Data Bases (VLDB 1990)*, pages 13–23. Morgan Kaufmann, 1990.
101. Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the Borealis stream processor. In Karl Aberer, Michael J. Franklin, and Shojiro Nishio, editors, *21st Int. Conf. on Data Engineering (ICDE 2005)*, pages 791–802. IEEE Computer Society, 2005.
102. Sheila Zingg, Srđan Krstić, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. Verified first-order monitoring with recursive rules. In Dana Fisman and Grigore Roşu, editors, *28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022), Part II*, volume 13244 of *LNCS*, pages 236–253. Springer, 2022.