Scalable Online Monitoring of Distributed Systems

David Basin, Matthieu Gras, Srđan Krstić, and Joshua Schneider

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

Abstract. Distributed systems are challenging for runtime verification. Centralized specifications provide a global view of the system, but their semantics requires totally-ordered observations, which are often unavailable in a distributed setting. Scalability is also problematic, especially for online first-order monitors, which must be parallelized in practice to handle high volume, high velocity data streams. We argue that scalable online monitors must ingest events from multiple sources in parallel, and we propose a general model for input to such monitors. Our model only assumes a low-resolution global clock and allows for out-of-order events, which makes it suitable for distributed systems. Based on this model, we extend our existing monitoring framework, which slices a single event stream into independently monitorable substreams. Our new framework now slices multiple event streams in parallel. We prove our extension correct and empirically show that the maximum monitoring latency significantly improves when slicing is a bottleneck.

1 Introduction

Runtime verification (or monitoring) is a technique that verifies systems while they run in their operational environment. It is realized using *monitors*, which are programs that systematically validate a specification by searching for counterexamples in sequences of observations recorded during system execution. *Online monitors* incrementally process the observations, which arrive as an unbounded stream while the system is running [4].

The specification language used significantly influences the monitors' efficiency. Monitors for propositional languages are very efficient and can process millions of observations per second [5, 36, 37]. However, these monitors are limited as they distinguish only a fixed, finite set of observations. The observations are often parameterized by values from (possibly) infinite domains, such as IP addresses and user names. Propositional monitors cannot look for patterns that take such parameters into account. In contrast, *first-order monitors* [10, 15, 30, 31, 38, 39, 42] do not suffer from this limitation, but they must be parallelized to reach the performance of propositional monitors [6, 29, 38–41].

In practice, even small IT systems are often built from many interacting subsystems, which are distributed across multiple machines. When monitored, each subsystem provides information about its behavior as a separate observation sequence. Some approaches adopt specification languages that refer to multiple observation sequences explicitly [21, 33], or whose semantics is defined on partially-ordered observations [35, 43]. However, it is challenging to express global system properties using such decentralized specification languages [25] as they couple the system's behavior with its distributed architecture. Moreover, the specifications must be adapted whenever the system's runtime architecture changes, e.g., when the system is scaled up or down.

We instead focus on *centralized specification languages* [25] that provide a global view of the distributed system. These languages abstract from the system architecture and are thus resilient to its changes. However, centralized specifications often assume totally-ordered observations and without additional information, the multiple observation sequences obtained from distributed systems induce only a partial order. Checking centralized specifications then becomes intractable, since exponentially many compatible total orders must be checked [8]. One therefore needs alternative solutions.

Some approaches opt for a *global clock* to tag every observation across every subsystem with the time when it was made. A global clock abstracts over a collection of local clocks used by each subsystem and synchronized using a clock synchronization protocol like NTP [34]. A clock's resolution is the number of its increments in a time period. The global clock establishes the true total order of observations if the local clocks have sufficient resolutions and are accurate [20] up to a small-enough error. In practice, it is difficult to achieve both conditions for distributed systems that provide observations at high rates [17]. Moreover, even when the observations are totally ordered, they may be received by a monitor in a different order. This can occur if the observations are transmitted over unreliable channels where messages can be delayed, dropped, or reordered [11].

Finally, existing monitors for centralized specifications typically verify a single observation sequence. This *single-source* design limits the monitors' throughput and thus their applicability to the online monitoring of large distributed systems. In previous work, scalable monitors with more than one source have so far been restricted to propositional [14, 18] or decentralized specifications [23, 33] (Section 2).

In this paper we develop a *multi-source* monitoring framework for centralized first-order specifications that takes multiple observation sequences as parallel inputs. It extends our scalable monitoring framework [40, 41], which parallelizes the online monitoring of specifications expressed in Metric First-Order Temporal Logic (MFOTL) [10]. The main idea behind the existing framework is to slice the input stream into multiple substreams (Section 3). Each substream is monitored independently and in parallel by a first-order (sub)monitor, treated as a black box. When instantiated by a concrete submonitor, the framework becomes an online monitor. However, the existing framework supports only a single source, which hampers scalability. It also cannot handle partially-ordered observations, which arise in distributed systems. We address both limitations in this work.

Our new multi-source framework can be used to monitor distributed systems. The framework's topology is independent of the system's topology, and the framework itself can be distributed. The notion of sources abstracts from the nature of the observation's origin. For example, each source could correspond to an independent component of the monitored system, but it may also be the result of aggregating other streams.

We require that all sources have access to a *low-resolution* global clock. Such a clock must have sufficient resolution to decide whether the given specification is satisfied, but it need not necessarily induce a total order on the observations. We argue that global clocks cannot be avoided when monitoring metric specifications as they refer to differences in real time. We account for the fact that observations may have the same creation time (according to the low-resolution clock) and in such cases restrict the specification language to a fragment that guarantees unambiguous verdicts [8]. Our multi-source framework additionally copes with out-of-order observations. This is important even if the sources use reliable channels, as the framework interleaves observations from different sources and its internal components exchange information concurrently.

We generalize the concept of a *temporal structure (TS)*, which models totally-ordered observations, to a *partitioned temporal structure (PTS)*, which represents partially-ordered observations that may be received out-of-order from multiple sources (Section 4.1). We introduce and explain the assumptions on the observation order in a PTS, which are sufficient to uniquely determine whether the specification is satisfied. To monitor a PTS, we add multiple input sources and a reordering step (Section 4.2) to our existing monitoring framework. We prove that this extended framework remains sound and complete: the submonitors collectively find exactly those patterns that exist in the input PTS. We extended the implementation (Section 5) and empirically evaluated it, showing that it significantly improves monitoring performance (Section 6).

In summary, our main contributions are: 1) the definition of the partitioned temporal structure as an input model for multi-source monitors; 2) the extension of our monitoring framework to support multiple sources; 3) its correctness proof, which has been formally verified in the Isabelle proof assistant; and 4) an empirical evaluation showing a significant performance improvement over the single-source framework. Overall, our work lays the foundations for the efficient, scalable, online monitoring of distributed systems using expressive centralized specifications languages like MFOTL.

2 Related Work

Centralized Monitors. Parametric trace slicing [38, 39] performs data slicing on a single input stream to improve monitoring expressivity, rather than its scalability. The stream-based language Lola 2.0 [26] extends parametric trace slicing with dynamic control over the active parameter instances. Lola 2.0 supports multiple input streams, but they must be modeled explicitly in the specification and, moreover, their monitoring is centralized.

Basin et al. [8] monitor distributed systems using a single-source, centralized monitor. They preprocess and merge locally collected traces prior to monitoring. Preprocessing assumes that observations with equal time-stamps happen simultaneously and restricts the specification to a fragment where the order of such observations does not influence the monitor's output. Our approach generalizes this idea, whereby it becomes a special case.

Monitors that handle missing and out-of-order observations [11, 13] are resilient to network failures, which commonly occur in large distributed systems. These centralized monitors, which support MTL and its variant with freeze quantifiers, are orthogonal to our approach and can be instantiated within our monitoring framework.

Decentralized Monitors. Our work builds on top of existing work on parallel black-box monitoring. Basin et al. [6] introduce the concept of slicing temporal structures. They provide composable operators that slice both data and time and support parallel offline monitoring using MapReduce. In prior work [40, 41], we generalized their data slicer and implemented it using the Apache Flink stream processing framework [19].

According to the distributed monitoring survey's terminology [27], the organization of our monitoring framework can be seen as orchestrated or choreographed. In the survey, the notion of a global clock implies the true total observation order, while we assume a low-resolution global clock. Our monitoring framework supports a more expressive specification language than the state-of-the-art alternatives reported on in the survey, which are mostly limited to LTL and the detection of global state predicates.

Bauer and Falcone [14] exploit the locality of the observations in monitored subsystems to organize the monitors hierarchically based on the structure of an LTL formula. In contrast, our parallel monitors each monitor the same (global) formula. By decomposing the specification, Bauer and Falcone reduce the communication overhead, but the monitors still must synchronize on every time-point in the trace. Similarly, El-Hokayem and Falcone [23, 24] propose a framework for decentralised monitoring of LTL and (automata-based) regular specifications. However, they focus only on propositional specifications, which limits the expressiveness of their framework.

Leucker et al. [33] describe a concurrent online monitor for multiple non-synchronized input streams. Unlike our work, the authors assume the existence of a global clock that establishes a total order. It is difficult to compare their specification language TeSSLa with ours. TeSSLa refers to multiple input streams directly, while our specification language specifies (global) properties of distributed systems. It is generally easier to write a centralized specification when observations can originate from multiple streams. In TeSSLA, one must either encode all possible interactions between the streams, or merge the streams first, which offsets any gains from the concurrent evaluation.

Stream Processing. A common mechanism for dealing with out-of-order observations in database and stream processing systems [3] is watermarks [2], which are special markers inserted in the data streams to provide a lower bound on the progress of time. Alternatively, a slack parameter [1] can be specified, which denotes the maximum number of positions that any observation can be delayed at a stream operator. It is used to allocate an appropriately sized buffer for each input of the stream operator to perform reordering. Observations delayed more than the slack value are discarded. Punctuations [45] are more general than watermarks in that they indicate the end of some subset of the stream. The semantics of punctuations can vary, e.g., there will be no more observations having certain attribute values in the stream. Heartbeats [44] resemble watermarks and can be seen as special punctuations about temporal attribute values.

3 Preliminaries

We recap the syntax and semantics of Metric First-Order Temporal Logic [10] and summarize our scalable monitoring framework [40], which slices a single temporal structure.

Metric First-Order Temporal Logic (MFOTL). We fix a set of names $\mathbb{E}$ and for simplicity assume a single infinite domain $\mathbb{D}$ of values. The names $r \in \mathbb{E}$ have associated arities $\iota(r) \in \mathbb{N}$. An *event* $r(d_1, \ldots, d_{\iota(r)})$ is an element of $\mathbb{E} \times \mathbb{D}^*$. We further fix an infinite set $\mathbb{V}$ of variables, such that $\mathbb{V}$, $\mathbb{D}$, and $\mathbb{E}$ are 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 $a < b$. Formulas φ are defined inductively, where t_i, r, x, and I range over $\mathbb{V} \cup \mathbb{D}$, $\mathbb{E}$, $\mathbb{V}$, and $\mathbb{I}$, respectively:

$$\varphi ::= r(t_1, \ldots, t_{\iota(r)}) \mid t_1 \approx t_2 \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \bullet_I \varphi \mid \bigcirc_I \varphi \mid \varphi \, \mathsf{S}_I \, \varphi \mid \varphi \, \mathsf{U}_I \, \varphi.$$

Formulas of the form $r(t_1, \ldots, t_{\iota(r)})$ are called *event formulas*. The temporal operators $\bullet_I$ (previous), $\bigcirc_I$ (next), S_I (since), and U_I (until) may be nested freely. We derive other operators: truth $\top := \exists x.\ x \approx x$, inequality $t_1 \not\approx t_2 := \neg(t_1 \approx t_2)$, conjunction $\varphi \wedge \psi := \neg(\neg\varphi \vee \neg\psi)$, implication $\varphi \rightarrow \psi := \neg\varphi \vee \psi$, eventually $\lozenge_I \varphi := \top \, \mathsf{U}_I \, \varphi$, always $\square_I \varphi := \neg \lozenge_I \neg\varphi$, and once $\blacklozenge_I \varphi := \top \, \mathsf{S}_I \, \varphi$. The set $\mathbb{V}_\varphi$ denotes the set of free variables

$v, i \models r(t_1, \dots, t_n)$	if $r(v(t_1), \dots v(t_n)) \in D_i$	$v, i \models \exists x.\, \varphi$	if $v[x \mapsto d], i \models \varphi$ for some $d \in \mathbb{D}$
$v, i \models t_1 \approx t_2$	if $v(t_1) = v(t_2)$	$v, i \models \bullet_I\, \varphi$	if $i > 0$, $\tau_i - \tau_{i-1} \in I$, and $v, i-1 \models \varphi$
$v, i \models \neg\varphi$	if $v, i \not\models \varphi$	$v, i \models \bigcirc_I\, \varphi$	if $\tau_{i+1} - \tau_i \in I$ and $v, i+1 \models \varphi$
$v, i \models \varphi \vee \psi$	if $v, i \models \varphi$ or $v, i \models \psi$		
$v, i \models \varphi\, \mathsf{S}_I\, \psi$	if $v, j \models \psi$ for some $j \leq i$, $\tau_i - \tau_j \in I$, and $v, k \models \varphi$ for all k with $j < k \leq i$		
$v, i \models \varphi\, \mathsf{U}_I\, \psi$	if $v, j \models \psi$ for some $j \geq i$, $\tau_j - \tau_i \in I$, and $v, k \models \varphi$ for all k with $i \leq k < j$		

Fig. 1. Semantics of MFOTL

of φ. A formula has *bounded future* iff all subformulas of the form $\bigcirc_{[a,b)}\, \alpha$ and $\alpha\, \mathsf{U}_{[a,b)}\, \beta$ (including derived operators) satisfy $b < \infty$.

MFOTL formulas are interpreted over *temporal structures (TS)*, which model totally-ordered observation sequences (or streams). A temporal structure ρ is an infinite sequence $(\tau_i, D_i)_{i \in \mathbb{N}}$, where $\tau_i \in \mathbb{N}$ is a discrete time-stamp, and the *database* $D_i \in \mathbb{DB} = \mathcal{P}(\mathbb{E} \times \mathbb{D}^*)$ is a finite set of events that happen concurrently in the monitored system. Databases at different time-points $i \neq j$ may have the same time-stamp $\tau_i = \tau_j$. The sequence of time-stamps must be *monotone* ($\forall i.\ \tau_i \leq \tau_{i+1}$) and *progressing* ($\forall \tau.\ \exists i.\ \tau < \tau_i$).

The relation $v, i \models_\rho \varphi$ defines the satisfaction of the formula φ for a valuation v at an index i with respect to the temporal structure $\rho = (\tau_i, D_i)_{i \in \mathbb{N}}$; see Fig. 1. Whenever ρ is fixed and clear from the context, we omit the subscript on $\models$. The valuation v is a mapping $\mathbb{V}_\varphi \to \mathbb{D}$, assigning domain elements to the free variables of φ. Overloading notation, v is also the extension of v to the domain $\mathbb{V}_\varphi \cup \mathbb{D}$, setting $v(t) = t$ whenever $t \in \mathbb{D}$. We write $v[x \mapsto d]$ for the function equal to v, except that the argument x is mapped to d.

Monitors. An *online monitor* for a formula φ receives time-stamped databases that are a finite prefix π of some TS ρ (denoted by $\pi \prec \rho$). The monitor incrementally computes a verdict, which is a set of valuations and time-points that satisfy φ given π. (Typically, one is interested in the violations of a specification $\Box\, \psi$, which can be obtained by monitoring $\neg\psi$ instead.) A monitor is *sound* if the verdict for π contains (v, i) only if $v, i \models_\rho \varphi$ for all $\rho \succ \pi$. It is *complete* if whenever $\pi \prec \rho$ is such that $v, i \models_{\rho'} \varphi$ for all $\rho' \succ \pi$, then there is another prefix $\pi' \prec \rho$ for which the verdict contains (v, i). In our formal treatment, we consider the monitor's output in the limit as the input prefix grows to infinity. Thus, a monitor implements an abstract *monitor function* $\mathcal{M}_\varphi : (\mathbb{N} \times \mathbb{DB})^\omega \to \mathcal{P}((\mathbb{V}_\varphi \to \mathbb{D}) \times \mathbb{N})$ that maps a TS ρ to the union of all verdicts obtained from all possible prefixes of ρ. We shall assume that the monitor implementing $\mathcal{M}_\varphi$ is sound and complete. If φ has bounded future, it follows that $\mathcal{M}_\varphi(\rho) = \{(v, i) \mid v, i \models_\rho \varphi\}$.

Slicing Framework. In prior work, we parallelized online first-order monitoring by slicing [40, 41] the temporal structure into N temporal structures that can be independently monitored. Figure 2 shows the dataflow graph constructed by our monitoring framework to monitor a given formula φ. The framework utilizes N parallel submonitors, which are independent instances of the monitor function $\mathcal{M}_\varphi$. Let $[n]$ denote the set $\{1, \dots, n\}$. The slicer $\mathcal{S}_g$ is parameterized by a *slicing strategy* $g : [N] \to \mathcal{P}(\mathbb{V}_\varphi \to \mathbb{D})$ satisfying $\bigcup_{k \in [N]} g(k) = (\mathbb{V}_\varphi \to \mathbb{D})$. The slicing strategy specifies the set of valuations $g(k)$ for which the submonitor k is responsible. Next, we describe which events the submonitor k receives to evaluate φ correctly on all $v \in g(k)$. Given an event e, let *sfmatches*(φ, e) be the set of all valuations v for which there is an event subformula ψ in φ with $v(\psi) = e$. (Here v is extended to event subformulas, such that $v(r(t_1, \dots, t_{\iota(r)})) = r(v(t_1), \dots, v(t_{\iota(r)}))$,

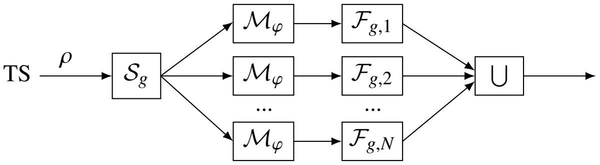

Fig. 2. Dataflow in the single-source monitoring framework

and we assume that φ's bound variables are disjoint from its free variables.) For a database D and a set of valuations R, we write $D \downarrow R$ for the restricted database $\{e \in D \mid \mathit{sfmatches}(\varphi, e) \cap R \neq \varnothing\}$. The same notation restricts the TS $\rho = (\tau_i, D_i)_{i \in \mathbb{N}}$ pointwise, i.e., $\rho \downarrow R = (\tau_i, D_i \downarrow R)_{i \in \mathbb{N}}$. Then, it is sufficient if the submonitor k receives the slice $\mathcal{S}_{g,k}(\rho) = \rho \downarrow g(k)$. The slicer $\mathcal{S}_g$ thus outputs N streams $\mathcal{S}_{g,1}(\rho), \ldots, \mathcal{S}_{g,N}(\rho)$.

The output of the monitor function $\mathcal{M}_\varphi$ on ρ can be reconstructed from the parallel submonitors' output on the N slices. Formally, $\mathcal{M}_\varphi(\rho) = \bigcup_{k \in [N]} (\mathcal{F}_{g,k}(\mathcal{M}_\varphi(\mathcal{S}_{g,k}(\rho))))$, where $\mathcal{F}_{g,k}(X) = X \cap (g(k) \times \mathbb{N})$. Note that Figure 2 illustrates the right-hand side of the equation defining $\mathcal{M}_\varphi(\rho)$. In [40], we proved this equation assuming a stronger completeness property of the online monitor. However, it can also be shown for the abstract function $\mathcal{M}_\varphi$, which operates on a TS. The intersection with $g(k) \times \mathbb{N}$ is needed to avoid spurious verdicts for some formulas, such as those involving equality.

Example 1. Consider an access control policy for a service operating on medical records, where *whenever a user requests to process a record, the service does so only if the user was authorized to access that record.* The policy is formalized in MFOTL as $\Box\, \Phi_1$ with $\Phi_1 \equiv \forall u.\ \mathsf{proc}(u,r) \rightarrow \blacklozenge\, \mathsf{auth}(u,r)$. The formula $\mathsf{proc}(u,r)$ denotes that u requested to process r and $\mathsf{auth}(u,r)$ denotes that u is authorized to access r. For the sake of this example, we leave r as the only free variable and assume numeric identifiers for u and r.

We monitor $\varphi \equiv \neg\Phi_1$ as shown in Figure 2, using the slicing strategy $g(k) = \{v \mid v(r) \bmod 3 = k-1\}$ with $N = 3$ slices. Recall that the set $g(k)$ contains valuations, which are mappings from the free variables $\{r\}$ to $\mathbb{D}$. The TS ρ models a service execution with the first database $D = \{\mathsf{auth}(1,1), \mathsf{auth}(1,2), \mathsf{auth}(1,3), \mathsf{proc}(1,3), \mathsf{proc}(1,4)\}$. The submonitor 1 receives $D \downarrow g(1) = \{\mathsf{auth}(1,1), \mathsf{proc}(1,4)\}$ as its first database and reports the verdict $\{(\{r \mapsto 4\}, 0)\}$ as a violation of Φ_1, which is the only violation evident from D. Submonitors 2 and 3 receive databases $\{\mathsf{auth}(1,2)\}$ and $\{\mathsf{auth}(1,3), \mathsf{proc}(1,3)\}$ and output empty verdicts after processing them, respectively.

Example 2. Now consider a centralized system running many instances of the service from the previous example. Each service handles user requests either by directly processing a record, or by recursively triggering requests to other (local) services. So, now *a service is allowed to process a record only if this was initially requested by a user authorized to access the record.* Notice that data processing can now happen after a chain of requests involving multiple services. Therefore, we assume that the services attach a unique session number s to all requests caused directly or indirectly by a user's request.

The MFOTL formula $\Box\, \Phi_2$ with $\Phi_2 \equiv (\blacklozenge\, \mathsf{req}(u,s)) \wedge \mathsf{proc}(s,r) \rightarrow \blacklozenge\, \mathsf{auth}(u,r)$ formalizes the new specification, now with free variables $\{u,s,r\}$. The new event formulas are $\mathsf{req}(u,s)$ (user u sends a request, starting a session s), and $\mathsf{proc}(s,r)$ (record r is processed within the session s). Let $N = 8$ and $g(k) = \{v \mid 4 \cdot (v(u) \bmod 2) + 2 \cdot (v(s) \bmod 2) + v(r) \bmod 2 = k-1\}$ be a slicing strategy. Note that according to g, submonitor 1 receives only valuations where each variable has an even value. When we monitor ρ with the first database $D = \{\mathsf{req}(2,2), \mathsf{auth}(2,1), \mathsf{proc}(2,2)\}$, each event in D is sent to two submonitors. For instance, $\mathsf{req}(2,2)$ is sent both to submonitors 1 and 2, whereas $\mathsf{proc}(2,2)$ is sent to submonitors 1 and 5. Such a slicing scheme ensures that submonitor 1 receives sufficient information to output the verdict $\{(\{u \mapsto 2, s \mapsto 2, r \mapsto 2\}, 0)\}$.

4 Monitoring Distributed Systems

We consider the online monitoring of a distributed system. A first problem that we must solve is the lack of a total order on the observations of the individual subsystems (machines, processes, threads, etc.). As explained in the introduction, such a total order is required by the semantics of centralized specifications, but it does not exist unless the subsystems' execution is perfectly synchronized. This cannot be assumed in general as one usually desires some parallelism in distributed systems.

A second problem is that distributed systems are often developed to achieve scalability, and online monitors used with such systems should be scalable as well. A monitor that physically combines observations from different sources into a single stream cannot satisfy this requirement: if the workload increases and additional events are generated, the processes working with the single stream will eventually be overloaded. Scalable online monitors must therefore ingest observations in parallel.

We solve the above problems by viewing online monitoring as an instance of distributed stream processing. Observations enter the monitor in multiple parallel streams, called *sources*. We give a general model of sources that captures a variety of distributed monitoring scenarios, while still allowing the efficient monitoring of metric specifications (Section 4.1). The model logically decouples the monitor from the monitored system, which ensures that the system's topology can be chosen independently. We then extend the slicing framework to utilize multiple sources (Section 4.2). The resulting multi-source framework does not require a total order on the observations, and it scales better than the single-source version, even if the monitored system is not truly distributed.

4.1 Input Model

We model the monitor's input as a *Partitioned Temporal Structure (PTS)*, which we define formally later in this section. We believe that this model is useful beyond our implementation using slicing. The model is based on several assumptions about the problem at hand. Below, we explain these assumptions, show how they are reflected in the PTS, and give examples of possible applications.

Assumption 1. We assume that the monitored specification has the form $\Phi = \Box\neg\varphi$, where φ has bounded future. We also assume that the specification is centralized, i.e., its event formulas are interpreted over all events from the entire system.

The restriction on the formula's structure is common for first-order monitors. It guarantees that every violation can be detected in finite time [10]. The assumption that the specification is centralized rules out various monitoring approaches. We already argued that centralized monitors are ill-suited for scalable distributed systems. Moreover, note that centralized specifications cannot be easily split into smaller parts that are handled by local monitors, as illustrated by the following example.

Example 3. Consider now the services from Example 2 deployed on a microservice architecture that is running on a cluster of machines. Each service generates its own TS. As requests span arbitrarily many machines, the specification cannot be checked locally.

We therefore treat the monitored system and the monitor as independent entities. They are connected by M sources, which are parallel observation streams. The sources may correspond the monitored system's components, e.g., the services in Example 3. This is not required by the model, which we will show in a later example.

The next assumption imposes an important restriction: it must be possible to arrive at a definite monitoring verdict even if the observations are only partially ordered. Otherwise, we would need to construct all possible interleavings of the concurrent observations, which is generally infeasible. We avoid relying on system-specific information, such as vector clocks, to reduce the number of interleavings [35] as this would diminish the generality of our approach.

Assumption 2. There exists a TS $\rho^\star$ that describes the actual sequence of events as they occur in real time. The time-stamps in $\rho^\star$ are obtained from the real time truncated to the precision used in the specification. (We do not assume that $\rho^\star$ can be observed directly.) The sources must have access to a global clock that provides time-stamps from $\rho^\star$ as well as sufficient information about the event order to decide whether $\rho^\star$ satisfies Φ.

Note that the system satisfies the specification Φ iff $\rho^\star$ satisfies Φ. We model the information provided by the global clock using *indices*, which are natural numbers attached to every observation. If the index of observation o_1 is less than the index of observation o_2, then o_1 must have happened before o_2. At one extreme, the index is simply the position of the observation in $\rho^\star$, i.e., a global sequence number. Then every specification has a definite verdict. A distributed system providing such sequence numbers would need a global clock with very high resolution, which is often unrealistic. However, centralized applications, which have access to sequence numbers, can be more efficiently monitored with a multi-source monitor than with a single-source monitor.

Example 4. Kernel event tracing creates streams with high event rates [22]. We may improve the monitor's throughput by distributing the events over multiple streams (see Section 6). For a single processor, its hardware counters provide global sequence numbers.

At the other extreme, the indices could simply be the time-stamps. We say that a clock providing such indices is *low resolution*, as its resolution may not be high enough to establish the true total order. Yet not all specifications can be monitored if the indices have lower resolution than global sequence numbers. We follow the *collapse* approach by Basin et al. [8], where events with the same time-stamp are collapsed into a single instantaneous observation. We generalize the collapse from time-stamps to indices, which unifies the presentation. We then rephrase the requirement on the global clock from Assumption 2 in terms of the collapse: monitoring the collapsed sequence must result in essentially the same output as monitoring $\rho^\star$. To make this precise, we add the indices to $\rho^\star$ itself, which results in the indexed temporal structure $\hat{\rho}^\star$.

Definition 1. *An* indexed temporal structure (ITS) *is a TS over extended tuples* (α_i, τ_i, D_i), *where* $\alpha_i \in \mathbb{N}$ *are indices. The indices must increase monotonically* ($\forall i.\ \alpha_i \leq \alpha_{i+1}$)*, and they must refine time-stamps* ($\forall i.\ \forall j.\ \alpha_i \leq \alpha_j \Longrightarrow \tau_i \leq \tau_j$).

Definition 2. *The* generalized collapse $\mathcal{C}(\hat\rho) = (\tau^c_i, D^c_i)_i$ *of an ITS $\hat\rho$ is characterized by the unique monotone and surjective function $f : \mathbb{N} \to \mathbb{N}$ that maps (only) positions with the same index to a common value ($\forall i.\ \forall j.\ \alpha_i = \alpha_j \iff f(i) = f(j)$). Then $\forall i.\ \tau^c_{f(i)} = \tau_i$ and $\forall j.\ D^c_j = \bigcup\{D_i \mid f(i) = j\}$.*

Since $\hat\rho^\star$ is the idealized totally-ordered sequence, its indices must increase monotonically. Indices must also refine time-stamps so that the generalized collapse is a TS. This requirement, which may seem quite strong, is necessary because the semantics of a metric specification language (like MFOTL) is defined with respect to a TS. Note, however, that the resolution of time-stamps is not fixed (Assumption 2). The resolution of time-stamps and thus indices can be quite low as long as it is possible to formalize the specification faithfully.

Definition 3. *We call $\hat\rho$* adequate *for the formula φ iff $v, i \models_{\mathcal{C}(\hat\rho)} \varphi \iff (\exists j.\ f(j) = i \wedge v, j \models_\rho \varphi)$ for all v and i, where ρ is obtained from $\hat\rho$ by omitting the indices.*

Monitoring a formula φ on the generalized collapse of an adequate ITS finds the same satisfying valuations as monitoring the ITS itself (modulo the remapping of time-points).

Lemma 1. *Suppose that $\hat\rho$ is adequate for the formula φ. Then $\mathcal{M}_\varphi(\mathcal{C}(\hat\rho)) = \{(v, f(j)) \mid (v, j) \in \mathcal{M}_\varphi(\rho)\}$, where f is as in Definition 2.*

If the indices of an ITS are global sequence numbers (e.g., $\forall i.\ \alpha_i = i$), the ITS is adequate for all φ. To gain intuition for other ITS, we focus again on the case where indices are time-stamps (*time-ITS*, $\forall i.\ \alpha_i = \tau_i$). Basin et al. [8] define the notion of *collapse-sufficient* formulas, which are essentially those formulas that can be monitored correctly on a time-based collapse. They provide an efficiently decidable fragment of formulas with this property. (More precisely, a time-ITS $\hat\rho$ is adequate for φ iff φ satisfies the properties ($\models\exists$) and ($\not\models\forall$) given in [8], which implies that $\Phi = \Box\neg\varphi$ is collapse-sufficient.) Often, a formula can be made collapse-sufficient by replacing subformulas $\blacklozenge_{[0,t]}\,\alpha$ (note the interval's zero bound) with $\blacklozenge_{[0,t]}\lozenge_{[0,0]}\,\alpha$, and dually for $\lozenge_{[0,t]}$. More complicated replacements are however needed for S and U.

Example 5. To obtain a collapse-sufficient formula from the specification in Example 2, we restrict the authorizations to happen at least one second before their use. Furthermore, we ignore the order of requests and process events (using the $\lozenge_{[0,0]}$ operator) as long as they have the same time-stamp. The specification is formalized as $\Box\,\Phi_3$ with $\Phi_3 \equiv (\blacklozenge\lozenge_{[0,0]}\,\mathsf{req}(u,s)) \wedge \mathsf{use}(s,d) \to \blacklozenge_{[1,60]}\,\mathsf{auth}(u,d)$.

It is common practice in distributed systems to process, aggregate, and store logging information in a dedicated service. The observations fed to the monitor are then taken from this service. In Example 3, the microservices could first send their events to a distributed message broker such as Kafka [32]. As a result, events from different services may be interleaved before they reach the monitor. We therefore allow that individual sources provide observations in a different order than their temporal order. This generalization adds almost no complexity to the monitor's design (Section 4.2): we must anyway reorder the observations, even for correctly ordered streams, to synchronize them across sources. Handling out-of-order observations thus comes almost for free.

Assumption 3. Sources may provide observations in any order. However, the delay of each observation must be bounded.

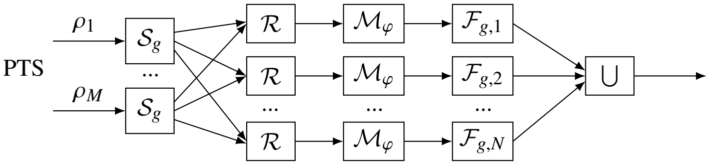

Fig. 3. Dataflow in the multi-source monitoring framework

The latter condition ensures that the monitor does not get stuck. We enforce it by adding watermarks, which are lower bounds on future indices, to the sources. Then, the observations' delay is bounded if the watermarks always eventually increase. In our implementation, watermarks are interspersed between regular observations. We simplify the formal definitions below by assuming that every database has an associated watermark, which is the one most recently seen. Note that an input model with watermarks is strictly more permissive than one without. If we know that the observations will be in the correct order, we can simply set each watermark equal to the next index.

We are now ready to give a formal definition of our input model. We recall the main idea: The monitor's input is a PTS, which partitions some ITS $\hat{\rho}^\star$ into multiple sources. If $\hat{\rho}^\star$ is adequate for the formula φ, it suffices to monitor the generalized collapse $\mathcal{C}(\hat{\rho}^\star)$ via the PTS to achieve the goal of monitoring $\rho^\star$.

Definition 4. *A* partitioned temporal structure (PTS) *is a finite list $\rho_1,\ldots,\rho_M$ of $M \geq 1$* sources. *A source ρ_k is an infinite sequence of tuples $(\alpha_{k,i},\beta_{k,i},\tau_{k,i},D_{k,i})_{i\in\mathbb{N}}$, where $\alpha_{k,i} \in \mathbb{N}$ is an index, $\beta_{k,i} \in \mathbb{N}$ is a watermark, and $\tau_{k,i}$ and $D_{k,i}$ are as in temporal structures. For all $k \in [M]$, ρ_k must satisfy (P1) monotone watermarks ($\forall i.\, \beta_{k,i} \leq \beta_{k,i+1}$); (P2) progressing watermarks ($\forall \beta.\, \exists i.\, \beta \leq \beta_{k,i}$); (P3) watermarks bound future indices ($\forall i.\, \forall j.\, i < j \Longrightarrow \beta_{k,i} \leq \alpha_{k,j}$); and (P4) progressing time-stamps ($\forall \tau.\, \exists i.\, \tau \leq \tau_{k,i}$).*

A PTS $\rho_1,\ldots,\rho_n$ partitions *an ITS $(\alpha_j,\tau_j,D_j)_{j\in\mathbb{N}}$ iff it is (Q1) sound ($\forall k.\, \forall i.\, \exists j.\, \alpha_{k,i} = \alpha_j \wedge \tau_{k,i} = \tau_j \wedge D_{k,i} \subseteq D_j$); and (Q2) complete wrt. indices ($\forall j.\, \exists k.\, \exists i.\, \alpha_{k,i} = \alpha_j \wedge \tau_{k,i} = \tau_j$) and events ($\forall j.\, \forall e \in D_j.\, \exists k.\, \exists i.\, \alpha_{k,i} = \alpha_j \wedge \tau_{k,i} = \tau_j \wedge e \in D_{k,i}$).*

Conditions P1–P3 have already been explained, while condition P4 is inherited from temporal structures. Conditions Q1–Q2 encode that the PTS contains the same information as the ITS. Specifically, the sources must have access to a low-resolution global clock providing the time-stamps in $\hat{\rho}^\star$. Its resolution is defined by the specification. For instance, we could use NTP-synchronized time in seconds in Example 5, where the specification requires recent authorization on the order of seconds.

We need both completeness wrt. indices and events (Q2) because the latter is trivially true for empty databases, but we must ensure that the corresponding index (and time-stamp) occurs in the PTS. Note that for every ITS, there is at least one PTS that partitions it into $M \geq 1$ sources: let $(\alpha_{k,i},\beta_{k,i},\tau_{k,i},D_{k,i}) = (\alpha_j,\alpha_j,\tau_j,D_j)$ with $j = i \cdot M + k - 1$.

4.2 Slicing Framework with Multiple Sources

Figure 3 shows the slicing framework's dataflow after extending it to multiple sources. Arrows represent streams of elements, and rectangles are stream transducers with possibly multiple inputs and outputs. The input consists of the M sources of a PTS. We apply the slicer $\mathcal{S}_g$ independently to each source, using the given slicing strategy g. The input of $\mathcal{S}_g$ thus carries additional indices and watermarks. Since slicing only affects the databases, we can easily lift it to source streams. Let the stream $\rho_{k,k'}$ be the output of the kth slicer on its k'th outgoing edge, where $k' \in [N]$. The k'th instance of $\mathcal{R}$ (described below) receives an interleaving of the streams $\rho_{1,k'}, \ldots, \rho_{M,k'}$. Stream processor implementations usually do not guarantee a particular order for such an interleaving. This also applies to our implementation (Section 5). Therefore, we assume that the interleaving is nondeterministic, with the only guarantee being one of fairness, namely that every input stream is visited infinitely often. We further assume that the elements in the streams $\rho_{k,k'}$ are tagged with their origin k.

The crucial new component is the *reordering algorithm* $\mathcal{R}$, which fulfills two purposes. First, it collapses databases according to their indices. This has an effect only if indices are time-stamps, i.e., the underlying ITS is time-indexed. Second, $\mathcal{R}$ ensures that the input to the monitor function $\mathcal{M}_\varphi$ is sorted correctly. Even if observations arrive in the correct order at PTS sources, reordering is necessary due to the shuffling between $\mathcal{S}_g$ and $\mathcal{R}$.

The pseudocode for $\mathcal{R}$ is given in Algorithm 1. It uses two global variables, *marks* and *buffer*, both finite associative maps. The expression $\mathrm{keys}(m)$ denotes the set of keys in the associative map m. If $x \in \mathrm{keys}(m)$, then $m[x]$ is the unique value that m associates with x. The map *marks* stores the largest watermark seen so far for each input stream. (Recall that the input to $\mathcal{R}$ is an interleaving of one slice from each input stream.) The map *buffer* maps indices to pairs of time-stamps and databases. Intuitively, *buffer* keeps all indices that may occur in the future as the watermarks have not advanced past them.

The procedure INITIALIZE(M) is called once when the monitor starts, where M is the number of sources. The watermarks are initially zero, which is a lower bound for all indices. The procedure PROCESS(x) is called for every stream element x received by $\mathcal{R}$. The first element of the tuple $x = (k, \alpha, \beta, \tau, D)$ identifies the source from which it originates, while the remaining elements are from the sliced PTS. Line 4 restores the invariant for *marks*. In lines 5–9, D's contents are added to the buffer. If the index α is already mapped by *buffer*, we take the union with the previously stored database to implement the collapse. Otherwise, τ and D are inserted into *buffer*. The value θ computed in line 10 is the minimum of all the latest watermarks across all inputs. By condition P3 of PTS (Definition 4), we know that all future indices that $\mathcal{R}$ will receive must be at least θ. Therefore, it is safe (only) to output everything in *buffer* with a smaller index. This happens in lines 11–13. Note that we iterate over the indices in ascending order, which ensures that the output is sorted correctly. The sequence of $\mathcal{R}$'s output elements (which are pairs of time-stamps and databases) forms the stream that is sent to the monitor $\mathcal{M}_\varphi$ in Figure 3.

The following theorem establishes the correctness of the multi-source framework. It is formalized [7] and verified along with Lemma 1 in the Isabelle/HOL proof assistant.

Theorem 1. *Let $\rho_1, \ldots, \rho_M$ be a PTS that partitions $\hat{\rho}^\star$. For all slicing strategies g, the result of the dataflow in Figure 3 (with inputs $\rho_1, \ldots, \rho_M$) is equal to $\mathcal{M}_\varphi(\mathcal{C}(\hat{\rho}^\star))$.*

Note that this theorem holds for all possible partitions of $\hat{\rho}^\star$ and all possible interleavings that can result from the shuffling step. However, it is only a statement about the infinite sequence of verdicts. Each verdict may be delayed by an arbitrary (but finite) amout of time, depending on the watermarks in the input and the shuffling implementation. Theorem 1 does not assume that $\hat{\rho}^\star$ is adequate for φ because it refers directly to the generalized

Algorithm 1 Reordering algorithm $\mathcal{R}$

```
 1: procedure INITIALIZE($M$)
 2:     $marks \leftarrow \{k \mapsto 0 \mid k \in [M]\}$,    $buffer \leftarrow \{\}$
 3: procedure PROCESS($(k, \alpha, \beta, \tau, D)$)
 4:     $marks[k] \leftarrow \beta$
 5:     if $\alpha \in \mathrm{keys}(buffer)$ then
 6:         $(\tau', D') := buffer[\alpha]$
 7:         $buffer[\alpha] \leftarrow (\tau', D \cup D')$
 8:     else
 9:         $buffer[\alpha] \leftarrow (\tau, D)$
10:     $\theta := \min\{marks[k] \mid k \in \mathrm{keys}(marks)\}$
11:     for $i \in \mathrm{keys}(buffer)$ in ascending order, while $i < \theta$ do
12:         output $buffer[i]$
13:         delete $i$ from $buffer$
```

collapse $\mathcal{C}(\hat{\rho}^\star)$. If we additionally know that $\hat{\rho}^\star$ is adequate, we get the same verdicts as if we were monitoring $\rho^\star$ directly, modulo the mapping of time-points (Lemma 1).

Example 6. We use the multi-source monitoring framework to monitor $\varphi \equiv \neg \Phi_3$ (Example 5) on $M = 2$ distributed services (Example 3), using $N = 8$ submonitors and the splitting strategy g (Example 2). The dataflow is shown in Figure 3. The input PTS consists of two sources ρ_1 and ρ_2 with prefixes $(0,0,0,\{\mathsf{req}(2,2)\})$, $(3,0,3,\{\mathsf{proc}(1,1)\})$, $(1,0,1,\{\mathsf{req}(2,1)\})$, $(4,4,4,\{\})$ and $(0,0,0,\{\mathsf{proc}(2,2), \mathsf{auth}(2,1)\})$, $(4,4,4,\{\})$, respectively. Note that the indices are equal to the time-stamps. As in Example 2, submonitor 1 receives events $\mathsf{req}(2,2)$ and $\mathsf{proc}(2,2)$ and produces the same verdict. However, the reordering algorithm sends these events only after receiving watermark 4 from both sources. All of the remaining events are sent to submonitor 3. The reordering algorithm ensures that they are received in the order defined by their indices. Hence, $\mathsf{auth}(2,1)$ is received first, followed by $\mathsf{req}(2,1)$, and then by $\mathsf{proc}(1,1)$. Due to the reordering, submonitor 3 correctly produces an empty verdict for the given prefixes.

We conclude with a remark about the time and space complexity of Algorithm 1. Both are unbounded in the worst case because of the problem with unbounded watermark delays mentioned above. However, we obtain a more meaningful result under reasonable additional assumptions. For example, assume that each database in the input has size at most d, that every index occurs at most c times, and that the number of stream elements between an index α and the time that θ (line 10) becomes greater than α is at most z. The parameter c is upper bounded by the *time-point rate* (Section 6) multiplied by M. The parameter z depends on the *watermark frequency* and the *maximum (event) delay* (Section 6), and also on the additional delay introduced by the shuffle step between slicing and reordering.

There are at most z different keys in *buffer* at any given time, each mapping to a database of size at most $c \cdot d$. The space complexity is thus $O(M + c \cdot d \cdot z)$ in the uniform RAM model, where M is the number of sources. By using a self-balancing search tree for *buffer* and hash tables for the databases contained therein, one invocation of PROCESS has an amortized average time complexity of $O(M + d + \log z)$, again in the uniform model. The summand M can be reduced to $\log M$ by using a binary heap to maintain θ instead of recomputing it in every invocation.

5 Implementation

We implemented a multi-source online monitoring framework based on the ideas outlined in Section 4. It extends our previous single-source framework [40,41] and is available online [7]. The implementation instantiates the submonitors with MonPoly [12], which supports a monitorable fragment of MFOTL [10] where, in particular, formulas must have bounded future. We modified about 4k lines of code (3.2k added and 0.8k deleted). In Section 4, we omitted many details, e.g., how events are delivered to and exchanged within the framework, which effect the efficiency and usability of the framework. We explain some implementation choices here and further details can be found in [28,40].

Our multi-source framework is built on top of Apache Flink [19], which provides an API and a runtime for fault tolerant distributed stream processing. Fault tolerance is important for distributed online monitors since increasing the number of machines on which a monitor runs also increases the risk of failures, which would otherwise disrupt the monitor's operation. The implementation's dataflow corresponds roughly to the dataflow in Figure 3, except that the streams' elements are individual events instead of databases. The events are interleaved with other control elements that carry additional metadata. We use Flink's API to define the logical dataflow graph, whose vertices are operators that transform potentially unbounded data streams. At runtime, operators can have multiple instances as defined by their degree of parallelism. Each operator instance works on a partition, i.e., a substream. Stream elements are repartitioned according to some strategy if the degree of parallelism changes from one operator to the next. In Figure 3, the parallelism changes from M to N at the shuffling step. Each slicer outputs is a single stream of elements labeled with their destination submonitor. Based on these labels, a stream partitioner ensures that the elements reach their intended destination.

We use two types of source operators (TCP and Kafka) with different trade-offs. In Flink, sources are operators without incoming edges in the dataflow graph. Their degree of parallelism, which must be chosen before execution starts, determines the number M of input streams. The TCP source reads simple text streams from multiple sockets by connecting to a list of address and port pairs. It is fast and thus useful for benchmarking the other components, but it is not fault tolerant. The Kafka [32] source operator implements a distributed persistent message queue and provides fault tolerance. However, we exclude it from the evaluation as it incurred a significant overhead in our preliminary experiments.

The slicer, submonitors, filtering, and verdict union are nearly unmodified (see [40]). However, there are now multiple instances of the slicing operator. The reordering function $\mathcal{R}$ is a straightforward implementation of Algorithm 1. In our implementation, the *buffer* is simply a hash table, and we access it by probing for increasing indices. A more efficient approach can be used if this becomes a bottleneck. Our implementation currently supports time-points and time-stamps as indices (see Section 4.1). With out-of-order input, only time-stamps are supported, but it should be easy to generalize the framework to time-points. We rely on *order elements*, which are a type of control elements, instead of associating watermarks with every database. For in-order inputs, the order elements are separators between databases, which are inserted by the input parser. In this case, we can synthesize the watermark from the database's time-point or time-stamp. If the input is out-of-order, watermarks must be provided as annotations in the input data. The input parser extracts the watermarks and embeds them in newly created order elements.

$$
\begin{aligned}
\varphi_s &\equiv P(x,y) \land \big((\blacklozenge_{[1,3s]}\, Q(x,z)) \land \blacklozenge_{[1,3s]}\, R(x,w)\big)\\
\varphi_i &\equiv \mathit{insert}(u,\mathsf{db1},p,d) \land d \not\approx \mathsf{null} \land \neg \lozenge_{[0,30h]} (\exists u'.\, \mathit{insert}(u',\mathsf{db2},p,d) \lor \mathit{delete}(u',\mathsf{db1},p,d))\\
\varphi_d &\equiv \Big(\big(\mathit{delete}(u,\mathsf{db1},p,d) \land d \not\approx \mathsf{null} \land \neg \blacklozenge_{[0,30h]}\, \exists u'.\, \exists p'.\, \mathit{insert}(u',\mathsf{db1},p',d)\big) \lor\\
&\qquad \big(\mathit{delete}(u,\mathsf{db1},p,d) \land d \not\approx \mathsf{null} \land (\exists u'.\, \exists p'.\, (\blacklozenge_{[0,30h]}\, \mathit{insert}(u',\mathsf{db1},p',d)) \lor\\
&\qquad\quad (\lozenge_{[0,30h]}\, \mathit{insert}(u',\mathsf{db2},p',d))))\Big) \land \neg \lozenge_{[0,30h]}\, \exists u'.\, \exists p'.\, \mathit{delete}(u',\mathsf{db2},p',d)
\end{aligned}
$$

Fig. 4. MFOTL formulas used in the evaluation

6 Evaluation

To assess the scalability of our extended framework we organized our evaluation (available online [7]) in terms of the following research questions (RQs).

RQ1: How do the input parameters affect the multi-source framework's scalability?
RQ2: What is the impact of imbalanced input sources on performance?
RQ3: Can multiple sources be used to improve monitoring performance?
RQ4: How much overhead does event reordering incur?

RQ1 and RQ2 assess the impact of input parameters (specifically, the event rate and time-point rate, defined below, as well as the number of inputs and submonitors) on our framework's performance. When events arrive out of order, we additionally control their maximum delay and the watermark frequency. We assess RQ1 by monitoring multiple traces with the same event rate, while for RQ2 we relax this restriction. RQ3 aims to evaluate the overall performance gain of introducing multiple inputs. We aim to validate our hypothesis that the slicer is no longer the performance bottleneck. We also assess the overhead introduced by the newly added reorder function (RQ4).

We run our experiments on both synthetic and real traces. The former are monitored with the collapse-sufficient formula φ_s (Figure 4), which is the common *star* database query [16] augmented with temporal operators. It contains only past temporal operators because these can be monitored more efficiently, which puts a higher load on the framework's input and slicers. We use a trace generator [40] to create random traces with configurable time span, event names, rate, and time-point rate. The trace's *time span* is the difference between the highest and the lowest time-stamp in the trace. Given a trace and a time-stamp, the *event rate* is the number of events with that time-stamp, while the *time-point rate* is the number of databases with that time-stamp. The generator synthesizes traces with the same event and time-point rates at all time-stamps, choosing randomly between the event names P, Q, and R. We configured the generator to produce mostly R events (99.8%). The events' parameters are sampled from the natural numbers less than 10^9. There is some correlation between the parameters of events with different names (see [40]), which is not relevant for our experiments because of the prevalence of R events. In general, it is highly unlikely that the generated traces satisfy φ_s.

The generator is extended to determine the order in which the events are supplied to the monitor by explicitly generating the *emission time* for each event. The emission times are relative to the monitoring start time. For traces received in-order, the events' emission times correspond to their time-stamps decreased by the value of the first time-stamp in the trace. Otherwise, each event's emission time is additionally delayed by a value sampled from a truncated normal distribution $\mathcal{N}(0,\sigma^2)$ over the interval $(0,\delta_{max})$. In our

Experiment groups	*Synthetic*$_1$	*Synthetic*$_2$	*Synthetic*$_3$	*Nokia*$_1$	*Nokia*$_2$
Formulas	φ_s	φ_s	φ_s	φ_i, φ_d	$\neg\top$
Source distribution	all uniform except in *Synthetic*$_3$, which also has $\left(\frac{2}{3}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}\right)$, $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{6}, \frac{1}{6}\right)$				
Event order	total, partial	partial	partial	partial	partial
Ingestion order	in-order	out-of-order	in-order	in-order	in-order
No. of input sources	1, 2, 4	1, 2, 4	4	1, 2, 4	1, 2, 4
No. of submonitors	16	16	16	1, 4, 16	16
Acceleration	1	1	1	3k, 5k, 7k	3k, 5k, 7k
Trace time span	60s	60s	60s		
Event rate $(1/s)$	500k, 700k, 900k	900k	500k, 700k, 900k	a one day fragment from the Nokia trace with 9.5 million events	
Time-point rate $(1/s)$	1, 2k, 4k	1	1		
Maximum delay (s)	n/a	1, 2, 4	n/a		
Watermark period (s)	n/a	1, 2, 4	n/a		
Use reorder function	✓	✓	✓	✓	✓, ✗
Repetitions	10	5	1	1	5

Table 1. Summary of the parameters used in the experiments

experiments we fix $\sigma = 2$ and vary the *maximum delay* δ_{max} of events. The generator also adds a watermark after fixed time-stamp increments called *watermark periods*.

Besides the synthetic traces, we also use a real system execution trace from Nokia's Data Collection Campaign [8]. The trace captures how Nokia's system handled the campaign's data. Namely, it collected phone data of 180 participants and propagated it through three databases: db1, db2, and db3. The data was uploaded directly to db1, while the system periodically copied the data to db2, where data was anonymized and copied to db3. The participants could query and delete their own data stored in db1. The system must propagate the deletions to all databases, which is formalized by formulas φ_i and φ_d (Figure 4). Since the trace spans a year, to evaluate our tool in a reasonable amount of time, we pick a one day fragment (starting at time-stamp 1282921200) containing roughly 9.5 million events with a high average event rate of about 110 events per second.

To perform online monitoring, we use a replayer tool [40] that emits the trace in real time based on its time-stamps or (the generated) explicit emission times. The tool can be configured to accelerate the emission of the trace proportionally to its event rate, which allows for a meaningful performance evaluation since the trace characteristics are retained. For our multi-source monitor we use one replayer instance per input source. We evaluate only the implementation that uses TCP sockets. The k input sources are obtained by assigning each event to one of the sources based on a discrete probability distribution called *source distribution*, e.g., the source distribution $\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right)$ is the uniform distribution for $k = 4$. We use other source distributions to investigate RQ2. Both the Nokia and the synthetic traces have explicit time-points, which are used as the partitions' indices. To simulate partially-ordered events, we replace the indices with the appropriate time-stamps.

Table 1 summarizes the parameters used in all our experiments. There are five experiment groups: three using the synthetic traces and two using the Nokia traces. We perform a separate monitoring run for each combination of parameters within one group.

We used a server with two sockets, each containing twelve Intel Xeon 2.20GHz CPU cores with hyperthreading. This effectively supports up to 48 independent parallel computations. We measure the worst-case latency achieved during our experiments.

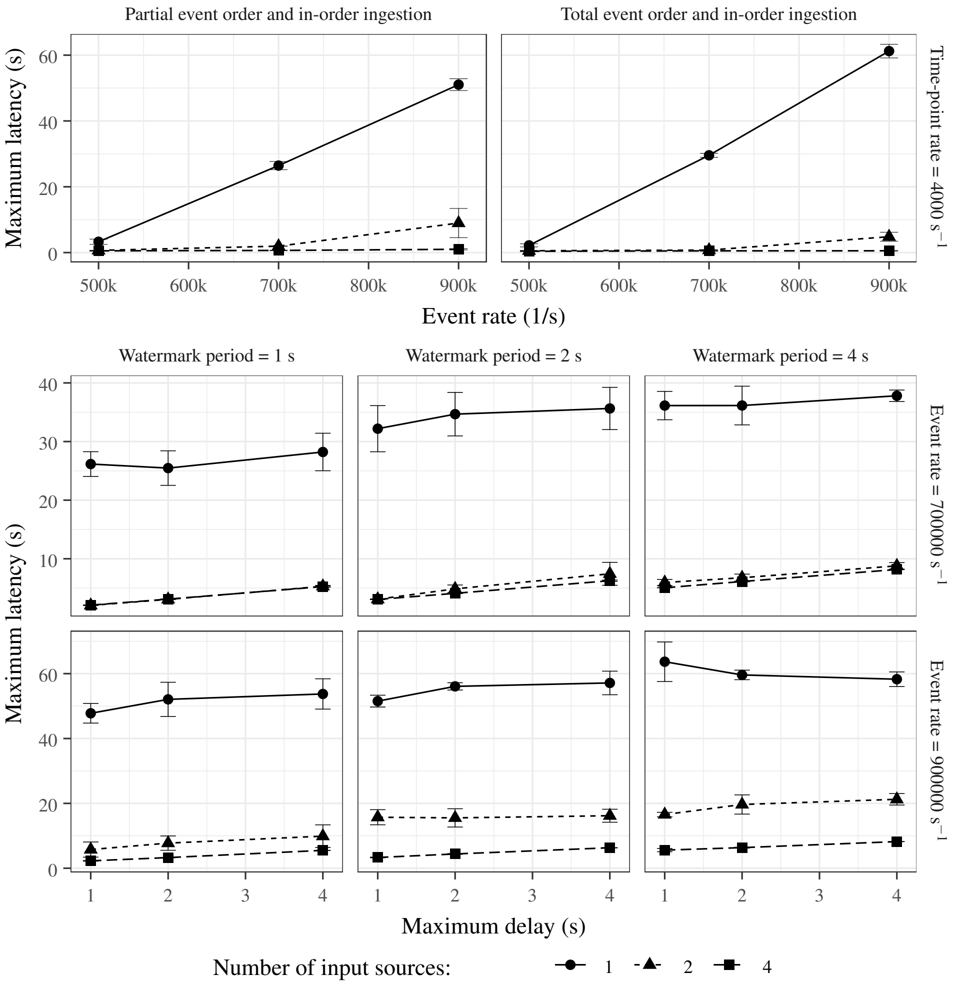

Fig. 5. Results of the *Synthetic*$_1$ (first row) and *Synthetic*$_2$ (second and third row) experiment groups

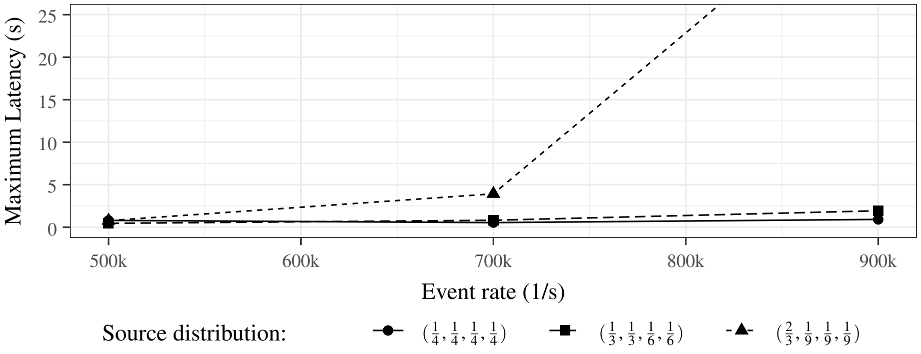

Fig. 6. Results of the *Synthetic*$_3$ experiment group

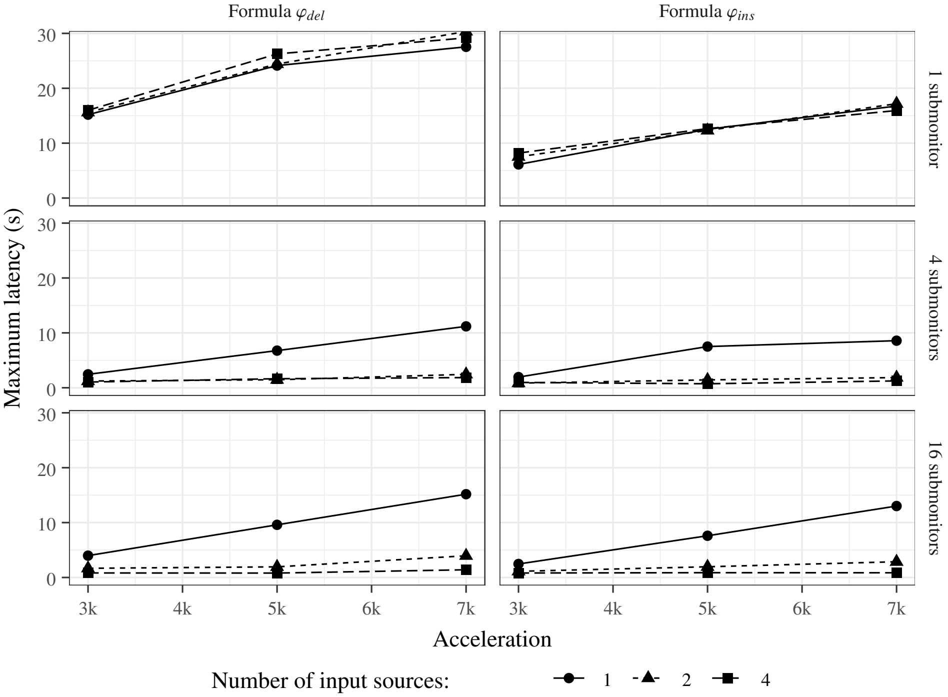

Fig. 7. Results of the *Nokia*$_1$ experiment group

In general, monitor latency is the difference between the time a monitor consumes an event and the time it finishes processing the event. Thus, at regular intervals, the replayer injects a *latency marker*, which is a special event tagged with its creation time and a sequence number local to its source. Each such marker is then propagated by our framework, preserving its order relative to other events from the same input source. It is treated as part of the preceding event, effectively measuring its processing time. The slicers duplicate and forward latency markers to all parallel submonitors, such that each submonitor receives every latency marker from each source. Finally, for every sequence number, the last operator in the framework aggregates all latency markers (coming both from the different input sources and the different parallel submonitors) and calculates the worst-case latency. For a single monitoring run, we report the maximum of the worst-case latency aggregated over the entire run. To avoid spurious latency spikes, some experiments are repeated (see Table 1) and the mean value is reported with error bars showing two standard errors.

The results of our experiments are shown in Figures 5–8. The experiments *Synthetic*$_1$ and *Synthetic*$_2$ (Figure 5) answer RQ1. Increasing the number of input sources decreases the worst-case latency, which is particularly evident with high event rates. For instance, when monitoring traces with event rate 900k, we improve the maximum latency by 10 seconds if we double the number of input sources. The relationship between the maximum event rate at a fixed latency and the number of sources appears to be slightly sublinear. We conjecture that this is due to duplicate events that the slicers necessarily emit for some formulas [40]. Therefore, having more slicers increases the framework's total load.

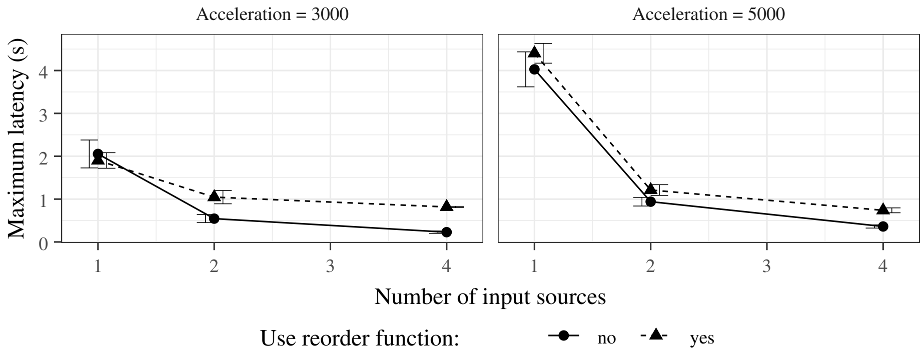

Fig. 8. Results of the *Nokia*$_2$ experiment group

As expected, *Synthetic*$_2$ shows that the watermark period and the maximum delay establish a lower bound on the maximum latency. These parameters determine the minimum amount of time the reorder function must buffer out-of-order events, which our latency measurements capture. We note that the time-point rate has not influenced the monitoring performance in our experiments; we therefore omitted the plots that show different time-point rates and fix the time-point rate to 4000 in *Synthetic*$_1$.

RQ2 is answered by experiment *Synthetic*$_3$ (Figure 6) where we fix the number of input sources to 4 and change the source distribution. The maximum latency is only affected for high event rates and highly skewed source distributions (i.e., when most of the events belong to one source). Otherwise, our framework shows robust performance.

The results of *Nokia*$_1$ (Figure 7) answer RQ3 and validate our hypothesis that increasing the number of sources can improve monitoring performance in realistic monitoring use cases. Increasing the number of sources is ineffective only when parallel submonitors are themselves the performance bottleneck (e.g., when using only one submonitor).

In *Nokia*$_2$ we monitor the Nokia trace without using the reorder function (RQ4). To retain soundness, we monitor the formula $\neg\top$. The experiment shows that the reorder function introduces negligible overhead: less than 1 second of maximum latency.

7 Conclusion

We have developed the first scalable online monitor for centralized, first-order specifications that can efficiently monitor executions of distributed systems. Specifically, we have defined a partitioned temporal structure (PTS) that models an execution of a distributed system, i.e., a sequence of partially-ordered observations received out-of-order. We have extended our monitoring framework to support multiple sources and proved its correctness. Moreover, we empirically show a significant performance improvement over the framework's single-source variant. For example, in our experiments with real data, we could more than double the event rate, from an average of about 330k to 770k events per second by using two sources instead of one, while achieving the same maximum latency. As future work, we plan to combine our framework with monitors that inherently support out-of-order observations [13] or imprecise time-stamps [9], and make our (now parallel) slicing adaptive [41] with respect to changes in the trace characteristics.

Acknowledgment. Dmitriy Traytel and the anonymous reviewers helped us improve this paper. This research is funded by the US Air Force grant "Monitoring at Any Cost" (FA9550-17-1-0306) and by the SNSF grant "Big Data Monitoring" (167162). The authors are listed alphabetically.

References

1. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., et al.: Aurora: a new model and architecture for data stream management. VLDB J. **12**(2), 120–139 (2003)
2. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The Dataflow Model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proc. VLDB Endow. **8**(12), 1792–1803 (2015)
3. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Popa, L., et al. (eds.) PODS 2002. pp. 1–16. ACM (2002)
4. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on RV, LNCS, vol. 10457, pp. 1–33. Springer (2018)
5. Basin, D., Bhatt, B.N., Krstić, S., Traytel, D.: Almost event-rate independent monitoring. FMSD **54**(3), 449–478 (2019)
6. Basin, D., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring of temporal specifications. FMSD **49**(1-2), 75–108 (2016)
7. Basin, D., Gras, M., Krstić, S., Schneider, J.: Implementation, experimental evaluation, and Isabelle/HOL formalization associated with this paper. `https://github.com/krledmno1/krledmno1.github.io/releases/download/v1.0/multi-source.tar.gz` (2020)
8. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: Monitoring data usage in distributed systems. IEEE Trans. Software Eng. **39**(10), 1403–1426 (2013)
9. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: On real-time monitoring with imprecise timestamps. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 193–198. Springer (2014)
10. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
11. Basin, D., Klaedtke, F., Zălinescu, E.: Failure-aware runtime verification of distributed systems. In: Harsha, P., Ramalingam, G. (eds.) FSTTCS 2015. LIPIcs, vol. 45, pp. 590–603. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2015)
12. Basin, D., Klaedtke, F., Zălinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Comp., vol. 3, pp. 19–28. EasyChair (2017)
13. Basin, D., Klaedtke, F., Zălinescu, E.: Runtime verification over out-of-order streams. ACM Trans. Comput. Log. **21**(1), 5:1–5:43 (2020)
14. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. FMSD **48**(1-2), 46–93 (2016)
15. Bauer, A., Küster, J., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 59–75. Springer (2013)
16. Beame, P., Koutris, P., Suciu, D.: Communication steps for parallel query processing. J. ACM **64**(6), 40:1–40:58 (2017)
17. Becker, D., Rabenseifner, R., Wolf, F., Linford, J.C.: Scalable timestamp synchronization for event traces of message-passing applications. Parallel Comput. **35**(12), 595–607 (2009)
18. Bersani, M.M., Bianculli, D., Ghezzi, C., Krstić, S., San Pietro, P.: Efficient large-scale trace checking using MapReduce. In: Dillon, L., et al. (eds.) ICSE 2016. pp.888–898. ACM (2016)
19. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink™: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)
20. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed systems - concepts and designs (3. ed.). International computer science series, Addison-Wesley-Longman (2002)

21. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., et al.: LOLA: runtime monitoring of synchronous systems. In: TIME 2005. pp. 166–174. IEEE (2005)
22. Desnoyers, M., Dagenais, M.R.: The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In: OLS 2006. pp. 209–224 (2006)
23. El-Hokayem, A., Falcone, Y.: THEMIS: a tool for decentralized monitoring algorithms. In: Bultan, T., Sen, K. (eds.) ISSTA 2017. pp. 372–375. ACM (2017)
24. El-Hokayem, A., Falcone, Y.: On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. ACM Trans. Softw. Eng. Methodol. **29**(1), 1:1–1:57 (2020)
25. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. (2020), to appear
26. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 152–168. Springer (2016)
27. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime verification for decentralised and distributed systems. In: Bartocci, E., Falcone, Y. (eds.) Lectures on RV, LNCS, vol. 10457, pp. 176–210. Springer (2018)
28. Gras, M.: Scalable Multi-source Online Monitoring. Bachelor's thesis, ETH Zürich (2020)
29. Hallé, S., Khoury, R., Gaboury, S.: Event stream processing with multiple threads. In: Lahiri, S.K., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 359–369. Springer (2017)
30. Havelund, K., Peled, D., Ulus, D.: First order temporal logic monitoring with BDDs. In: Stewart, D., Weissenbacher, G. (eds.) FMCAD 2017. pp. 116–123. IEEE (2017)
31. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Lectures on RV, LNCS, vol. 10457, pp. 61–102. Springer (2018)
32. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: NetDB 2011. vol. 11, pp. 1–7 (2011)
33. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Runtime verification of real-time event streams under non-synchronized arrival. Software Quality Journal (2020)
34. Mills, D.L.: Internet time synchronization: The network time protocol. RFC **1129**, 1 (1989)
35. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: IPDPS 2015. pp. 494–503. IEEE (2015)
36. Raszyk, M., Basin, D., Krstić, S., Traytel, D.: Multi-head monitoring of metric temporal logic. In: Chen, Y., et al. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 151–170. Springer (2019)
37. Raszyk, M., Basin, D., Traytel, D.: Multi-head monitoring of metric dynamic logic. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302. Springer (2020), to appear.
38. Reger, G., Rydeheard, D.E.: From first-order temporal logic to parametric trace slicing. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 216–232. Springer (2015)
39. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Log. Methods Comput. Sci. **8**(1) (2012)
40. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Scalable online first-order monitoring. In: Colombo, C., Leucker, M. (eds.) RV. pp. 353–371. Springer (2018)
41. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Adaptive online first-order monitoring. In: Chen, Y., et al. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 133–150. Springer (2019)
42. Schneider, J., Basin, D., Krstić, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 310–328. Springer (2019)
43. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Finkelstein, A., et al. (eds.) ICSE 2004. pp. 418–427. IEEE (2004)
44. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: Beeri, C., Deutsch, A. (eds.) PODS 2004. pp. 263–274. ACM (2004)
45. Tucker, P.A., Maier, D.: Exploiting punctuation semantics in data streams. In: Agrawal, R., Dittrich, K.R. (eds.) ICDE 2002. p. 279. IEEE (2002)