A Benchmark Generator for Online First-Order Monitoring

Srđan Krstić and Joshua Schneider

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

Abstract. We present a randomized benchmark generator for attesting the correctness and performance of online first-order monitors. The benchmark generator consists of three components: a stream generator, a stream replayer, and a monitoring oracle. The stream generator produces random event streams that conform to user-defined characteristics such as event frequencies and distributions of the events' parameters. The stream replayer reproduces event streams in real time at a user-defined velocity. By varying the stream characteristics and velocity, their impact on the monitor's performance can be analyzed. The monitoring oracle provides the expected result of monitoring the generated streams against metric first-order regular specifications. The specification languages supported by most existing monitors are either a subset of or share a large common fragment with the oracle's language. Thus, we envision that our benchmark generator will be used as a standard correctness and performance testing tool for online monitors.

Keywords: Online Monitoring · Temporal Logic · Benchmark

1 Introduction

Monitors lie at the core of runtime verification (RV) [4]. Given a sequence of time-stamped events and a property formulated in a specification language, they determine whether the property is satisfied at every point in the stream. The monitored properties can vary in their complexity, ranging from simple invariants to complex patterns expressing qualitative [20,15] and quantitative [17,8] temporal relations between events. Particularly challenging are first-order [7,5] and aggregation [11,9] properties, which additionally refer to the events' parameters. Monitors for such properties perform complex data process tasks, which introduces the potential for bugs that are difficult to detect. Moreover, theoretical analysis of the monitors' algorithms can often not provide sufficient insight into their performance. These two reasons motivate thorough, automated testing of monitors for expressive specification languages. We present a benchmark generator for this purpose that randomly generates events with reference verdicts.

We distinguish between *online* and *offline* monitors [16]. Offline monitors read events from a finite event stream (called an event *log*) in an arbitrary fashion, while online monitors must sequentially analyze a (potentially unbounded) *event stream*. Due to the nature of streams, each event can be read only once.

Hence, an online monitor must keep all relevant events in its memory. Another challenge for online monitors is events arriving out-of-order, which may be caused by unreliable communication channels over which the events are transmitted.

The performance of an online monitor can be assessed in terms of its memory usage and its latency. The latency of processing a single event is the time difference between the moment the event is read and the moment it has been fully processed by the monitor. Latency and memory usage depend on two main factors: the complexity of the monitored property and the characteristics of the event stream, such as its velocity (i.e., the number of events per second), the distribution of the different event types, and the maximum delay of out-of-order events.

The benchmark generator presented in this paper focuses on the event stream characteristics. The main idea is to generate streams that are particularly challenging for online first-order monitors. We provide three tools: a stream GENERATOR, a stream REPLAYER, and a monitoring ORACLE. The GENERATOR generates a stream with user-defined characteristics and passes it to the REPLAYER, which feeds it to an online monitor at a user-defined velocity. The ORACLE provides the expected correct result (a stream of verdicts) for the generated stream, given a property specified in a monitorable fragment of metric first-order dynamic logic (MFODL) [5]. Since MFODL is a very expressive language, our tools can be used to test existing monitors for a large class of properties.

The GENERATOR and REPLAYER components were originally developed to assess the performance of our scalable online monitor [22,21,6], which is sensitive to the event stream characteristics. Together with the ORACLE component, the GENERATOR can be used to test the correctness of monitoring tools, which we have already done [5,23] for a number of existing monitors via differential testing [18].

This work was presented in the benchmark challenge [24] at the RV 2018 conference [14]. Since then, we extended our benchmark generator to 1) generate streams with arbitrary event signatures; 2) use the correct-by-design monitor VeriMon [5,23] as the ORACLE; and 3) generate out-of-order events streams.

2 Benchmark Description

In this section, we first introduce event streams and define the stream characteristics that can be configured in our benchmark generator. We then describe the generator's three main components.

2.1 Event Streams and Stream Characteristics

An *event* is a tuple of data values that is labeled with an *event type*. The values' domain $\mathbb{D}$ typically includes strings and integers. Every event type R has an associated arity $\alpha(R)$ defining the number of data values for this type. We call $1, \ldots, \alpha(R)$ the *attributes* of the type R. For example the following line in the `/var/log/auth.log` file

```
Jul 7 17:14:11 mbp sshd[375]: Accepted publickey for root from 10.11.1.3:5161
```

corresponds to the event login("10.11.1.3", 5161, "mbp", "root", 375, "publickey") with type login and arity $\alpha(\mathsf{login}) = 6$. Every event has an associated time-stamp,

Name	Notation	Definition
Index rate	ι_τ	$\lvert\{i \in \mathbb{N} \mid \tau = \tau_i\}\rvert$
Event rate	ε_τ	$\lvert\{e \in D_i \mid \tau = \tau_i\}\rvert$
Relation rate	$\rho_\tau(R)$	$\lvert\{R(d_1, ..., d_{\alpha(R)}) \in D_i \mid \tau = \tau_i\}\rvert$
Relation frequency	$f_\tau(R)$	$\rho_\tau(R)/\varepsilon_\tau$
Data rate	$\delta_\tau(d, R, i)$	$\lvert\{R(d_1, \ldots, d_{\alpha(r)}) \in D_i \mid d_i = d \land \tau = \tau_i\}\rvert$
Heavy hitters	$\mathcal{H}_\tau(R, i)$	$\left\{ d \in \mathbb{D} \;\middle\vert\; \sum_{\tau_j \in [\tau_0, \tau]} \delta_{\tau_j}(d, R, i) > \frac{\sum_{\tau_j \in [\tau_0,\tau]} \rho_{\tau_j}(R)}{p} \right\}$

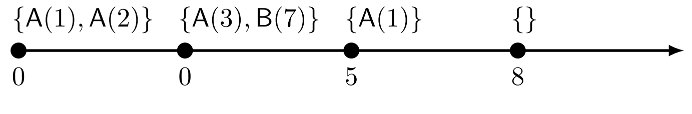
Table 1: Summary of stream characteristics

modeled as a natural number. The use of naturals is realistic as time is often recorded in the UNIX format. For example, the event in the above log line has the associated time-stamp 1594142051, which encodes July 7 2020, 17:14:11 in UNIX format, assuming the GMT time zone and a one second time granularity.

We group a finite number of events that happen concurrently (from the event source's point of view) into *databases*. An (*event*) *stream* is thus an infinite sequence $\langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$ of databases D_i with associated time-stamps τ_i. We distinguish between the time-stamp τ_i and its index in the stream i, also called a *time-point*. Specifically, a stream may have the same time-stamp $\tau_i = \tau_j$ at different indices $i \neq j$, i.e., event sources may record the order of events with higher precision than the time-stamps' granularity. Time-stamps must be non-decreasing ($\forall i.\ \tau_i \leq \tau_{i+1}$) and always eventually strictly increasing ($\forall \tau.\ \exists i.\ \tau < \tau_i$). The above example can be represented by the tuple $\langle 1594142051, D \rangle$ where D is a singleton database containing the login event.

In the following, we introduce the relevant stream characteristics. Their definitions are summarized in Table 1, where we fix a stream $\langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$. The *index rate* ι_τ at time τ is the number of time-points in one time unit. The *event rate* ε_τ at time τ is the total number of events in one time unit. The rate of events with type R is called R's *relation rate*. The *relation frequency* of R at τ, denoted by $f_\tau(R)$, is the ratio of R's relation rate and ε_τ. The *data rate* of a data value d at time τ with respect to the ith attribute of R is the rate of events R that carry the value d in the ith attribute. Finally, we define *heavy hitters* $\mathcal{H}_\tau(R, i)$ as the set of data values that occur as the ith attribute of R events disproportionately often in the stream prefix up to τ. This characteristic differs from the previous ones in that it refers to a prefix instead of a single time-stamp. Heavy hitters are parameterized by $p \in \mathbb{N} - \{0\}$, which is typically the monitor's level of parallelism [22].

We exemplify all the stream characteristics on the stream $\langle 0, \{\mathsf{A}(1), \mathsf{A}(2)\}\rangle$ $\langle 0, \{\mathsf{A}(3), \mathsf{B}(7)\}\rangle \langle 5, \{\mathsf{A}(1)\}\rangle \langle 8, \{\}\rangle \ldots$ depicted in the following figure.



The stream is represented as an arrow with time-points shown as black circles. Databases are drawn above, while time-stamps are the numbers below the circles. The table below shows all the stream characteristics for $p = 3$ and $\tau \in \{0, 8\}$.

Name	Examples
Index rate	$\iota_0 = 2$, $\iota_8 = 1$
Event rate	$\varepsilon_0 = 4$, $\varepsilon_8 = 0$
Relation rate	$\rho_0(\mathsf{A}) = 3$, $\rho_0(\mathsf{B}) = 1$, $\rho_8(\mathsf{A}) = 0$, $\rho_8(\mathsf{B}) = 0$
Relation frequency	$f_0(\mathsf{A}) = \frac{3}{4}$, $f_0(\mathsf{B}) = \frac{1}{4}$, $f_8(\mathsf{A}) = \infty$, $f_8(\mathsf{B}) = \infty$
Data rate	$\delta_0(1, A, 1) = \delta_0(2, A, 1) = \delta_0(3, A, 1) = \frac{1}{3}$, $\delta_0(1, B, 1) = 1$ $\delta_8(1, A, 1) = \delta_8(2, A, 1) = \delta_8(3, A, 1) = \delta_8(1, B, 1) = 0$
Heavy hitters	$\mathcal{H}_0(\mathsf{A}, 1) = \{\}$, $\mathcal{H}_0(\mathsf{B}, 1) = \mathcal{H}_8(\mathsf{B}, 1) = \{7\}$, $\mathcal{H}_8(\mathsf{A}, 1) = \{1\}$

2.2 Specification and Oracle

Our benchmark generator can generate streams for any specification that is expressible as a monitorable metric first-order dynamic logic (MFODL) [5] formula. MFODL extends MFOTL [7] with regular expressions.

The GENERATOR gives more control over the stream generation process for a special family $\mathcal{F}$ of specifications. For example, it is possible to define the expected violation frequency. The family is built around a single temporal pattern consisting of three event types A, B, and C. The specifications differ only in the way these events are related among each other. They can be formalized using the parametric MFODL formula $\Box\,\forall \boldsymbol{v}.\ \big(\blacklozenge_{[0,w)}\, A(\boldsymbol{v}_A)\big) \wedge B(\boldsymbol{v}_B) \rightarrow \Box_{[0,w)}\, \neg C(\boldsymbol{v}_C)$, where is w is a positive integer and $\boldsymbol{v}_A$, $\boldsymbol{v}_B$, and $\boldsymbol{v}_C$ are variable patterns. Informally, the formula states that *whenever there is a B event that was preceded by a matching A event less than w time units ago, there must* not *be a matching C event within the next w time units.* The events are parametrized by integer values. Two events with different types match if their values coincide according to the variable patterns $\boldsymbol{v}_A$, $\boldsymbol{v}_B$, and $\boldsymbol{v}_C$, respectively. For example, if $\boldsymbol{v}_A = (x, y)$ and $\boldsymbol{v}_B = (y, z)$, then the events $A(1, 2)$ and $B(2, 5)$ match, but $A(1, 2)$ and $B(1, 5)$ do not. The variable patterns can be any three non-empty lists of variables such that at least two pairs of patterns each have at least one variable in common. We chose to specialize our benchmark for this family of specifications since they are commonly used in database systems to benchmark the performance of joins [10].

The ORACLE provides the expected output of monitoring any MFODL formula on any *in-order* stream generated by the GENERATOR. The ORACLE is implemented by VeriMon [5], a correct-by-design monitor, that has been formalized in a proof assistant. Its high trustworthiness and expressiveness allows us to attest the correctness of a wide variety of existing monitors [23,5].

2.3 Generating Streams

The GENERATOR produces a random but reproducible event stream. Since it generates output as quickly as possible, one must use the REPLAYER (see Section 2.4 below) to simulate a more realistic real-time stream for an online monitor.

When used with arbitrary specifications, the GENERATOR expects a signature file describing all the event types and their arities. Users can also configure the event rate, index rate and the value of the first time-stamp. The GENERATOR then creates a random stream with consecutive time-stamps and constant event and index rates. Event types are chosen uniformly at random. The GENERATOR maintains a configurable number of unique most recently sampled data values. It samples from this pool with a configurable probability. Otherwise, a fresh value is sampled uniformly from the set $D = \{1, \ldots, 10^9\}$. This ensures common parameter values among events, which exercises non-trivial join computations.

If the stream is to be monitored against the fixed family $\mathcal{F}$ (Section 2.2), the variable patterns can be chosen freely by the user. There are also three built-in patterns: star ($\boldsymbol{v}_A = (w, x)$, $\boldsymbol{v}_B = (w, y)$, and $\boldsymbol{v}_C = (w, z)$), linear ($\boldsymbol{v}_A = (w, x)$, $\boldsymbol{v}_B = (x, y)$, and $\boldsymbol{v}_C = (y, z)$), and triangle ($\boldsymbol{v}_A = (x, y)$, $\boldsymbol{v}_B = (y, z)$, and $\boldsymbol{v}_C = (z, x)$). Events A, B and C are generated randomly and independently according to user-specified relation frequencies $f_\tau(A)$, $f_\tau(B)$, and $f_\tau(C)$, which are constant with respect to τ. There are, however, some constraints: (1) the sum of all three frequencies must be 1; (2) $f_\tau(A)$ can be at most $f_\tau(B)$; and (3) the frequency of violations can be at most the minimum of $f_\tau(A)$ and $f_\tau(C)$.

For $\mathcal{F}$, the data values are chosen randomly and independently under the following constraints: (1) every A event must be matched with a B event within the interval w to ensure that the premise of the specification is satisfied frequently; (2) a user-specified percentage of violations must be generated. Constraint (2) is enforced by generating an appropriate number of C events matching both a proceeding B event and an A event before that (both within appropriate intervals). By default, values are sampled uniformly from D. It is also possible to select a Zipf distribution per variable, which has the probability mass function $p(x) = (x - s)^{-z} / \sum_{n=1}^{10^9} n^{-z}$ for $x \in \{s + 1, s + 2, \ldots, s + 10^9\}$. The larger the exponent $z > 0$ is, the fewer values have a large relative frequency. The parameter s is the start value, which can also be configured to control the specific heavy hitter values. Events that form a violation are always drawn from the uniform distribution to prevent unintended matchings. Likewise, Zipf-distributed values of C events are increased by 1 000 000. Note that there is still a nonzero probability that additional violations occur, even though D is large.

The generator is extended to determine the order in which the events are supplied to the monitor by generating the explicit *emission time* for each event. The emission times are relative to the time when monitoring starts. For in-order event streams, the events' emission times correspond to their time-stamps decreased by the value of the first time-stamp in the stream. To create out-of-order streams the GENERATOR increases each event's emission time by a value sampled from a truncated normal distribution $\mathcal{N}(0, \sigma^2)$ over the interval $[0, \delta_{max}] \cap \mathbb{N}$. Both the *standard deviation* σ and the *maximum delay* δ_{max} are configurable. The GENERATOR also adds a watermarks after configurable time-stamp increments called *watermark periods*. A watermark is a time-stamp which is a strict lower bound on all time-stamps of the events received in the future. They are commonly used in stream processing systems to deal with out-of-order events [2,1,19,13,12].

2.4 Replaying Streams

The time-stamps in an event stream do not necessarily correlate to the (real) times at which the corresponding events are received by an online monitor. Therefore, we distinguish the *ingestion time* of an event from its time-stamp. The *ingestion rate* is the total number of events received by the monitor per unit of (real) time. The REPLAYER tool reproduces an event stream (or log) with an ingestion rate proportional to the stream's event rate. The proportionality constant, called *acceleration*, is chosen by the user. For example, an acceleration of 2 will replay the stream twice as fast. Thus the REPLAYER can be used to generate workloads with different ingestion rates from the same data. This allows for a meaningful performance evaluation as the stream characteristics are retained.

Upon startup, the REPLAYER immediately outputs all events with the smallest time-stamp in its input. The subsequent events with the next time-stamp are delayed proportionally to the difference between the two time-stamps (which are interpreted as seconds), where the delay factor is the inverse of the acceleration parameter. This process is repeated for each unique time-stamp in the stream.

To reproduce streams with out-of-order events, the REPLAYER uses the *emission times* provided by the GENERATOR, instead of the events' time-stamps.

3 User Guide

We provide our benchmark generator as a Docker image [25]. In the following, we assume that Docker version 19.03.8 or higher is installed and configured properly. The components of the benchmark generator can be invoked with the command

```
docker run -i infsec/benchmark component [arguments ...]
```

where `component` is replaced by the name of the component. Below, we omit the Docker part of the invocation and only show the component and its arguments.

The GENERATOR prints the generated stream to the standard output.

```
generator {-sig s | -S | -L | -T | -P p} [options ...] [length]
```

If `length` is given, a finite log of that length (in seconds) is produced instead of an unbounded stream. Options `-e rate`, `-i rate`, `-t time`, and `-seed seed` modify the event rate, the index rate, the first time-stamp, and the GENERATOR's initial random seed. They default to values 10, 1, 0, and 314159265, respectively.

It is required to provide either a signature file describing arbitrary event types (the `-sig` flag) or use predefined events. When using the signature, the user can additionally use flags `-q size` and `-r prob` to define the number of the most recently sampled unique data values (default 100) and the probability to sample a fresh data value (default 0.1). If one opts for predefined events, one can select either a built-in or a custom variable pattern. The flags `-S` (star), `-L` (linear), and `-T` (triangle) select the built-in star, triangle, or linear pattern, respectively.

A pattern consists of names for the three event types in the specification family $\mathcal{F}$, each followed by a non-empty list of variable names. Whenever a variable name occurs multiple times, the appropriate events must match. A custom pattern is supplied as a single argument after option `-P`. For example,

```
$ generator -P "e1(x)e2(x,y,z)e3(y,z)" 10
```

generates a log spanning 10 seconds, where the predefined events A, B, and C are renamed to `e1`, `e2`, and `e3`, respectively. Exactly three event types must be specified. Events `e1` have one attribute, while events `e2` have three. They match if they agree on their first attribute. Events `e2` and `e3` match if the second and third attribute of `e2` are equal to the first and second attribute of `e3`. Event and variable names can be arbitrary alphanumeric strings, including ' and _.

The relation frequencies of the predefined event types are set with `-pA` *`ratio`* and `-pB` *`ratio`*. The frequency of type C is implied by the frequencies of type A and B because their sum is always 1. For example,

```
$ generator -S -pA 0.1 -pB 0.5
```

generates an unbounded stream for the star pattern. The relation frequency of A events is (approximately) 10 %, that of B events is 50 %, and that of C events is 40 %. If no ratios are given, all event types have roughly the same frequency.

To sample values from a Zipf distribution, the exponent of the distribution can be specified per variable. The exponents for all Zipf-distributed variables are passed as a single argument after option `-z`. For example,

```
$ generator -T -z "x=1.5+3,z=2" -e 1 1 > log.csv
```

generates events following the triangle pattern, with the values of variables x and z following a Zipf distribution with exponents 1.5 and 2. The start value for variable x is 3, while for variable z it is 0 (default). Variable y is distributed uniformly.

The frequency of violations relative to the number of events is set with `-x` *`ratio`* (default: 0.01). The interval size w, which bounds the distance of related events, is set with `-w` *`interval`* (in seconds, default: 10).

The GENERATOR's output uses the (slightly modified) CSV format from the first RV competition [3]. For example, the `log.csv` file contains the following line

```
C, tp=0, ts=0, x0=1000001, x1=1000007
```

representing the finite log $\langle 0, \mathsf{C}(1000001, 1000007)\rangle$ with a single time-point (`tp=0`). The flag `-et` instructs the generator to add explicit emission times to the events based on maximum delay (flag `-md` *`delay`*) and standard deviation (flag `-s` *`stddev`*). The GENERATOR also outputs watermarks after configurable periods (flag `-wp` *`period`*). The emission times are prepended to every line in the output followed by the `'` character. Invoke `generator --help` for more details. For example, the `log.csv` file generated with the flag `-et` set has the following content

```
2'C, tp=0, ts=0, x0=1000001, x1=1000007
2'>WATERMARK 0<
```

The REPLAYER reads events from standard input and copies them with a delay to its standard output. It is invoked with

```
replayer [options ...]
```

The most important option is the acceleration (flag `-a` *acc*), which is 1 by default. Events can be read from another process through a pipe, or from a log file. If a process is used, it needs to be fast enough for the events to be replayed at the proper time. Use a shell redirection to read events from the `log.csv` file

`$ replayer -a 5 < log.csv` or a pipe `$ generator -S -et 5 | replayer -e`

to replay the out-of-order stream based on the explicit emission times (flag `-e`).

With option `-f verimon`, the REPLAYER prints the output in VeriMon's format, which is also used by the ORACLE. Otherwise, the modified CSV format is used. Together with an acceleration of 0, which replays the stream as fast as possible, the tool thus becomes a converter from the CSV to the ORACLE's format.

If the option `-o` *host*`:`*port* is given, a TCP server listening to the provided hostname and port will be created. The first client connecting to the TCP server will receive the event stream. The events are sent once a client has connected. The client may reconnect (flag `-k`), but multiple clients cannot connect.

The REPLAYER maintains some statistics about the number of events processed. It also keeps track of the delay between the scheduled event time and the time the event could be written to the output. The latter may be useful to analyze monitoring latency from the event source point of view. The option `-v` (or `-vv`) generates a compact (or verbose) report once every second, which is written to standard error. The format of the reports is explained in the `README` file accessible via the `--help` flag. Note that delays are tracked only up to the operating system's buffer that is associated with the pipe or socket to the monitor.

The implementation of the REPLAYER uses two threads, one for reading and one for writing events, which are connected by a queue with limited capacity. The verbose report (`-vv`) displays the number of times the the queue is fully drained. If this number is non-zero and especially if it is growing, the queue capacity should be increased with option `-q` *size* (default: 1024).

By default the REPLAYER writes a terminator sequence (`;;`) after every emitted database. This is useful for monitors that use databases as units of input. Terminators are supported by the ORACLE and can be suppressed (flag `-nt`).

The ORACLE reads events in VeriMon format from the standard input and prints a stream of verdicts to the standard output. It is invoked with

```
oracle {{-S | -L | -T | -P p} [-w i] | -sig s -formula f} [options ...]
```

where the arguments define the specification as described for the GENERATOR. It assumes that the input stream is generated following the same event signature, or for the fixed specification from $\mathcal{F}$. Invoke `oracle -help` for possible options. Finally, we give an example combining all three components into a single command:

```
$ generator -S 100 | replayer -f verimon -a 10 | oracle -S
```

This generates a stream spanning 100 seconds according to the star pattern, speeds it up 10 times and converts it to the ORACLE (i.e., VeriMon) format. The ORACLE then produces the stream of expected violations.

Acknowledgment. We thank Matthieu Gras for his contributions to the stream generator. VeriMon was developed in collaboration with Dmitriy Traytel and Martin Raszyk. This research is supported by the US Air Force grant "Monitoring at Any Cost" (FA9550-17-1-0306) and by the Swiss National Science Foundation grant "Big Data Monitoring" (167162). The authors are listed alphabetically.

References

1. T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, 2013.
2. A. Alexandrov, R. Bergmann, S. Ewen, J. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.
3. E. Bartocci, B. Bonakdarpour, and Y. Falcone. First international competition on software for runtime verification. In B. Bonakdarpour and S. A. Smolka, editors, *RV 2014*, volume 8734 of *LNCS*, pages 1–9. Springer, 2014.
4. E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In E. Bartocci and Y. Falcone, editors, *Lectures on Runtime Verification*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018.
5. D. Basin, T. Dardinier, S. Krstić, L. Heimes, M. Raszyk, J. Schneider, and D. Traytel. A formally verified, optimized monitor for metric first-order dynamic logic. In *10th International Joint Conference on Automated Reasoning (IJCAR)*, 2020. To appear.
6. D. Basin, M. Grass, S. Krstić, and J. Schneider. Scalable online monitoring of distributed systems. Submitted at 20th International Conference on Runtime Verification (RV), 2020.
7. D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.
8. D. Basin, S. Krstić, and D. Traytel. Almost event-rate independent monitoring of metric dynamic logic. In S. Lahiri and G. Reger, editors, *RV 2017*, volume 10548 of *LNCS*, pages 85–102. Springer, 2017.
9. D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods Syst. Des.*, 46(3):262–285, 2015.
10. P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017.
11. D. Bianculli, C. Ghezzi, and S. Krstić. Trace checking of metric temporal logic with aggregating modalities using MapReduce. In D. Giannakopoulou and G. Salaün, editors, *SEFM 2014*, volume 8702 of *LNCS*, pages 144–158. Springer, 2014.
12. P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink$^{\text{TM}}$: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.
13. C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In B. G. Zorn and A. Aiken, editors, *PLDI 2010*, pages 363–375. ACM, 2010.
14. C. Colombo and M. Leucker, editors. *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*. Springer, 2018.

15. G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In F. Rossi, editor, *IJCAI-13*, pages 854–860. AAAI Press, 2013.
16. Y. Falcone, S. Krstić, G. Reger, and D. Traytel. A taxonomy for classifying runtime verification tools. In C. Colombo and M. Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*, pages 241–262. Springer, 2018.
17. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, 1990.
18. W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.
19. H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox: Modern stream processing on a multicore machine. In D. D. Silva and B. Ford, editors, *USENIX ATC 2017*, pages 617–629. USENIX Association, 2017.
20. A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.
21. J. Schneider, D. Basin, F. Brix, S. Krstić, and D. Traytel. Adaptive online first-order monitoring. In Y. Chen, C. Cheng, and J. Esparza, editors, *ATVA 2019*, volume 11781 of *LNCS*, pages 133–150. Springer, 2019.
22. J. Schneider, D. Basin, S. Krstić, F. Brix, and D. Traytel. Scalable online first-order monitoring. In C. Colombo and M. Leucker, editors, *RV 2018*. Springer, 2018.
23. J. Schneider, D. A. Basin, S. Krstic, and D. Traytel. A formally verified monitor for metric first-order temporal logic. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 310–328. Springer, 2019.
24. J. Schneider and S. Krstić. 2018 Runtime Verification Benchmark Challenge – FOStreams benchmark. `https://github.com/runtime-verification/benchmark-challenge-2018.git`, 2018.
25. J. Schneider and S. Krstić. A benchmark generator for online first-order monitoring (Docker image v.1.2.0). `https://hub.docker.com/r/infsec/benchmark/`, 2020.