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, one can analyze their impact on the monitor's performance. 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 timestamped events and a specification (i.e., a property formulated in a specification language), a monitor checks that the specification holds at each point in the sequence and otherwise reports the violations. The monitored properties can range from simple state invariants to complex patterns expressing qualitative [22,14] and quantitative [18,11] temporal relations between events. Particularly challenging are first-order [8,5] and aggregation [13,7] properties, which additionally refer to the events' parameters. The implementation of such monitors is a non-trivial task, which can introduce bugs that are difficult to detect. Moreover, the theoretical analysis of a monitor's algorithm often does not provide sufficient insight into its performance. These two reasons motivate thorough, automated testing. In this paper, we present a benchmark generator that *tests the correctness* and *evaluates the performance* of monitors for expressive specification languages.

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 read from 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 specification 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 mainly on the event stream characteristics. They are not only useful for evaluating a monitor's performance, but also for testing its correctness, as streams with specific characteristics can trigger corner cases in the monitoring algorithm. We provide three tools: a stream GENERATOR, a stream REPLAYER, and a monitoring ORACLE.

The GENERATOR randomly generates a stream with user-defined characteristics. The GENERATOR has two modes. In the first mode, it supports arbitrary specifications by generating events independently at random. This mode is useful for the correctness testing of a monitor against a large number of specifications involving different event types. The second mode is restricted to a family of specifications for which a monitor must compute joins over three relations. This is known to be a difficult problem [12,21] and a core task in first-order monitoring. The second mode is thus tailored to the performance evaluation of first-order monitors. For the restricted family of specifications, the GENERATOR uses biased sampling to match the average violation frequency specified by the user.

The REPLAYER feeds the generated stream to an online monitor at a user-defined velocity, which allows for latency measurements under realistic conditions. The REPLAYER can optionally simulate out-of-order streams by exploiting the randomized emission time-stamps that the GENERATOR adds to the stream.

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 very expressive, our benchmark generator can be used to test the correctness of the majority of the existing monitors over a large class of specifications.

The GENERATOR and REPLAYER were originally developed to assess the performance of our online first-order monitor [28,26,6], which is sensitive to the event stream characteristics. Together with the ORACLE, the GENERATOR can be used to test the correctness of monitoring tools, which we have already done [5,29] for a number of existing monitors via differential testing [20]. We summarize these applications of our benchmark in Section 4.

An earlier version of this work, called FOStreams, was presented in the benchmark challenge [1] at the RV 2018 conference. Since then, we extended our benchmark generator to 1) generate streams with arbitrary event signatures; 2) use the correct-by-design monitor VERIMON [5,29] as the ORACLE; and 3) generate out-of-order event streams.

Related work. From 2014 to 2016, the RV community organized an annual tool competition to address a lack of standardized benchmarks in the field [3]. Its goals (among others) were to design and discuss evaluation methods for RV tools and to inspire new efficient implementations of such tools. A follow-up workshop [25], which replaced the competition in 2017, concluded that one of the obstacles in achieving standardized benchmarks is the diversity of the tools' specification languages. Our benchmark generator focuses on the event streams characteristics, which avoids a strong dependence on the specification language. Such a dependence still exists in the ORACLE, but we hope that its highly expressive language allows meaningful testing of the majority of existing tools.

The community continued to collect and curate benchmarks after 2017 [1]. The benchmark by Li and Rozier [19] uses SMT solvers to generate satisfying or violating event streams for propositional monitors. In contrast, our work supports first-order specifications and it relies on an orthogonal approach to stream generation: the ORACLE provides verdicts which are correct by design, while the GENERATOR uses a best-effort strategy to reach the user-defined violation rate. Ulus [30] provides a benchmark generator tailored to propositional monitors and common specification patterns [15] involving parameterized time constraints, whereas we focus on data constraints and the reproduction of real-time streams.

2 The Benchmark Generator

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 benchmark 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
```

can be represented by 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, 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 set of events that happen concurrently (from the event source's point of view) into *databases*. An (*event*) *stream* is thus an infinite sequence $(\tau_i, D_i)_{i \in \mathbb{N}}$ 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

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	$\delta_\tau(d, R, k)$	$\lvert\{R(d_1, \ldots, d_{\alpha(r)}) \in D_i \mid d_k = d \wedge \tau = \tau_i\}\rvert$
Heavy hitters	$\mathcal{H}_\tau(R, k)$	$\left\{ d \in \mathbb{D} \;\middle\vert\; \frac{\sum_{0 \le \tau' \le \tau} \delta_{\tau'}(d, R, k)}{\sum_{0 \le \tau' \le \tau} \rho_{\tau'}(R)} > \frac{1}{p} \right\}$

Table 1: Summary of stream characteristics for the event stream $(\tau_i, D_i)_{i \in \mathbb{N}}$

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 ($\forall i.\ \tau_i \le \tau_{i+1}$) and always eventually strictly increasing ($\forall \tau.\ \exists i.\ \tau < \tau_i$). The above example can be represented by the tuple $(1594142051, D)$ 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 $(\tau_i, D_i)_{i \in \mathbb{N}}$. 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* $\delta_\tau(d, R, k)$ of a data value d at time τ with respect to the kth attribute of R is the number of events R that carry the value d in the kth attribute. Finally, we define the sets of *heavy hitters* $\mathcal{H}_\tau(R, k)$. A heavy hitter is a data value that occurs as the kth attribute of R events disproportionately often in the stream prefix up to τ. This characteristic differs from the previous ones in that it is computed over a prefix instead of a single time-stamp. A value is a heavy hitter if its data rate, relative to the corresponding relation rate, exceeds the threshold $1/p$. The parameter $p \in \mathbb{N} - \{0\}$ is typically the monitor's level of parallelism [27].

We exemplify all the stream characteristics using the stream ρ_{ex} depicted in the following figure, which shows the first four time-points as black circles. Databases are drawn above, while time-stamps are the numbers below the circles.

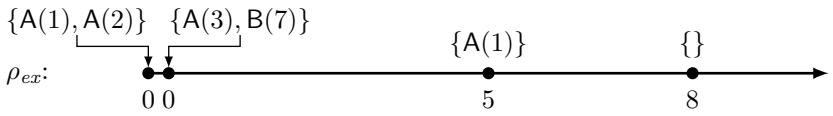

Table 2 lists all the stream characteristics for this stream, where $p = 3$ and $\tau \in \{0, 8\}$. For example, the index rate ι_0 is two because there are two time-points, 0 and 1, with time-stamp 0. Note that two out of the three A events in the time interval $[0, 8]$ carry the data value 1, and $2/3$ is greater than the heavy hitter threshold $1/p = 1/3$. Therefore, the set $\mathcal{H}_8(\mathsf{A}, 1)$ contains 1 as the single heavy hitter (as of time-stamp 8) in the first attribute of A events.

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

Table 2: Stream characteristics of the example stream ρ_{ex}

2.2 Specification and Oracle

Our benchmark generator can be used with arbitrary specifications. Depending on the benchmark's mode, the generated streams are either compatible with all specifications that use a given *signature*, or they are tailored to a single specification from a fixed family (see Section 2.3). A specification's signature describes the finite set of relevant event types together with their arities.

The ORACLE provides the expected output of monitoring any specification expressible in monitorable metric first-order dynamic logic (MFODL) [5] on any *in-order* event stream. MFODL extends MFOTL [8] with regular expressions. The ORACLE is implemented using VERIMON [5], a correct-by-design monitor that has been formally verified in a proof assistant. Its high trustworthiness and expressiveness allows us to attest the correctness of a wide variety of existing monitors [29,5] by comparing their output to the ORACLE's output.

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) to simulate a more realistic real-time stream for an online monitor. The GENERATOR can be operated in two different modes, which we detail below.

Mode I (arbitrary specifications). 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, the 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, which ensures common data values across events and thus increases the likelihood of exercising non-trivial computation inside the monitor. Otherwise, a fresh value is sampled uniformly from the set $\{1, \ldots, 10^9\}$.

Mode II (temporal three-way conjunctions). The GENERATOR gives more control over the stream generation process for a special family $\mathcal{F}_3$ of specifications, which

we call *temporal three-way conjunctions*. For example, it is possible to define the expected violation frequency. The family $\mathcal{F}_3$ is inspired by query patterns that are commonly used in database systems to benchmark the performance of relational joins [12]. Joins are an important operation also for first-order monitors because (the negations of) many specifications contain conjunctions, e.g., any specification involving a response constraint [15]. We augment the conjunctions with temporal operators to increase the joins' input size.

A three-way conjunction is a temporal pattern referring to three event types A, B, and C with integer data values. The specifications differ only in the way these events are related among each other. They can be formalized using the parametric MFODL formula $\Box \forall \boldsymbol{v}.\ \big(\blacklozenge_{[0,w)}\, \mathsf{A}(\boldsymbol{v}_\mathsf{A})\big) \wedge \mathsf{B}(\boldsymbol{v}_\mathsf{B}) \rightarrow \Box_{[0,w)}\, \neg \mathsf{C}(\boldsymbol{v}_\mathsf{C})$, where is w is a positive integer and $\boldsymbol{v}_\mathsf{A}$, $\boldsymbol{v}_\mathsf{B}$, and $\boldsymbol{v}_\mathsf{C}$ are lists of variables. 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. Two events with different types match if their data values coincide according to the variables $\boldsymbol{v}_\mathsf{A}$, $\boldsymbol{v}_\mathsf{B}$, and $\boldsymbol{v}_\mathsf{C}$, respectively. For example, if $\boldsymbol{v}_\mathsf{A} = (x, y)$ and $\boldsymbol{v}_\mathsf{B} = (y, z)$, then the events $\mathsf{A}(1, 2)$ and $\mathsf{B}(2, 5)$ match, but $\mathsf{A}(1, 2)$ and $\mathsf{B}(1, 5)$ do not.

The variable lists, which must be non-empty, can be chosen freely by the user. There are three built-in configurations: star ($\boldsymbol{v}_\mathsf{A} = (w, x)$, $\boldsymbol{v}_\mathsf{B} = (w, y)$, $\boldsymbol{v}_\mathsf{C} = (w, z)$), linear ($\boldsymbol{v}_\mathsf{A} = (w, x)$, $\boldsymbol{v}_\mathsf{B} = (x, y)$, $\boldsymbol{v}_\mathsf{C} = (y, z)$), and triangle ($\boldsymbol{v}_\mathsf{A} = (x, y)$, $\boldsymbol{v}_\mathsf{B} = (y, z)$, $\boldsymbol{v}_\mathsf{C} = (z, x)$). These configurations are again well-known in the database literature [12].

For $\mathcal{F}_3$, the events of type A, B and C are generated randomly and independently according to the user-specified relation frequencies $f_\tau(\mathsf{A})$, $f_\tau(\mathsf{B})$, and $f_\tau(\mathsf{C})$, which are constant with respect to τ. The data values are also 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 the appropriate time intervals). The above constraints imply some restrictions on the user-specified frequencies: the sum of all three frequencies must be 1, $f_\tau(\mathsf{A})$ can be at most $f_\tau(\mathsf{B})$, and the frequency of violations can be at most the minimum of $f_\tau(\mathsf{A})$ and $f_\tau(\mathsf{C})$.

By default, values are sampled uniformly from $D = \{1, \ldots, 10^9\}$. 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 correspondingly larger relative frequency and are thus more likely to be heavy hitters. The parameter s is the start value, which can be used to further 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 the set D is large.

Out-of-order streams. The GENERATOR optionally attaches an *emission time* to every event. The emission times, which are time differences relative to the start of the stream, may be used to determine the order in which the events are supplied to the monitor. For in-order event streams, the emission times correspond to the events' 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 the 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 watermarks after configurable time-stamp increments called *watermark periods* to the stream. 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 handle out-of-order events [2].

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 Usage Examples

We provide our benchmark generator as a ready-to-use Docker image.[1] The source code is available online.[2] 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 -iv `pwd`:/work infsec/benchmark component [options ...]
```

where *component* is one of `generator`, `replayer`, or `oracle`. The command makes the current working directory available to the Docker container. Hence,

[1] `https://hub.docker.com/r/infsec/benchmark/` (version 1.2.1)
[2] `https://bitbucket.org/krle/scalable-online-monitor`

one can access all the files below the current directory using relative paths in the components' options. Each component prints detailed usage information if it is invoked with the `--help` option. In the examples below, we omit the Docker part of the invocation and only show the component and its arguments.

Example: differential testing with Mode I. We explain the steps needed to test the correctness of a monitor against the ORACLE. An MFODL formula and, if necessary, its translation to the monitor's native language must be provided. Here, we use the MFODL formula $\square \forall ip, port.\ \mathsf{login}(ip, port) \rightarrow \Diamond_{[0,60]}\, \mathsf{logout}(ip, port)$, which is loosely inspired by the example from the beginning of Section 2.1. In words, every login from some IP address and port combination must be eventually followed by a matching logout within 60 time units. For simplicity, we assume that the time unit is minutes. Note that the interpretation of the time unit is irrelevant for the GENERATOR; the REPLAYER interprets time-stamps in seconds.

We first describe the signature in a text file `ssh.sig` with the content

```
login(ip,port) logout(ip,port)
```

and the specification (without the prefix $\square \forall$) in a separate file `ssh.spec`:

```
login(ip,port) IMPLIES EVENTUALLY[0,60] logout(ip,port)
```

The syntax for the MFOTL subset is described in [9]. Next, the following command generates a random log for the signature with a length of 300 minutes.

```
$ generator -sig ssh.sig -i 10 -q 20 -r 0.01 300 > ssh.csv
```

The GENERATOR prints the events to its standard output. We use a shell redirection to save them in a file. The option `-i 10` sets the index rate to 10. Together with the default event rate (option `-e`), which is also 10, this implies ten databases per minute with one event each. Options `-q` and `-r` define the number of the most recently sampled unique data values and the probability to sample a fresh data value. Here we use few values (20) and a low probability (0.01) because otherwise there would be many violations of the specification.

The GENERATOR outputs the CSV format from the first RV competition [3]. For example, the `ssh.csv` file begins with the line

```
login, tp=0, ts=0, x0=569872521, x1=373321178
```

representing the event $\mathsf{login}(569872521, 373321178)$ at time-stamp 0. The random generator's seed is fixed and the output is deterministic. The seed can be customized using the `-seed` option. Since VERIMON expects a different format for the input event stream, we invoke the REPLAYER to translate the formats:

```
$ replayer -f verimon -a 0 < ssh.csv > ssh.log
```

Note that `-a 0` disables the real-time replay and events are emitted as quickly as possible. Finally, the ORACLE provides the reference verdicts:

```
$ oracle -sig ssh.sig -formula ssh.spec < ssh.log
@0. (time point 7): (703748452,559514287)
[...]
```

Each line in the output represents a violation, showing the time-stamp, the time-point, and values of *ip* and *port*. If we now ran another monitoring tool on the same specification and log, we could compare its output to this reference.

Example: online performance measurements with Mode II. Here, we illustrate the generation of a *real-time* stream with out-of-order events for the specification family $\mathcal{F}_3$ (Section 2.3). By varying the stream characteristics, one can analyze their impact on the monitor's throughput, latency, and memory usage.

Recall that $\mathcal{F}_3$ is parameterized by three variable lists. One can select either a built-in or a custom variable configuration. The options `-S` (star), `-L` (linear), and `-T` (triangle) select the respective built-in configuration. A custom pattern is supplied as a single argument after the option `-P`. In this example, we will use the triangle specification, i.e., $\Box \forall x, y, z.\ \big(\blacklozenge_{[0,w)}\, \mathsf{A}(x, y)\big) \wedge \mathsf{B}(y, z) \rightarrow \Box_{[0,w)}\, \neg \mathsf{C}(z, x)$.

```
$ generator -T -pA 0.1 -pB 0.5 -z "x=1.5+3,z=2" -e 100
```

The relation frequencies of the three event types are set with `-pA` and `-pB`. The frequency of type C is implied by the frequencies of type A and B because their sum is always 1. In the invocation above, the relation frequency of A events is approximately 10 %, that of B events is 50 %, and that of C events is 40 %. To obtain values from a Zipf distribution, the exponent of the distribution can be specified per variable. The exponents of all Zipf-distributed variables are passed as a single argument after option `-z`. In our case, the values of variables x and z follow 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.

We did not specify the frequency of violations (option `-x`) nor the interval size w (option `-w`), so they assume their default values of 0.01 and 10, respectively. No log length was specified either, which prompts the GENERATOR to produce an unbounded stream as quickly as possible. We can pipe its output into the REPLAYER to obtain a real-time stream, which can be further sent to the monitor under test:

```
$ generator [...] | replayer | some monitor tool
```

The REPLAYER outputs 100 events per second because the generated stream's event rate is 100 (option `-e 100` in the GENERATOR's invocation). With the REPLAYER option `-a 2`, the stream would be replayed twice as fast at 200 events per second. If a pipeline connects the GENERATOR and the REPLAYER, the former needs to be fast enough for the events to be replayed at the proper time. For higher accelerations or event rates, a finite log should be generated and written to a file from where the REPLAYER can read it.

To obtain an out-of-order stream, we must pass additional options:

```
$ generator [...] -et -md 5 -s 2 -wp 1 | replayer -e
```

The flag `-et` instructs the GENERATOR to add explicit emission times to the events based on maximum delay (option `-md`) and standard deviation (option `-s`). The GENERATOR also outputs watermarks after configurable periods (option `-wp`), which appear as lines of the form `>WATERMARK` *time-stamp* `<` in the stream.

4 Applications

We used previous versions of our benchmark generator to assess the performance of our scalable monitoring framework [28], which relies on first-order (sub)monitors to monitor event streams in parallel. The framework initially supported only MonPoly [9] as a submonitor, but it was later extended [27] to also support DejaVu [17]. The framework's performance depends on the stream characteristics shown in Section 2.1. We used the generator in Mode II during the evaluation, which revealed a noticeable impact of the index rate and the specific variable configurations on the monitoring framework's throughput. The framework was later extended to adapt to dynamically changing stream characteristics [26] and to handle multiple event streams with events arriving out-of-order [6]. The evaluation of these extensions was again driven by the generator and replayer. For example, we could confirm a direct relationship between the monitoring latency and both the maximum delay and the watermark period.

In conjunction with the development of VeriMon [29], the generator (in Mode I) and the oracle were used to perform differential testing of both propositional (Aerial [10] and Hydra [23,24]) and first-order monitors (MonPoly [9] and DejaVu [17]). Bugs were discovered in each tool [5].

5 Conclusion and Future Work

Online first-order monitors implement complex algorithms, whose correctness is rarely obvious. Furthermore, they require a highly optimized join implementation to achieve competitive performance. We proposed a benchmark generator for evaluating first-order monitors. It consists of three components: a stream generator, stream replayer, and a monitoring oracle. The stream generator and replayer produce random event streams in real time with highly customizable characteristics suitable for evaluating the performance of join implementations in monitors. The monitoring oracle provides the correct monitoring output for monitorable metric first-order regular specifications, which allows for the correctness testing of a large class of first-order monitors.

In the future, we would like to support other event stream formats (e.g., JSON) and additional data value types (e.g., strings). Moreover, the current stream generator determines the time-stamps based on the event rate and log length only. We would like to give the users additional control over the distribution of the time-stamp values. Finally, we plan to improve and publish a version of the generator that provides multiple randomized event streams resembling those obtained from distributed systems [6].

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. Runtime Verification Benchmark Challenge. `https://github.com/runtime-verification/benchmark-challenge-2018`, 2018.
2. T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. 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. E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Rosu, J. Signoles, D. Thoma, E. Zălinescu, and Y. Zhang. First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. *Int. J. Softw. Tools Technol. Transf.*, 21(1):31–70, 2019.
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, L. Heimes, S. Krstić, M. Raszyk, J. Schneider, and D. Traytel. A formally verified, optimized monitor for metric first-order dynamic logic. In N. Peltier and V. Sofronie-Stokkermans, editors, *IJCAR 2020*, volume 12166 of *LNCS*, pages 432–453. Springer, 2020.
6. D. Basin, M. Gras, S. Krstić, and J. Schneider. Scalable online monitoring of distributed systems. In J. Deshmukh and D. Ničković, editors, *RV 2020*, LNCS. Springer, 2020. To appear.
7. D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods Syst. Des.*, 46(3):262–285, 2015.
8. D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.
9. D. Basin, F. Klaedtke, and E. Zălinescu. The MonPoly monitoring tool. In G. Reger and K. Havelund, editors, *RV-CuBES 2017*, volume 3 of *Kalpa Publications in Computing*, pages 19–28. EasyChair, 2017.
10. D. Basin, S. Krstić, and D. Traytel. AERIAL: almost event-rate independent algorithms for monitoring metric regular properties. In G. Reger and K. Havelund, editors, *RV-CuBES 2017*, volume 3 of *Kalpa Publications in Computing*, pages 29–36. EasyChair, 2017.
11. 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.
12. P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017.
13. 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.
14. G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In F. Rossi, editor, *IJCAI 2013*, pages 854–860. AAAI Press, 2013.
15. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In B. W. Boehm, D. Garlan, and J. Kramer, editors, *ICSE 1999*, pages 411–420. ACM, 1999.

16. Y. Falcone, S. Krstić, G. Reger, and D. Traytel. A taxonomy for classifying runtime verification tools. In C. Colombo and M. Leucker, editors, *RV 2018*, volume 11237 of *LNCS*, pages 241–262. Springer, 2018.
17. K. Havelund, D. Peled, and D. Ulus. First order temporal logic monitoring with BDDs. In D. Stewart and G. Weissenbacher, editors, *FMCAD 2017*, pages 116–123. IEEE, 2017.
18. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, 1990.
19. J. Li and K. Y. Rozier. MLTL benchmark generation via formula progression. In C. Colombo and M. Leucker, editors, *RV 2018*, volume 11237 of *LNCS*, pages 426–433. Springer, 2018.
20. W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.
21. H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018.
22. A. Pnueli. The temporal logic of programs. In *FOCS 1977*, pages 46–57. IEEE Computer Society, 1977.
23. M. Raszyk, D. Basin, S. Krstić, and D. Traytel. Multi-head monitoring of metric temporal logic. In Y. Chen et al., editors, *ATVA 2019*, volume 11781 of *LNCS*, pages 151–170. Springer, 2019.
24. M. Raszyk, D. Basin, and D. Traytel. Multi-head monitoring of metric dynamic logic. In D. V. Hung and O. Sokolsky, editors, *ATVA 2020*, volume 12302 of *LNCS*. Springer, 2020. To appear.
25. G. Reger. A report of RV-CuBES 2017. In G. Reger and K. Havelund, editors, *RV-CuBES 2017*, volume 3 of *Kalpa Publications in Computing*, pages 1–9. EasyChair, 2017.
26. 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.
27. J. Schneider, D. Basin, F. Brix, S. Krstić, and D. Traytel. Scalable online first-order monitoring. *Int. J. Softw. Tools Technol. Transf.*, 2020. To appear.
28. 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.
29. J. Schneider, D. Basin, S. Krstić, and D. Traytel. A formally verified monitor for metric first-order temporal logic. In B. Finkbeiner and L. Mariani, editors, *RV 2019*, volume 11757 of *LNCS*, pages 310–328. Springer, 2019.
30. D. Ulus. Timescales: A benchmark generator for MTL monitoring tools. In B. Finkbeiner and L. Mariani, editors, *RV 2019*, volume 11757 of *LNCS*, pages 402–412. Springer, 2019.