Stream Characteristics for First-Order Monitoring

Joshua Schneider and Srđan Krstić

Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland

Abstract

We present a benchmark suite for generating and reproducing streams of time-stamped parametric events. The benchmark generates a stream according to user-defined characteristics and reproduces it at a user-defined velocity. The characteristics relate to the frequencies of the different events and their data values. The benchmark also provides the expected result of monitoring the generated streams against a family of first-order temporal specifications. We envision the benchmark being used to attest the performance of online first-order monitors.

1 Introduction

We consider the *online monitoring problem*: Given a stream of time-stamped data, called events, and a property formulated in a specification language, determine whether the property is satisfied at every point in the stream. An *online monitor* is an algorithm that solves the online monitoring problem for some property. It processes an event stream, one event at a time, and produces a stream of verdicts, i.e., evaluations of the monitored property at every position in the event stream. Streams can contain simple atomic events, or more complex, parametric events, where each event has a number of data values.

The performance of an online monitor is often assessed in terms of its maximum memory usage and maximum latency over all processed events. The latency of processing a single event is the time difference between the moment the event appears in the stream until it has been fully processed by the monitor. Latency and memory usage of online monitors depend on two main factors: the complexity of the monitored property and the characteristics of the event stream. The benchmark suite presented in this paper focuses on the latter factor. It consists of three components: a stream GENERATOR, a stream REPLAYER, and an ORACLE. The main idea of the benchmark is to generate streams with characteristics that are particularly challenging for online monitoring. The GENERATOR generates a stream with user-defined characteristics and passes it to the REPLAYER that can feed it to a monitor at a user-defined velocity (i.e., number of events per second). The ORACLE provides the expected result (a stream of violations) for the generated stream and a specific property specified in metric first-order temporal logic (MFOTL) [2].

This benchmark was originally developed to assess the performance of our parallel online monitor [4], which is sensitive to the event stream characteristics.

2 Benchmark Description

Event Streams and Replay

We first recall some basic notions about event streams. An *event* is a tuple of data values that is labeled with an *event type*. Every event type R has an associated arity $\iota(R)$, which defines the number of data values for this type. We call $1, \ldots, \iota(R)$ the *attributes* of type R. We group a finite number of events that happen concurrently (from the event source's point of view) into *databases*. An (*event*) *stream* ρ 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 a time-stamp τ_i and its index in the stream i, also called a *time-point*. We assume discrete time-stamps, modeled as natural numbers, and allow event sources to have finer time granularity than the one used by time-stamps. Specifically, a stream may have the same time-stamp $\tau_i = \tau_j$ at different indices $i \neq j$. The sequence of time-stamps must be non-decreasing ($\forall i.\ \tau_i \leq \tau_{i+1}$) and always eventually increasing ($\forall \tau.\ \exists i.\ \tau < \tau_i$).

In the following, we define some stream characteristics. Fix a stream $\rho = (\tau_i, D_i)_{i \in \mathbb{N}}$. We define the *index rate* at time τ as the number of stream indices in one time unit, i.e., $|\{i \mid \tau = \tau_i\}|$. The *event rate* at time τ is defined as the total number of events in one time unit, i.e., $|\{e \in D_i \mid \tau = \tau_i\}|$. We call the rate of those events with type R the *relation rate* for R. The relative frequency of R is the ratio of its relation rate and the event rate. The relative frequency of a data value with respect to a specific attribute is the rate of events that carry this value for the attribute, divided by the relation rate.

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 reproduces an existing event stream (or finite event log) with an ingestion rate that is proportional to the stream's event rate. This allows us to simulate realistic, but reproducible workloads for online monitors, for example for latency measurements. The events with the initial time-stamp are all issued immediately. The subsequent events with the next time-stamp are delayed proportionally to the difference between the two time-stamps (which are interpreted as seconds). This process is repeated for each unique time-stamp in the stream. The *acceleration* of the stream, i.e., the inverse of the delay factor, is a parameter of the tool. For example, an acceleration of 2 will replay the stream twice as fast. This parameter can be used to generate workloads with different ingestion rates from the same data. The input of the REPLAYER is either a stream produced by another program or a finite log stored in a file. Input and output use the modified CSV format from the first RV competition [1]:

```
event type, tp = time-point, ts = time-stamp, attribute1 = value1, ...
```

Specification and Oracle

Our benchmark targets a family of specifications that are 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. They can be formalized using the following parametric MFOTL formula [2]:

$$\Box \forall \vec{x}.\ \left(\blacklozenge_{[0,w)}\, A(\vec{x}_A)\right) \wedge B(\vec{x}_B) \rightarrow \Box_{[0,w)}\, \neg C(\vec{x}_C),$$

where is w is a positive integer and $\vec{x}_A$, $\vec{x}_B$, and $\vec{x}_C$ are variable patterns. Informally, it 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 $\vec{x}_A$, $\vec{x}_B$, and $\vec{x}_C$, respectively. For example, if $\vec{x}_A = (x, y)$ and $\vec{x}_B = (y, z)$, 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.

The ORACLE provides the expected output of monitoring the above-mentioned family of specifications on a stream generated by the GENERATOR. It makes use of the MONPOLY monitoring tool [3], conveniently wrapping its invocation with the appropriate formula.

Stream Generation

The GENERATOR produces a random but reproducible event stream in the same format used by the REPLAYER. It generates output as quickly as possible, thus to tune the ingestion rate one must use the REPLAYER. The stream is intended to be monitored against the above-mentioned family of specifications. The variable patterns can be chosen freely by the user. There are also three built-in patterns: star ($\vec{x}_A = (w, x)$, $\vec{x}_B = (w, y)$, and $\vec{x}_C = (w, z)$), linear ($\vec{x}_A = (w, x)$, $\vec{x}_B = (x, y)$, and $\vec{x}_C = (y, z)$), and triangle ($\vec{x}_A = (x, y)$, $\vec{x}_B = (y, z)$, and $\vec{x}_C = (z, x)$).

Data values are chosen randomly and independently with 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 that match both a proceeding B event and an A event before that (both within appropriate intervals of length w). By default, values are sampled uniformly from the set $D = \{0, 1, \ldots, 10^9 - 1\}$. It is also possible to select a Zipf distribution per variable, which has the probability mass function $p(x) = x^{-z} / \sum_{n=1}^{10^9} n^{-z}$ for $x \in \{1, 2, \ldots, 10^9\}$. The larger the exponent $z > 0$ is, the fewer values have a large relative frequency. Events that form a violation are always drawn from the uniform distribution to prevent unintended matchings. For the same reason, 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 domain D is large.

Events with types A, B and C are generated randomly and independently according to user-specified relative frequencies p_A, p_B, and p_C. There are, however, some constraints: (1) the sum of all three frequencies must be 1; (2) p_A can be at most p_B; and (3) the relative frequency of violations can be at most the minimum of p_A and p_C.

3 User Guide

We provide the components of our benchmark as a Docker image [6] with all required dependencies installed. The components can also be built manually from the source repository [5] by running `mvn package` (needs Maven). In the following, we assume that Docker version 1.13 or higher is installed and configured properly. The components can be invoked with the command

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

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

The **generator** prints the generated stream to the standard output. It is invoked with

`generator {-S | -L | -T | -P` *`pattern`*`} [`*`options ...`*`] [`*`length`*`]`

If *`length`* is given, a finite log of that length (in seconds) is produced instead of an unbounded stream. It is required to select either a built-in or a custom variable pattern. The flags `-S` (star), `-L` (linear), and `-T` (triangle) select a built-in pattern. A custom pattern can be given after flag `-P`. The format is explained in the `README.txt` file in the source repository [5].

The relative frequencies of the 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.

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

```
generator -T -z "x=1.5,z=2"
```

generates events following the triangle pattern, with the values of variables x and z following a Zipf distribution with exponents 1.5 and 2. Variable y values are distributed uniformly.

The GENERATOR has the following additional options. `-e` *`rate`* and `-i` *`rate`* modify the event and index rates, which default to 10 and 1, respectively. 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 **replayer** reads events from standard input and copies them with a delay to standard output. It is invoked with

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

The most important option is the acceleration factor, which can be changed with option `-a` *`acceleration`* (default: 1). 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 such that the events can be replayed at the proper time. Use a shell redirection to read events from a file:

```
replayer -a 10 < test.csv
```

With option `-m` set, the component prints the output in MONPOLY's format [3]. Otherwise, the modified CSV format is used. Together with an acceleration of 0, which replays the stream as quickly as possible, the tool can thus function as a converter from CSV to MONPOLY format.

If option `-o` *`host:port`* is given together with a hostname and port, a TCP server listening to that address will be created. The first client connecting to the TCP server will receive the event stream. The first event is sent only once the client has connected. No more clients will be accepted afterwards.

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 a monitor that reads the event stream via a pipe or socket. The options `-v` (`-vv`) generate a compact (or verbose) report once every second, which is written to standard error. The format of the reports is explained in the accompanying `README.txt` file. Note that delays are tracked only up to the operating system's buffer that is associated with the pipe or socket.

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. If the queue is drained fully, an underrun occurs and events may not be reproduced at the appropriate time. The verbose report (`-vv`) displays the number of underruns. If this number is non-zero and especially if it is growing, the queue capacity should be increased with option `-q` *`size`* (default: 1024).

The **oracle** reads events in MONPOLY format from the standard input and prints a stream of violations to the standard output. It is invoked with

```
oracle {-S | -L | -T | -P pattern} [-w interval]
```

where the arguments define the specification as described for the GENERATOR. It assumes that the input stream is generated following the same specification. Finally, we give an example that combines all three components into a single command:

```
generator -S | replayer -m -a 10 | oracle -S
```

This generates a stream according to the star pattern, speeds it up 10 times and converts it to the MONPOLY format. The ORACLE then converts the stream to a stream of expected results.

Acknowledgment. Joshua Schneider is supported by the US Air Force grant "Monitoring at Any Cost" (FA9550-17-1-0306). Srđan Krstić is supported by the Swiss National Science Foundation grant "Big Data Monitoring" (167162).

References

[1] 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.

[2] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.

[3] D. Basin, F. Klaedtke, and E. Zalinescu. 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.

[4] 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.

[5] J. Schneider and S. Krstić. 2018 Runtime Verification Benchmark Challenge – FOStreams benchmark. https://github.com/runtime-verification/benchmark-challenge-2018.git, 2018.

[6] J. Schneider and S. Krstić. FOStreams benchmark (Docker image). https://hub.docker.com/r/infsec/benchmark/, 2018.