TIMESCALES Benchmark Suite

Submitted to Runtime Verification Benchmark Competition 2018

Dogan Ulus

Boston University
Boston, MA, USA
doganulus@gmail.com

Abstract

This document describes the benchmark suite TIMESCALES for Runtime Verification Benchmark Competition 2018. I present motivations behind these benchmarks and some details on the benchmark generation.

1 Introduction

Real-time systems are systems that need to satisfy some physical timing constraints for their correct operation. Metric Temporal Logic (MTL) is a popular formalism to specify temporal properties with timing constraints (called timed properties) over the behavior of real-time systems. Typical examples of timed properties have been studied in the works [3, 2], which extend the *pattern-scope* system developed by Dwyer et. al for untimed specifications [1]. In this system, a property specification consists of (1) a pattern, which describes what must be observed, and (2) a scope, which describes the temporal extent of the observation. For example, a property such that *an event* `p` *occurs at least every 10 seconds between events* `q` *and* `r` would be an instance of bounded recurrence pattern between two events. We consider these patterns and scopes to be a good representative of real world cases a runtime verification tool has to cope with. Therefore, the benchmark suite TIMESCALES contains a benchmark generator for a selection of such patterns and scopes.

Real-time systems consists of different components with different variables with different change rates. Continuously changing variables such as sensor readings usually employ a fixed step sampling with a small step size. On the other hand, higher-level processes often emit events at arbitrary time periods. Consequently, it is common to use a very small base time unit to express states and events uniformly on the timeline. This fact leads to large timing constraints (with respect to base time unit) in specifications. Ideally a real-time monitoring tool should handle large timing constraints as good as small ones. Hence, the purpose of this benchmark suite is to measure the performance and scalability of MTL monitoring tools with respect to varying timing constraints in the specification. Besides typical properties explained above, we include some worst cases (called challenges) in which large time bounds may degrade the performance of MTL monitoring significantly despite the structural simplicity of the specification.

The structure of this doucment as follows. Section 2 describes supported properties for the benchmark generation. Currently 10 typical properties and 2 challenges are available for the generation with different parameters. Section 3 presents two predefined sets of benchmarks generated through the interface. This section also give more details for customization of benchmark generation.

2 Supported Properties

This section describes the supported properties (10 typical properties, 2 challenges) in the benchmark suite. These properties contain one or two timing parameters in order to generate benchmarks with different timing characteristics. The final two properties are designed to be challenges for MTL monitoring as they attack the weak points of MTL monitoring algorithms [4] using large time bounds and some certain traces.

Bounded Absence After Q. This property takes one parameter `U`. Intuitively it means that it is always the case that the event P does not occur at least for `U` time units after the event Q occurs. The property is formally expressed as

```
once[<=U](q) -> (not(p) since q)
```

Bounded Absence Before R. This property takes one parameter `U`. Intuitively it means that it is always the case that the event P does not occur at least for u time units before the event Q occurs. The property is formally expressed as

```
r -> (always[<=U](not p))
```

Bounded Absence Between Q and R. This property takes two parameters `L` and `U`. Intuitively it means that it is always the case that the event P does not occur between events Q and R and the duration between Q and R is in l and u time units. The property is formally expressed as

```
(r && !q && once q ) -> ((not p) since[L,U] q)
```

Bounded Universality After Q. This property takes one parameter `U`. Intuitively it means that it is always the case that the event P always occurs at least for u time units after the event Q occurs. The property is formally expressed as

```
(once[<=U](q)) -> (p since q)
```

Bounded Universality Before R This property takes one parameter `U`. Intuitively it means that it is always the case that the event P always occurs at least for u time units before the event Q occurs. The property is formally expressed as

```
r -> (always[<=U](p))
```

Bounded Universality Between Q and R. This property takes two parameters `L` and `U`. Intuitively it means that it is always the case that the event P always occurs between events Q and R and the duration between Q and R is in l and u time units. The property is formally expressed as

```
(r && !q && once q) -> (p since[L,U] q)
```

Bounded Recurrence Globally. This property takes one parameter `U`. Intuitively it means that it is always the case that the event P occurs at least for every u time units. The property is formally expressed as

```
once[<=U](p)
```

Bounded Recurrence Between Q and R. This property takes two parameters `L` and `U`. Intuitively it means that it is always the case that the event P occurs at least for every u time units between events Q and R. The property is formally expressed as

```
(r && !q && once q) -> ((once[<=U](p or q)) since q)
```

Bounded Response Globally. This property takes two parameters `L` and `U`. Intuitively it means that it is always the case that the event S responds to the event P in l and u time units. The property is formally expressed as

```
(s -> once[L,U] p) && not( !s since[>=U] p)
```

Bounded Response Between Q and R. This property takes two parameters `L` and `U`. Intuitively it means that it is always the case that the event S responds to the event P in l and u time units between events Q and R. The property is formally expressed as

```
(r && !q && once q) -> (((s->once[L,U] p) and not( !s since[>=U] p)) since q)
```

Challenge PandQ This property takes two parameters `L` and `U` as well as has a structure such that

```
(once[>=L] p) -> (p since[L,U] q)
```

For a trace where `p` and `q` holds continuously, this property becomes a challenge when `U` is a large number such as 1000. This is due to some MTL monitoring algorithm explicitly store occurrences of `q` during the time window. In short. by this property and trace, we force the algorithm to store a long list of occurences that eventually slows down the monitoring.

Challenge Delay This property takes two parameters `L` and `U` as well as has a simple structure such that

```
(p since[L,U] q) or p
```

However, this property (or similar) is a challenge under two conditions: (1) the difference `L-U` is much smaller than `U` and (2) a trace where the `q` happens very frequently but not continuously. For example, we use parameters `L=U=1000` as well as a trace that `p` continuously holds and that `q` holds every other time point. In other words, the monitor of this property over such a trace should precisely remember the occurrence of a very frequent event in the far past and this is not an easy task.

3 Benchmark Generation and Usage

Timescales benchmark suite contains a benchmark generator called `reelay_benchgen`, originally developed for the monitoring tool Reelay[1]. The benchmark generator generates an input trace, as a standard CSV file, and a past MTL formula, as a standard YAML file, that satisfies the formula at every time point (except the last point by default). These output formats are simple human-readable text files and very well supported in virtually all major programming languages. The generator itself is written in Python and there is no dependency other than a Python3 installation.

[1] https://github.com/doganulus/reelay

The benchmark suite includes a Makefile to generate two predefined sets (small and large) of benchmarks using the generator. The command `make small` generates one benchmark for each supported properties (12 in total) with small timing bounds over short traces with a duration of 1000 time units. The small suite, which is included in the distribution, is intended for initial tests and demonstration purposes. On the other hand, the command `make large` generates three benchmarks for each property (36 in total) with increasingly larger time bounds (1x, 10x, and 100x) over traces with a duration of one million time units. The total size of the large benchmark is about 400MB and not included in the distribution. We ideally expect a constant time performance from MTL monitoring tools in processing these three benchmarks of each property. This indicates the monitoring algorithm does not rely on the naive discretization of timeline and robust against changes in the base time unit and sampling rates.

Benchmark Customization. It is possible to customize benchmark generation using the command line interface of the generator. Parameters of lower and upper time bounds in properties and the duration of trace is adjusted by options `-l, --lbound INT` for the lower bound, `-u, --ubound INT` for the upper bound, and `-d, --duration INT` for the duration. In addition, the recurrence chacteristics of traces for some properties can be adjusted by `--min-recur INT` and `--max-recur INT` options. For example, the benchmark generation procedure would pick a random number of recurrence for `p` between these values when generating a benchmark for the property bounded recurrence between `q` and `r`. By default we append a sequence to the trace that makes the property fail at the last time point. This is intended to be a sanity check in benchmarking monitoring algorithms. The option `--no-failing-end` disables this behavior. Generated traces may contain successive repeated values for events called stuttering. It is possible to limit the maximum duration of stuttering in traces by the option `--limit-stutter INT`. Choosing a large limit (such as larger than the total duration) would eliminate any stuttering in the trace while choosing zero means there is no limit on the stutterring.

References

[1] Matthew B Dwyer, George S Avrunin, and James C Corbett. “Property specification patterns for finite-state verification”. In: *Proceedings of the second workshop on Formal methods in software practice*. ACM. 1998, pp. 7–15.

[2] Volker Gruhn and Ralf Laue. “Patterns for timed property specifications”. In: *Electronic Notes in Theoretical Computer Science* 153.2 (2006), pp. 117–133.

[3] Sascha Konrad and Betty HC Cheng. “Real-time specification patterns”. In: *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 372–381.

[4] Dogan Ulus. “Online Monitoring of Metric Temporal Logic using Sequential Networks”. In: *Hybrid Systems: Computation and Control (HSCC)*. (Submitted). 2019. URL: https://github.com/doganulus/reelay/blob/master/docs/mtl_monitoring.pdf.