StreamLAB: Stream-based Monitoring of Cyber-Physical Systems

Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger(✉), Marvin Stenger, Leander Tentrup, and Hazem Torfah

Reactive Systems Group, Saarland University, Saarbrücken, Germany
{faymonville,finkbeiner,schledjewski,schwenger, stenger,tentrup,torfah}@react.uni-saarland.de

Abstract. With ever increasing autonomy of cyber-physical systems, monitoring becomes an integral part for ensuring the safety of the system at runtime. StreamLAB is a monitoring framework with high degree of expressibility and strong correctness guarantees. Specifications are written in RTLola, a stream-based specification language with formal semantics. StreamLAB provides an extensive analysis of the specification, including the computation of memory consumption and run-time guarantees. We demonstrate the applicability of StreamLAB on typical monitoring tasks for cyber-physical systems, such as sensor validation and system health checks.

1 Introduction

In stream-based monitoring, we translate input streams containing data collected at runtime, such as sensor readings, into output streams containing aggregate statistics, such as an average value, a counter, or the integral of a signal. Trigger specifications define thresholds and other logical conditions on the values on these output streams, and raise an alarm or execute some other predefined action if the condition becomes true. The advantage of this setup is great expressiveness and easy-to-reuse, compositional specifications. Existing stream-based languages like Lola [9,12] are based on the synchronous programming paradigm, where all streams are synchronized via a global clock. In each step, the new values of all output streams are computed in terms of the values of the other streams at a previous time step or. This paradigm provides a simple and natural evaluation model that fits well with typical implementations on

This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center "Foundations of Perspicuous Software Systems" (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300).

© The Author(s) 2019
I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 421–431, 2019.
https://doi.org/10.1007/978-3-030-25540-4_24

synchronous hardware. In real-time applications, however, the assumption that all data arrives synchronously is often simply not true. Consider, for example, an autonomous drone with several sensors, such as a GPS module, an inertia measurement unit, and a laser distance meter. While a synchronous arrival of all measured value would be desirable, some sensors' measurement frequency is higher than others. Moreover, the sensors do not necessarily operate on a common clock, so their readings drift apart over time.

In this paper we present the monitoring framework StreamLAB. We lift the synchronicity assumption to allow for monitoring asynchronous systems. Basis for the framework is RTLola, an extension of the steam-based runtime verification language Lola. RTLola introduces two new key concepts into Lola:

1. Variable-rate input streams: we consider input streams that extend at a-priori unknown rates. The only assumption is that each new event has a real-valued timestamp and that the events arrive in-order.
2. Sliding windows: A sliding window aggregates data over a real-time window given in units of time. For example, we might integrate the readings of an airspeed indicator.

As with any semantic extension, the challenge in the design of RTLola is to maintain the efficiency of the monitoring. Obviously, not all RTLola specifications can be monitored with constant memory since the rates of the input streams are unknown, an arbitrary number of events may occur in the span of a fixed real-time unit. Thus, for aggregations such as the mean requiring to store the whole sequence of value, no amount of constant memory will always suffice. We can, nevertheless, again identify an efficiently monitorable fragment that covers many specifications of practical interest. For the space-efficient aggregation over real-time sliding windows, we partition the real-time axis into equally-sized intervals. The size of the intervals is dictated by the rate of the output streams. For certain common types of aggregations, such as the sum or the number of entries, the values within each interval can be pre-aggregated and then only stored in this summarized form. In a static analysis of the specification, we identify parts of the specification with unbounded memory consumption, and compute bounds for all other parts of the specification. In this way, we can determine early, whether a particular specification can be executed on a system with limited memory.

Related Work. There is a rich body of work on monitoring real-time properties. Many monitoring approaches are based on real-time variants of temporal logics [3,11,16–18,24]. Maler and Nickovic present a monitoring algorithm for properties written in signal temporal logic (STL) by reducing STL formulas via a boolean abstraction to formulas in the real-time logic MITL [21]. Building on these ideas, Donze et al. present an algorithm for the monitoring of STL properties over continuous signals [10]. The algorithm computes the robustness degree in which a piecewise-continuous signal satisfies or violates an STL formula. Towards more practical approaches, Basin et al. extend metric logics with parameterization [8]. A monitoring algorithm for the extension is implemented in the tool MonPoly [5]. MonPoly was introduced as a tool for monitoring usage-control policies. Another extension to metric dynamic logic was implemented in

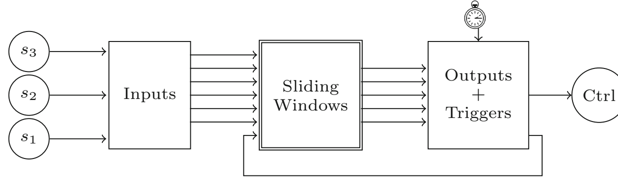

Fig. 1. Illustration of the decoupled input and output using aggregations.

the tool Aerial [7]. However, most monitors generated from temporal logics are limited to Boolean verdicts.

StreamLAB uses the stream-based language RTLola as its core specification language. RTLola builds upon Lola [9,12], which is a stream-based language originally developed for monitoring synchronous hardware circuits, by adding the concepts discussed above. Stream-based monitoring languages are significantly more expressive than temporal logics. Other prominent stream-based monitoring approaches are the *Copilot* framework [23] and the tool BeepBeep 3 [15]. Copilot is a dataflow language based on several declarative stream processing languages [9,14]. From a specification in Copilot, constant space and constant time C programs implementing embedded monitors are generated. The BeepBeep 3 tool uses an SQL-like language that is defined over streams of events. In addition to stream-processing, it contains operators such as slicing, where inputs can be separated into several different traces, and windowing where aggregations over a sliding window can be computed. Unlike RTLola, BeepBeep and Copilot assume a synchronous computation model, where all events arrive at a fixed rate. Two asynchronous real-time monitoring approaches are TeSSLa [19] and Striver [13]. TeSSLa allows for monitoring piece-wise constant signals where streams can emit events at different speeds with arbitrary latencies. Neither language provides the language feature of sliding windows and the definition of fixed-rate output streams. The efficient evaluation of aggregations on sliding windows [20] has previously been studied in the context of temporal logic [4]. Basin et al. present an algorithm for combining the elements of subsequences of a sliding window with an associative operator, which reuses the results of the subsequences in the evaluation of the next window [6].

2 Real-Time Lola

RTLola extends the stream-based specification languages Lola [12] with real-time features. In the stream-based processing paradigm, sensor readings are viewed as input streams to a stream processing engine that computes outputs in form of streams on top of the values of the input streams. For example, the RTLola specification

```
input altitude : Float32
output tooLow := altitude < 200.0
```

checks whether a drone flies with an altitude less than 200 feet. For each reading of the velocity sensor, a new value for the output stream *tooLow* is computed. Streams marked with the "**trigger**"-keyword alert the user when the value of the trigger is true. In the following example, the user is warned when the drone flies below the allowed altitude.

```
trigger tooLow "flying below minimum altitude"
```

Output streams in RTLola are computed from values of the input streams, other output streams and their own past values. If we want to count the number of times the drone dives below 200 feet we can specify the stream

```
output count := (if tooLow then 1 else 0)
  + count.offset(by:-1).defaults(to:0)
```

Here, the stream *count* computes its new values by increasing its latest value by 1 in case the drone currently flies below the permitted altitude. The expression `count.offset(by:-1)` represents the last value of the stream. We call such expressions "lookup expressions". The default operator `e.defaults(to:0)` returns the value 0 in case the value of `e` is not defined. This can happen when a stream is evaluated the first time and looks up its last value.

In RTLola, we do not impose any assumption on the arrival frequency of input streams. Each stream can produce new values individually and at arbitrary points in time. This can lead to problems when a burst of new input values occur in a short amount of time. Subsequently, the monitor needs to evaluate all output streams, exerting a lot of pressure on the system. To prevent that, RTLola distinguishes between two kinds of outputs. *Event-based* outputs are computed whenever new input values arrive and should thus only contain inexpensive operations. All streams discussed above where event-based. In contrast to that, there are *periodic* outputs such as the following:

```
output freqDev @5Hz := altitude.aggregate(over : 200ms,
                                           using: count) < 5
```

Here, *freqDev* will be evaluated every 200 ms as indicated by the "@ 5 Hz" label, independently of arriving input values. The stream *freqDev* does not access the event-based input *altitude* directly, but uses a *sliding window* expression to count the number of times a new value for *altitude* occurred within the last 200 ms. The value of *freqDev* represents the number of measurements the monitor received from the altimeter. Comparing this value against the expected number of readings allows for detecting deviations and thus a potentially damaged sensor.

Sliding windows allow for decoupling event-based and periodic streams, as illustrated in Fig. 1. Since the specifier has no control over the frequency of event-based streams, these streams should be quickly evaluatable. More expensive operations, such as sliding windows, may only be used in periodic streams to increase the monitor's robustness.

2.1 Examples

In the following, we will present several interesting properties showcasing RTLola's expressivity. The specifications are simplified for illustration and thus not immediately applicable to the real-world.

Sensor Validation. When a sensor starts to deteriorate, it can misbehave and drop single measurements. To verify that a GPS sensor produces values at its specified frequency, in this example 10 Hz, we count the number of sensor values in a continuous window and compare it against the expected amount of events in this time frame.

```
input lat: Float32, lon : Float32
output gps_freq@10Hz:=
  lat.aggregate(over: =1s, using: count).defaults(to:9)
trigger gps_freq < 9 "GPS sensor frequency < 9 Hz"
```

Assuming that we have another sensor measuring the true air speed, we can check whether the measured data matches the GPS data using RTLola's computation primitives. For this, we first compute the difference in longitude and latitude between the current and last measurement. The Euclidean distance provides the length of the movement vector, which can be derived discretely by dividing by the amount of time that has passed between two GPS measurements.

```
input velo : Float32
output δlon := lon - lon.offset(by:-1).defaults(to:lon)
output δlat := lat - lat.offset(by:-1).defaults(to:lat)
output gps_dist := sqrt(δlon * δlon + δlat * δlat)
output gps_velo := gps_dist
  / (time - time.offset(by:-1).defaults(to:0.0))
trigger abs(gps_velo - velo) > 0.1 "Deviating velocity"
```

When the pathfinding algorithm of the mission planner takes longer than expected, the system remains in a state without target location and thus hovers in place. Such a hover period can be detected by computing the covered distance in the last seconds. For this, we integrate the assumed velocity. We also exclude a strong headwind as a culprit for the low change in position.

```
input wnd_dir: Float32, wnd_spd : Float32
output dir := arctan(lat/lon)
output headwind := abs(wnd_dir - dir) < 0.2
  ∧ wnd_spd > 10.0
output hovering @ 1Hz := velo.aggregate(over: 5s, using: ∫)
  .defaults(to:0.5) < 0.5 ∧ ¬headwind.hold().defaults(to:⊥)
trigger hovering "Long hover phase"
```

3 Performance Guarantees via Static Analysis

3.1 Type System

RTLola is a strongly-typed specification language. Every expression has two orthogonal types: a value type and a stream type. The *value* type is `Bool`, `String`, `Int`, or `Float`. It indicates the usual semantics of a value or expression and the amount of memory required to store the value. The *stream* type indicates when a value is evaluated. For periodic streams, the stream type defines the frequency in which it is computed. Event-based streams do not have a predetermined period. The stream type for an event-based stream identifies a set of input streams, indicating that the event-based stream is extended whenever there is, synchronously, an event on all input streams. Event-based streams may also depend on input streams not listed in the type; in such cases, the type system requires an explicit use of the 0-order *sample&hold* operator.

The type system provides runtime guarantees for the monitor: Independently of the arrival of input data, it is guaranteed that all required data is available whenever a stream is extended. Either the data was just received as input event, was computed as output stream value, or the specifier provided a default value. The type system can, thus, eliminate classes of specification problems like unintentionally accessing a slower stream from a faster stream. Whenever possible, the tool provides automatic type inference.

3.2 Sliding Windows

We use two techniques to ensure that we only need a bounded amount of memory to compute sliding windows. Meertens [22] classifies an aggregations $\gamma\colon A^* \to B$ as list homomorphism if it can be split into a mapping function $m\colon A \to T$, an associative reduction function $r\colon T \times T \to T$, a finalization function $f\colon T \to B$, and a neutral element $\varepsilon \in T$ with $\forall t \in T\colon r(t, \varepsilon) = r(\varepsilon, t) = t$. For these functions, rather than aggregating the whole list at once, one can apply m to each element, reduce the intermediate results with an arbitrary precedence, and finalize the result to get the same value. The second technique by Li et al. [20] divides a time interval into panes of equal size. For each pane, we aggregate all inputs and store only the fix amount of intermediate values. The type system ensures that sliding windows only occur in periodic streams so by choosing the pane size as the inverse of the frequency, paning does not change the result. In StreamLAB there are several pre-defined aggregation functions such as count, integration, summation, product, mini-, and maximization available.

3.3 Memory Analysis

StreamLAB computes the worst-case memory consumption of the specification. For this, an annotated dependency graph (ADG) is constructed where each stream s constitutes a node v_s and whenever s accesses s', there is an edge from v_s to $v_{s'}$. Edges are annotated according to the type of access: if s accesses

s' discretely with offset n or with a sliding window aggregation of duration d and aggregation function γ, then the edge $e = (v_s, v_{s'})$ is labeled with $\lambda(e) = n$ or $\lambda(e) = (d, \gamma)$, respectively. Nodes of periodic streams are now annotated with their periodicity, if stream s has period 200 ms then the node is labeled with $\pi(v_s) = 5\,\text{Hz}$. Memory bounds for discrete-time offsets can be computed as for Lola [9]. We extend this algorithm with new computational rules to determine the memory bounds for real-time expressions. For each edge $e = (v, v')$ in the ADG we can determine how many events of v' must be stored for the computation of v using the rules in Fig. 2. Here, only γ is a list homomorphism. The strict upper bound on required memory is now the sum of the memory requirement of each individual stream. This, however, is only the amount of memory needed for storing values and does not take book-keeping data structures and the internal representation of the specification into account. Assuming reasonably small expressions (depth ≤ 64), this additional memory can be bounded with 1 kB per stream plus a flat 10 kB for working memory.

$\pi(v)$	$\pi(v')$	$\lambda(e) = (d, \gamma)$	$\lambda(e) = (d, \gamma^*)$
var	*var*	*unbounded*	zd
xHz	*var*	*unbounded*	$\min(zd, xd)$
var	yHz	yd	$\min(zd, yd)$
xHz	yHz	$\min(xd, yd)$	$\min(xd, yd)$

Fig. 2. Computation of memory bound over the dependency graph.

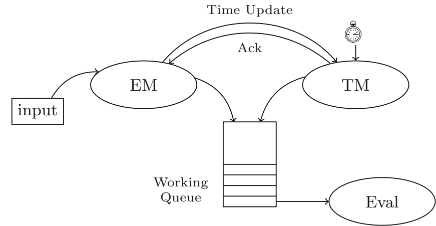

Fig. 3. Illustration of the data flow. The EM manages input events, TM schedules periodic tasks, and Eval manages the evaluation of streams.

4 Processing Engine

The processing engine consists of three components: The *EventManager (EM)* reads events from an input such Standard In or a CSV file and translates string values into the internal representation. The values are mapped to the corresponding input streams in the specification. Using a multiple-sender-single-receiver channel, the EM pushes the event on a working queue. The *TimeManager (TM)* schedules the evaluation of periodic streams. The TM computes the hyper-period of all streams and groups them by equal deadlines. Whenever a deadline is due, the corresponding streams are pushed into the working queue using the same channel as the EM. This ensures that event-based and periodic evaluation cycles occur in the correct order even under high pressure. Lastly, the *Evaluator (Eval)* manages the evaluation of streams and storage of computed values. The Eval repeatedly pops items off the working queue and evaluates the respective streams.

When monitoring a system online, the TM uses the internal system clock for scheduling tasks. When monitoring offline, however, this is no longer possible because the point in time when a stream is due to be evaluated depends on the input event. Thus, before the EM pushes an event on the working queue, it transmits the latest timestamp to the TM. The TM then decides whether some periodic streams need to be evaluated. If so, it effectively goes back in time by pushing the respective task on the working queue before acknowledging the TM. Only upon receiving the acknowledgement, the TM sends the event to the working queue. Figure 3 illustrates the information flow between the components.

5 Experiments

StreamLAB[1] is implemented in Rust. A major benefit of a Rust implementation is the connection to LLVM, which allows a compilation to a large variety of platforms. Moreover, the requirements to the runtime environment are as low as for C programs. This allows StreamLAB to be widely applicable.

The specifications presented in Sect. 2.1 have been tested on traces generated with the state-of-the-art flight simulator Ardupilot[2]. Each trace is the result of a drone flying one or more round-trips over Saarland University and provides sensor information for longitude and latitude, true air velocity, wind direction and speed, as well as the number of available GPS satellites. The longest trace consists of slightly less than 433,000 events. StreamLAB successfully detected a variety of errors such as delayed sensor readings, GPS module failures, and phases without significant movement. For an online runtime verification, the monitor reads an event of the simulator's output, processes the input data and pauses until the next event is available. Whenever necessary, periodic streams are evaluated. Online monitoring of a simulation did not allow us to exhaust the capabilities of StreamLAB because the generation of events took significantly longer than processing them. The offline monitoring function of StreamLAB allows the user to specify a delay in which consecutive events are read from a file. By gradually decreasing the delay between events until the pressure was too high, we could determine a maximum input frequency of 647.2 kHz. When disabling the delay and running the monitor at maximum speed, StreamLAB processes a trace of length 432,961 in 0.67 s, so each event takes 1545 ns to process while three threads utilized 146% of CPU. In terms of memory, the maximum resident set size amounted to 16 MB. This includes bookkeeping data structures, the specification, evaluator code, and parts of the C standard library. While the evaluation does not require any heap allocation after the setup phase, the average stack size amounts to less than 1kB. The experiment was conducted on 3.3 GHz Intel Core i7 processor with 16 GB2133 MHz LPDDR3 RAM.

[1] www.stream-lab.org.
[2] ardupilot.org.

6 Outlook

The stream-based monitoring framework StreamLAB demonstrates the applicability of stream monitoring for cyber-physical systems. Previous versions of Lola have successfully been applied to networks and unmanned aircraft systems in cooperation the with German Aerospace Center DLR [1,2,12]. StreamLAB provides a modular, easy-to-understand specification language and design-time feedback for specifiers. This helps to improve the development process for cyber-physical systems. Coupled with the promising experimental results, this lays the foundation for further applications of the framework on real-world systems.

References

1. Adolf, F.-M., Faymonville, P., Finkbeiner, B., Schirmer, S., Torens, C.: Stream runtime monitoring on UAS. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 33–49. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_3
2. Adolf, F., Faymonville, P., Finkbeiner, B., Schirmer, S., Torens, C.: Stream runtime monitoring on UAS. CoRR arXiv:abs/1804.04487 (2018). http://arxiv.org/abs/1804.04487
3. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. In: [1990] Proceedingsof the Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 390–401, June 1990. https://doi.org/10.1109/LICS.1990.113764
4. Basin, D., Bhatt, B.N., Traytel, D.: Almost event-rate independent monitoring of metric temporal logic. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 94–112. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_6
5. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: MONPOLY: monitoring usage-control policies. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 360–364. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_27
6. Basin, D., Klaedtke, F., Zălinescu, E.: Greedily computing associative aggregations on sliding windows. Inf. Process. Lett. **115**(2), 186–192 (2015). https://doi.org/10.1016/j.ipl.2014.09.009
7. Basin, D., Traytel, D., Krstić, S.: Aerial: almost event-rate independent algorithms for monitoring metric regular properties (2017). https://www21.in.tum.de/~traytel/papers/rvcubes17-aerial_tool/index.html
8. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015). https://doi.org/10.1145/2699444
9. D'Angelo, B., et al.: Lola: Runtime monitoring of synchronous systems. In: TIME 2005, pp. 166–174. IEEE Computer Society Press, June 2005
10. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 264–279. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_19
11. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_9. http://dl.acm.org/citation.cfm?id=1885174.1885183

12. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 152–168. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_10
13. Gorostiaga, F., Sánchez, C.: Striver: stream runtime verification for real-time event-streams. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 282–298. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_16
14. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language lustre. In: Proceedings of the IEEE, pp. 1305–1320 (1991)
15. Hallé, S.: When RV meets CEP. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 68–91. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_6
16. Harel, E., Lichtenstein, O., Pnueli, A.: Explicit clock temporal logic. In: LICS 1990, pp. 402–413. IEEE Computer Society (1990). https://doi.org/10.1109/LICS.1990.113765
17. Jahanian, F., Mok, A.K.L.: Safety analysis of timing properties in real-time systems. IEEE Trans. Softw. Eng. **SE-12**(9), 890–904 (1986). https://doi.org/10.1109/TSE.1986.6313045
18. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (1990). https://doi.org/10.1007/BF01995674
19. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Tessla: runtime verification of non-synchronized real-time streams. In: Haddad, H.M., Wainwright, R.L., Chbeir, R. (eds.) PSAC 2018, pp. 1925–1933. ACM (2018). https://doi.org/10.1145/3167132.3167338
20. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Rec. **34**(1), 39–44 (2005). https://doi.org/10.1145/1058150.1058158
21. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
22. Meertens, L.: Algorithmics: towards programming as a mathematical activity (1986)
23. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 345–359. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_26
24. Raskin, J.-F., Schobbens, P.-Y.: Real-time logics: fictitious clock as an abstraction of dense time. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 165–182. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0035387

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.