Extending RTLola with External Data Queries

Bernd Finkbeiner[1,2], Jakob Hirschler[3],
Frederik Scheerer(✉)[1], and Sebastian Schirmer[3]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{finkbeiner, frederik.scheerer}@cispa.de
[2] Technical University of Munich, Germany
[3] German Aerospace Center (DLR), Braunschweig, Germany
{jakob.hirschler, sebastian.schirmer}@dlr.de

Abstract. Stream-based monitoring enables the concise specification of complex temporal properties. However, existing stream-based monitors are limited when dealing with large external data sources, a task that is better handled by specialized data management systems. We address these limitations by extending stream-based monitors with the ability to query external data sources. We implement this approach in RTLola and investigate challenges such as handling delayed responses, type checking of returned data, and runtime error management. A unified interface enables the seamless integration of existing systems into our approach, such as static databases or dynamic endpoints, e.g., a weather API. Our evaluation shows that accessing data through custom data structures is beneficial, as it results in improved runtime performance. Our extension decouples the specification from the underlying data representation, allowing the data structure to be optimized independently of the specification.

Keywords: Stream-Based Monitoring · Cyber-Physical Systems · Databases

1 Introduction

Runtime monitoring is particularly important for safety-critical applications such as aviation, where violations of safety requirements may have severe consequences and timely reactions are needed. In practical monitoring scenarios, specifications often depend not only on the sensor data produced by the monitored system itself, but also on large external data sources. These sources include static geospatial data, such as terrain or obstacle information, as well as dynamically changing information, such as live weather data. One possible approach is to store all static information directly within the monitor, for example, obstacle locations used to determine which obstacles are currently close to the aircraft. However, stream-based monitoring systems are designed for the efficient evaluation of temporal properties over streams and are not optimized for storing and querying large amounts of data. In other situations, such as retrieving live weather information, the required data is not even available when the monitor is initialized and must instead be obtained dynamically through external services.

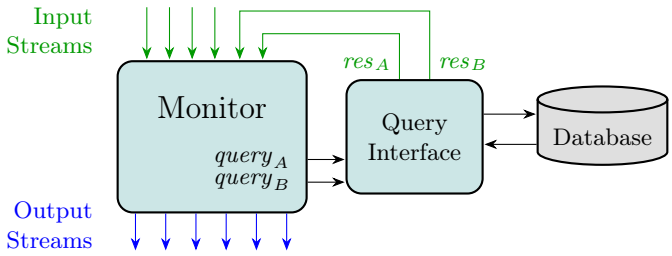

Fig. 1: Overview of query extension for stream-based monitors.

We therefore propose storing such information in dedicated external systems specifically designed and optimized for these tasks, and enabling the monitor to actively query these systems whenever additional information is required during runtime.
Querying external information from the monitor raises several challenges:

1. Queries are asynchronous and may exhibit significant latency. During this time, the monitor must not block and must continue the evaluation of unrelated monitoring tasks.
2. Queries may fail due to unavailable services or connectivity issues. The monitor must not crash and must allow the specification to detect when queries fail and define suitable fallback behavior.
3. The data format expected and returned by the queried system must be validated to ensure type safety and prevent runtime errors.
4. Different monitoring applications require querying a wide variety of backends, including general-purpose relational databases, specialized geospatial database systems, and external APIs. All of which may use different querying languages, communication protocols, and data formats.

In this paper, we present our approach for integrating external data sources into stream-based runtime monitoring. The overall architecture is illustrated in Figure 1. As shown in the figure, queries are issued directly from the monitor as part of stream equations. These queries are not executed synchronously, but are forwarded to a query interface, which acts as a communication layer between the monitor and external systems. The interface forwards queries to an external backend, for example, a database, and asynchronously returns responses back to the monitor through dedicated input streams. We have demonstrated our approach in an experimental evaluation using RTLola [5]. RTLola's pacing type system allows us to precisely define when new queries are triggered. To associate responses with the originating requests, each query returns a unique identifier, which is part of the results arriving at the inputs. The implemented interface is general and extensible, abstracting away the complexities of asynchronous external queries. In particular, it automatically handles issuing the queries according to the specification, associating the responses with the query identifier as well as

with the associated inputs, and error management, while encapsulating backend-specific communication and serialization details. We show the effectiveness of our approach on multiple monitoring scenarios from the aviation domain.

The rest of this paper is structured as follows: First, we introduce the RTLola query extension in Section 2. Then, we present implementation details and experimental results in Section 3.

1.1 Related Work

Monitoring systems often require spatial and temporal reasoning. This motivated logics like STREL [2,19], SaSTL [17], SSTL [20,21], and more recently STL-GO [23]. These extend temporal logic by operators over spatial objects, but do not address the underlying data sources, limiting scalability. We extend RTLola to use external data sources that efficiently store and retrieve only relevant data.

RTLola [5] is a stream-based language similar to TeSSLa [16] and Striver [14]. Unlike logics, it allows computations over input and output streams, enabling spatial computation such as the distance between two objects without introducing new operators. Traditionally, such languages rely on external components to provide all relevant input streams. In contrast, our approach follows an active monitoring paradigm [7], where the monitor actively queries external data sources to obtain the information needed.

Although all objects could be encoded as constant streams in stream languages, specialized systems like geospatial databases [22,1,12] are more efficient for storing and processing large-scale data. Instead of reimplementing these within the monitor, we directly leverage them. At the same time, we provide essential guarantees for safety-critical applications, including delayed responses, type-safe processing of returned data, and runtime error management.

2 Query-Extension to RTLola

RTLola [5] is a stream-based monitoring language that has been applied in domains including network monitoring [13], aviation systems [9,4], automotive applications [11] and algorithmic fairness [8,6]. Stream-based specifications define streams, which represent an infinite sequence of values. *Input streams* capture data produced by the monitored system, and *output streams* are defined through stream equations, filtering and aggregating the data. *Triggers* are special, boolean-valued, output streams that specify violations of the specification: whenever a trigger evaluates to true, the specification is considered violated. Consider the following specification from the aviation domain as an example.

```
1 input pos : (Float64, Float64)
2 constant obstacle_lat: Float64 := 249.301
3 constant obstacle_lon: Float64 := 23.453
4 output distance := √((obstacle_lat − pos.0)² + (obstacle_lon − pos.1)²)
5 output closer := distance < distance.last(or: distance)
6 trigger closer ∧ distance < 0.1 "Too close to the obstacle"
```

The specification defines one input stream, `pos`, representing the current position of the aircraft. Whenever a new GPS sensor reading arrives, a new tuple is added to this input stream. Based on these inputs, the output stream `distance` computes the distance between the current aircraft position and a fixed obstacle. Another output stream, `closer`, evaluates whether the newly computed distance is smaller than the previous one. Finally, a trigger issues a warning whenever the aircraft moves closer to the obstacle while the current distance is below 0.1. For a more detailed introduction to RTLola based on exactly this scenario, we refer the reader to the RTLola Tutorial [5].

While the specification above successfully monitors a single obstacle, real-world scenarios typically require keeping track of a large number of obstacles simultaneously. To handle this efficiently, in the following, we present our query extension, which addresses this challenge using the drone scenario as a running example. Since obstacle information is static and potentially very large, it is maintained in an external geospatial database. The monitor periodically queries this database to retrieve obstacles close to the current drone position. The specification in this section is intentionally simplified; more realistic specifications illustrating the interaction of temporal operators with database queries are presented in the evaluation section.

Consider the following specification, extending the example above to monitor multiple obstacles, and counting the number of nearby ones.

```
1 input pos : (Float64,Float64) as (lat,lon)
2 input obstacle : (QueryId,(Float64,Float64)) as (res_id,(o_lat,o_lon))
3 output obstacle_queries : QueryId @10s :=
4     query("NEARBY obstacles POINT ?1 ?2 100",
5           with: pos.hold(or: (0.0, 0.0)), into: obstacle)
6 output o_count @10s := obstacle.aggregate(over: 10s, using: count)
7 trigger o_count > 3 "Too many obstacles"
```

We initially assume that every query returns all results within ten seconds, so that queries can be processed independently without overlapping or interleaving results. In addition to the `pos` stream (Line 1), the specification defines an additional input stream `obstacle` (Line 2), which receives the results of issued queries. Notice the `as` expression in the stream declaration, which introduces syntactic sugar that allows tuple values to be destructured and named directly. For example, the latitude component of an obstacle can be accessed via `o_lat`. The output stream `obstacle_queries` (Line 3) is evaluated periodically every `@10s`. Whenever the `query` expression is evaluated, a new query is issued to the database backend. In this example, the backend is the geospatial database Tile38 [1], and the query string therefore corresponds to a Tile38 command. While the query string itself is static, it contains placeholders `?1` and `?2`, which are instantiated at runtime through the `with` expression (Line 5). In this example, the query retrieves all obstacles within 100m around the current drone position (Line 4). As specified by the `into` clause (Line 5), all results are streamed to the `obstacle` input stream. Finally, an output stream counts the obstacles

returned in the last ten seconds (Line 6), which is then checked by a trigger (Line 7).

To address type safety, our extension provides two variants of the query expression. The `query` expression statically checks that the query result matches the declared type of the `into` stream and that the placeholders are consistent with the `with` expression, rejecting the specification at compile time if either check fails. Where such static checking is not possible, for example, if the backend cannot report its schema in advance, the unchecked variant `query_unchecked` can be used instead. In this case, the user must handle runtime errors in the specification directly.

Note, however, that the current specification relies on the assumption that all query results arrive within 10 seconds and that a new query is only issued after the previous interval has completed. In time-critical domains, such a coarse interval is often insufficient. Furthermore, in the specification above, the monitor can only reason about the number of nearby obstacles once the entire 10-second interval has elapsed, even if the query itself completes within a fraction of a second. To address this, we introduce *query identifiers*, which allow issued queries to be associated with the arriving results. Each evaluation of the `query` expression returns a unique identifier, and every returned result is annotated with the identifier of the query that produced it. This allows the monitor to correlate results with their originating query regardless of when they arrive, decoupling the issuance of queries from the arrival of their results. In particular, responses may arrive out of order, e.g., a query issued later may return before an earlier one, since each response is matched to its originating query independently via its identifier rather than by arrival order. Using this mechanism, results can be processed independently of timing assumptions, allowing us to define a specification that counts the results associated with each issued query:

```
1 output count(qid)
2     spawn with obstacle_queries
3     eval @obstacle when result_id = qid with count(qid).last(or: 0)+1
```

This specification introduces a *parameterized stream*, consisting of multiple stream instances identified by the parameter `qid`. The `spawn` expression determines when new instances are created. In this example, a new instance is spawned for every query identifier produced by the `obstacle_queries` stream. The `eval` expression is evaluated whenever a new value arrives on the `obstacle` stream. A new output value is produced only if the identifier attached to the result matches the identifier of the corresponding stream instance. Each stream instance, therefore, maintains an independent count of the results of its associated query.

While query identifiers allow results to be associated with their corresponding queries, the monitor still lacks information about when a query is completed. To resolve this, we introduce a dedicated status stream that communicates status information for each query in the specification. Because this stream is shared between all queries, it is not associated with a query function, but instead globally annotated using RTLola's annotation mechanism [7]. The backend reports successful completion or query failures over this input stream:

```
#[query_status]
input status : (QueryId,QueryStatus) as (status_id,status)
trigger(qid)
    spawn with obstacle_queries
    eval @status status_id = qid ∧ status = QueryStatus::Finished
        ∧ count_per(qid).hold(or: 0) > 3 with "Too many obstacles"
    close @status when status_id = qid
```

The specification defines a parameterized trigger that is instantiated for every issued query. Whenever a status update arrives, the trigger checks whether the corresponding query has completed successfully. Only after completion does the specification check the number of obstacles from that query. Further, all stream instances concerning a given identifier can be closed whenever a query finishes.

3 Implementation and Experiments

This section presents our implementation and evaluation with specifications from the aviation domain. All specifications can be found in Section A.

3.1 Implementation

Our implementation extends the RTLola StreamIR interpreter [3] with a query interface and four backend integrations. The entire framework is implemented in Rust and uses `crossbeam` channels for non-blocking communication between the monitor and backend threads, enabling concurrent query execution without blocking the monitor.

Interface. To support a wide range of external systems, we provide a unified query interface that abstracts from backend-specific details, enabling the integration of databases and external APIs with minimal effort. The interface exposes four methods. *openDatabase* establishes a connection to the backend. *prepareQuery* is called once per query statement before monitoring begins, allowing backends to create prepared statements and validate result types against the declared streams. *runQuery* is executed whenever a query is evaluated during monitoring. It receives the current parameter values and two callback channels for streaming results and reporting status updates asynchronously. Finally, *finish* is called on monitor termination.

3.2 Experiments

Celltower Monitoring. As the first part of our evaluation, we consider a scenario in which a drone must periodically approach cell towers in order to transmit collected data. To ensure transmission, the drone must remain sufficiently close to a tower continuously for two minutes. The specification is satisfied if such a transmission period occurs at least once every four minutes. Consider the following specification:

(a) A valid flight (b) An invalid flight

Fig. 2: Traces of drones monitoring closeness to celltowers.

```
1 input pos : (Float64, Float64)
2 input ct_res : (QueryId, UInt64) as (_, in_range_count)
3 output queries @1s :=
4     query("NEARBY celltowers LIMIT 1 COUNT POINT ?1 ?2 700",
5         with: pos.hold(or: (0.0, 0.0)), into: ct_res)
6 output in_range := in_range_count > 0
7 output in_range_s @1s := in_range.aggregate(over: 2min, using: forall)
8 trigger @1s ¬in_range_s.aggregate(over: 4min, using: exists)
```

The monitor periodically issues queries through the output stream `queries` (Line 3), requesting the number of cell towers within a radius of 700 meters around the drone's current position. Query results are received asynchronously via the input stream `ct_res` (Line 2), from which the number of nearby towers is extracted as `in_range_count`. The stream `in_range` (Line 6) evaluates whether at least one tower is currently within range. Every second, `in_range_s` (Line 7) checks that the drone has remained continuously within range throughout the preceding 2-minute interval using a sliding window aggregation. Finally, the trigger (Line 8) verifies that at least one such uninterrupted interval has occurred within the last four minutes.

We generated synthetic flight traces that either satisfy or violate this property, as visualized in Figure 2. Cell towers are marked by white dots, each surrounded by a circle representing the threshold for a drone to be considered nearby. The points at which the drone was sufficiently long within this threshold are marked with a green dot, while violations are marked in red. In the left trace, the drone circles around each tower, thereby satisfying the specification. In contrast, the right trace omits circling one tower, resulting in a violation of the specification.

Powerline Monitoring. In this section, we compare the runtime of different database backends against an RTLola baseline without external queries. We implement an interface to SQLite3 [15], a general-purpose relational database system. In addition, we integrate specialized backends for geospatial queries, including the Tile38 database [1], as well as our own implementation based on

Table 1: Overview of the evaluated query backends and their average runtime on a trace issuing 190 queries.

Backend	Purpose	Query Language	Runtime		
			Small	Medium	Full
RTLola	Baseline	-	876ms	24.4s	-
SQLite3 [15]	General-purpose	SQL	1.04s	11.6s	250s
Tile38 [1]	Geospatial	Tile38 Commands	4.0ms	3.9ms	4.3ms
Our kd-tree	Geospatial	Tile38 Commands	2.8ms	2.7ms	3.0ms

k-d trees. A k-d tree [10] is a data structure that organizes geospatial points in k-dimensional space, enabling fast nearest-neighbor queries by recursively subdividing the search space along alternating axes.

We evaluate all backends using a specification that monitors the distance to the nearest obstacle. The full dataset consists of approximately 1.7 million powerline poles extracted from OpenStreetMap [4], with a small and medium subset of 10.000 and 100.000 poles, respectively. As a baseline, we compare against a pure RTLola implementation that uses parameterized streams to store and query all data internally. The runtimes were obtained on a system with a 13th Gen Intel Core i7-1355U with 200 runs for the geospatial backends, and 20 for SQLite3 and the RTLola baseline.

The runtimes for monitoring a trace with 190 queries are depicted in Table 1. For small datasets, the overhead of issuing external database queries dominates the runtime, leading to SQLite3 performing slightly worse than the pure RTLola baseline. However, the baseline scales poorly as the dataset size increases and is infeasible for the full dataset of 1.7 million entries, and is therefore omitted from the table. SQLite3 scales slightly better, but still requires 250s for the full database, making it unsuitable for real-time monitoring. In contrast, the geospatial backends achieve consistently low runtimes across all databases. Our own implementation consistently outperforms the other backends and achieves the best overall runtime.

Weather-Aware Monitoring. As the final part of our evaluation, we monitor a dynamic backend in the form of a weather API. In particular, we use the Bright Sky API [18] to query the current temperature at the drone's position. The specification tracks the delay between issuing a query and receiving the corresponding response. The results of a drone flight are shown in Figure 3. During the flight, we artificially vary the available bandwidth, which is depicted in blue in the graph, and at one point, the internet connection fails completely. This interval is shaded red. The scatter plot illustrates the issued triggers from the monitor throughout the flight. Green circles indicate that a response arrived within the expected time bound (less than 0.5s), while orange markers denote warnings is-

[4] `www.openstreetmap.org`

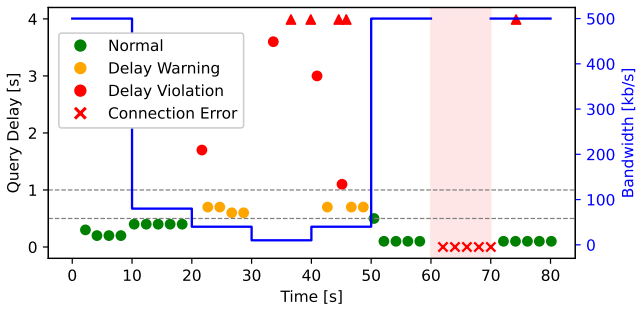

Fig. 3: Plot showing the monitored query delay when querying a weather API with varying connection bandwidth.

sued by the monitor for delays between 0.5s and 1.0s. Red circles indicate delay violations exceeding 1.0s, and points exceeding the upper limit of the figure are represented by red triangles. Finally, red crosses denote failed queries caused by the temporary loss of internet connectivity. These results demonstrate that the monitor correctly detects degraded API performance and gracefully handles complete connectivity loss without crashing.

4 Conclusion

We presented an extension to stream-based runtime monitoring that enables monitors to actively query external data sources at runtime. Our approach ensures asynchronous and type-safe queries, allows specifications to handle query failures gracefully, and provides a backend-agnostic interface that abstracts away the complexities of integration into the monitor. An evaluation on monitoring scenarios from the aviation domain confirms that the approach is practical and effective for applications that depend on both static and dynamic external data.

While our evaluation focused on the aviation domain, the approach is applicable to a wide range of systems that rely on external data sources. An interesting direction for future work would be the integration of large language models as a type of backend, enabling monitors to reason about unstructured data such as natural language.

Acknowledgments. This work was partially supported by the German Research Foundation (DFG) as part of TRR 248 (No. 389792660) and PreCePT (No. 521273327), and by the European Research Council (ERC) Grant HYPER (No. 101055412). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Baker, J.: Tile38: Ultra fast geospatial database & geofencing server. https://tile38.com (2026)
2. Bartocci, E., Bortolussi, L., Loreti, M., Nenzi, L.: Monitoring mobile and spatially distributed cyber-physical systems. In: Talpin, J., Derler, P., Schneider, K. (eds.) Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017. pp. 146–155. ACM (2017). https://doi.org/10.1145/3127041.3127050, https://doi.org/10.1145/3127041.3127050
3. Baumeister, J., Correnson, A., Finkbeiner, B., Scheerer, F.: An intermediate program representation for optimizing stream-based languages. In: Piskac, R., Rakamaric, Z. (eds.) Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III. Lecture Notes in Computer Science, vol. 15933, pp. 393–407. Springer (2025). https://doi.org/10.1007/978-3-031-98682-6_20
4. Baumeister, J., Finkbeiner, B., Kohn, F., Löhr, F., Manfredi, G., Schirmer, S., Torens, C.: Monitoring unmanned aircraft: Specification, integration, and lessons-learned. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II. pp. 207–218. Lecture Notes in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_10, https://doi.org/10.1007/978-3-031-65630-9_10
5. Baumeister, J., Finkbeiner, B., Kohn, F., Scheerer, F.: A tutorial on stream-based monitoring. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II. pp. 624–648. Lecture Notes in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_33, https://doi.org/10.1007/978-3-031-71177-0_33
6. Baumeister, J., Finkbeiner, B., Krsmanovic, V., Scheerer, F., Siber, J., Wagenpfeil, T.: Fairmon: A tool for monitoring and visualizing algorithmic fairness. In: Kauffman, S., Pedrielli, G. (eds.) The 26th International Conference on Runtime Verification, RV 2026, Kingston, Canada, October 6-9, 2026. Lecture Notes in Computer Science, Springer (2026)
7. Baumeister, J., Finkbeiner, B., Scheerer, F.: Active monitoring with RTLola: A specification-guided scheduling approach. In: Könighofer, B., Torfah, H. (eds.) Runtime Verification - 25th International Conference, RV 2025, Graz, Austria, September 15-19, 2025, Proceedings. pp. 181–201. Lecture Notes in Computer Science, Springer (2025). https://doi.org/10.1007/978-3-032-05435-7_11, https://doi.org/10.1007/978-3-032-05435-7_11
8. Baumeister, J., Finkbeiner, B., Scheerer, F., Siber, J., Wagenpfeil, T.: Stream-based monitoring of algorithmic fairness. In: Gurfinkel, A., Heule, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I. Lecture Notes in Computer Science, vol. 15696, pp. 60–81. Springer (2025). https://doi.org/10.1007/978-3-031-90643-5_4
9. Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., Torens, C.: RTLola cleared for take-off: Monitoring autonomous aircraft. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020,

Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. pp. 28–39. Lecture Notes in Computer Science, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_3, https://doi.org/10.1007/978-3-030-53291-8_3
10. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975). https://doi.org/10.1145/361002.361007, https://doi.org/10.1145/361002.361007
11. Biewer, S., Finkbeiner, B., Hermanns, H., Köhl, M.A., Schnitzer, Y., Schwenger, M.: RTLola on board: testing real driving emissions on your phone. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 365–372. Springer (2021)
12. Breunig, M., Bradley, P.E., Jahn, M., Kuper, P.V., Mazroob, N., Rösch, N., Al-Doori, M., Stefanakis, E., Jadidi, M.: Geospatial data management research: Progress and future directions. ISPRS Int. J. Geo Inf. **9**(2), 95 (2020). https://doi.org/10.3390/IJGI9020095, https://doi.org/10.3390/ijgi9020095
13. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: International Conference on Runtime Verification. pp. 152–168. Springer (2016)
14. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: International Conference on Runtime Verification. pp. 282–298. Springer (2018)
15. Hipp, D.R.: SQLite. https://www.sqlite.org (2026)
16. Kallwies, H., Leucker, M., Schmitz, M., Schulz, A., Thoma, D., Weiss, A.: Tessla– an ecosystem for runtime verification. In: International Conference on Runtime Verification. pp. 314–324. Springer (2022)
17. Ma, M., Bartocci, E., Lifland, E., Stankovic, J.A., Feng, L.: Sastl: Spatial aggregation signal temporal logic for runtime monitoring in smart cities. In: 11th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2020, Sydney, Australia, April 21-25, 2020. pp. 51–62. IEEE (2020). https://doi.org/10.1109/ICCPS48487.2020.00013, https://doi.org/10.1109/ICCPS48487.2020.00013
18. de Maeyer, J.: Bright Sky: JSON API for DWD open weather data. https://brightsky.dev (2026)
19. Nenzi, L., Bartocci, E., Bortolussi, L., Loreti, M.: A logic for monitoring dynamic networks of spatially-distributed cyber-physical systems. Log. Methods Comput. Sci. **18**(1) (2022). https://doi.org/10.46298/LMCS-18(1:4)2022, https://doi.org/10.46298/lmcs-18(1:4)2022
20. Nenzi, L., Bortolussi, L.: Specifying and monitoring properties of stochastic spatio-temporal systems in signal temporal logic. In: Haviv, M., Knottenbelt, W.J., Maggi, L., Miorandi, D. (eds.) 8th International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2014, Bratislava, Slovakia, December 9-11, 2014. ICST (2014). https://doi.org/10.4108/ICST.VALUETOOLS.2014.258183, https://doi.org/10.4108/icst.valuetools.2014.258183
21. Nenzi, L., Bortolussi, L., Ciancia, V., Loreti, M., Massink, M.: Qualitative and quantitative monitoring of spatio-temporal properties. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9333, pp. 21–37. Springer (2015). https://doi.org/10.1007/978-3-319-23820-3_2, https://doi.org/10.1007/978-3-319-23820-3_2
22. PostGIS Development Group: PostGIS. https://postgis.net (2025)

23. Zhao, Y., Yu, X., Hoxha, B., Fainekos, G., Deshmukh, J., Lindemann, L.: STL-GO: spatio-temporal logic with graph operators for distributed systems with multiple network topologies. ACM Trans. Embed. Comput. Syst. **24**(5s), 134:1–134:23 (2025). https://doi.org/10.1145/3760258, https://doi.org/10.1145/3760258

A Specifications

A.1 Powerline Monitoring

RTLola Baseline

```
import math

input lat : Float64
input lon : Float64

input time_str : String

input powerline_lat : Float64
input powerline_lon : Float64

output powerline_distance(pl_lat, pl_lon)
  spawn with (powerline_lat, powerline_lon)
  eval @lat&&lon with sqrt((pl_lat - lat)**2.0 + (pl_lon - lon)**2.0)

output closest_powerline :=
    powerline_distance.aggregate(over_instances: fresh, using:
    min).defaults(to: 0.0)
```

SQLite

```
import query

input close_powerlines : (QueryId, (Float64,Float64,Float64))
  as (query_id, (pl_lat,pl_lon,pl_distance))

#[query_status]
input query_status : (QueryId, QueryStatus) as (result_id,
    result_status)

input lat : Float64
input lon : Float64

input time_str : String

output query_close_powerlines := (time_str, query_unchecked("
SELECT
  lat,
  lon,
  ROUND(haversine(lat, lon, ?1, ?2), 1) as distance
  FROM static_objects
  ORDER BY distance
  LIMIT 1;
", with: (lat,lon), into: "close_powerlines"))
```

Tile38

```
import query

input close_powerlines : (QueryId, (Float64,Float64,Float64))
  as (query_id, (pl_lat,pl_lon,pl_distance))

#[query_status]
input query_status : (QueryId, QueryStatus) as (result_id,
    result_status)

input lat : Float64
input lon : Float64

input time_str : String

output query_close_powerlines := (time_str, query_unchecked("NEARBY
    static_objects DISTANCE LIMIT 1 POINT ?1 ?2", with: (lat,lon),
    into: "close_powerlines"))
```

A.2 Weather-Aware Monitoring

```
input lat : Float64
input lon : Float64
input time : Float64
#[query_status]
input status: (QueryId, QueryStatus) as (status_id, status_msg)
input temperature : (QueryId, Float64)
#[public]
output q @2s := query_unchecked(query: "", with: (lat.hold(or: 0.0),
    lon.hold(or: 0.0)), into: "temperature")
output q_time(qid)
  spawn with q
  eval when q == qid with time.hold(or: 0.0)
  close when status_id = qid
#[public]
output query_delay
  eval @status with time.hold(or: 0.0) - q_time(status_id).hold(or:
      0.0)
trigger query_delay > 0.5 "slight delay"
trigger query_delay > 1.0 "long delay"
trigger status_msg == QueryStatus::Error "query error"
```