Leveraging Static Analysis: An IDE for RTLola

Bernd Finkbeiner[0000−0002−4280−8441], Florian Kohn[0000−0001−9672−2398], and Malte Schledjewski[0000−0002−5221−9253]

CISPA Helmholtz Center for Information Security
66123 Saarbrücken, Germany
`{finkbeiner, florian.kohn, malte.schledjewski}@cispa.de`

Abstract. Runtime monitoring is an essential part of guaranteeing the safety of cyber-physical systems. Recently, runtime monitoring frameworks based on formal specification languages gained momentum. These languages provide valuable abstractions for specifying the behavior of a system. Yet, writing specifications remains challenging as, among other things, the specifier has to keep track of the timing behavior of streams. This paper presents the RTLola Playground, a browser-based development environment for the stream-based runtime monitoring framework RTLola. It features new methods to explore the static analysis results of RTLola, leveraging the advantages of such a formal language to support the developer in writing and understanding specifications. Specifications are executed locally in the browser, plotting the resulting stream values, allowing for intuitive testing. Step-wise execution based on user-provided system traces enables the debugging of identified errors.

Keywords: Integrated Development Environment · Runtime Monitoring · Static Analysis · Visualization.

Acknowledgements. This work was supported by the European Research Council (ERC) Grant HYPER (No. 101055412), by DFG grant 389792660 as part of TRR 248, and by the Aviation Research Programm LuFo of the German Federal Ministry for Economic Affairs and Energy as part of “Volocopter Sicherheits-Technologie zur robusten eVTOL Flugzustands- Absicherung durch formales Monitoring” (No. 20Q1963C).

1 Introduction

Cyber-physical systems have become an essential part of our everyday lives. Being safety-critical, their failure threatens humans and the environment. Consequently, new methods are needed to ensure their correct and safe behavior. While synthesizing or verifying such systems based on logics is an active field of research, applying these approaches to more extensive systems is computationally infeasible. Runtime verification techniques provide scalability by monitoring the system’s behavior at runtime. This methodology has proven to be applicable in many real-world scenarios [14,18]. In Runtime Verification, a monitoring

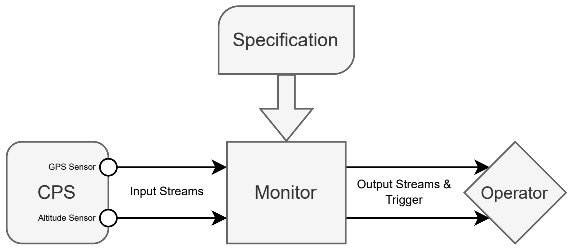


Fig. 1: The stream-based Monitoring Approach

component is deployed alongside the system, observing it and producing verdicts about its health and conformity. Such monitors can be realized through conventional programming or generated automatically from a specification given in a formal specification language. One class of formal specification languages adequate for such a task are stream-based specification languages. Pioneered by Lola [15], they process incoming data as streams from which new streams can be computed. Trigger conditions can be defined to assess the system's state and notify an operator in case of a violation. This stream-based monitoring approach is summarized in Figure 1. One stream-based specification language is RTLola [17]. It features real-time capabilities paired with a strong type system. Other such languages are, for example, TeSSLa [21], and Striver [19].

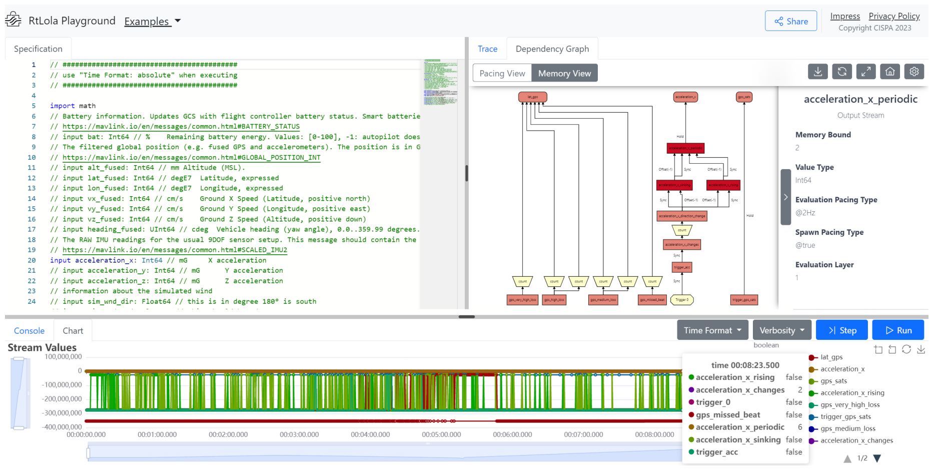
While stream-based specification languages provide useful abstractions to model the behavior of cyber-physical systems, writing correct specifications and reasoning about them is equally crucial as it is challenging [16]. Especially concerning autonomous aircraft, understanding the specification is essential with regard to certification and regulation conformity [23].

This paper presents the RTLola Playground[1]. A new web-based integrated development environment for RTLola. It eases the process from adopting runtime verification techniques to writing and testing specifications. It is based on the RTLola Framework extended with new static analyses that are then visualized by the tool. For example, directed graph-based analysis results can be explored interactively, similar to Evonne [13], a tool for visualizing proof trees generated by automated reasoning methods. Taking inspiration from other "playground"-style web-based IDEs for programming languages [20,4], the RTLola Playground executes specifications directly in the browser based on user-provided system traces. Monitor verdicts and intermediate values are plotted in graphs to assess the specification's correctness visually. To ease the debugging of specifications, a method for their step-wise execution is included. Other web-based tools for formal methods take a similar approach. The stream-based specification language TeSSLa also features a playground [21] where users can quickly test specifications. Yet, it does not aid the specifier in understanding static analysis results.

[1] RTLola Playground: https://rtlola.org/playground

The rest of this paper is organized as follows: Section 2 presents motivating examples highlighting the benefits of the RTLola Playground. Following, Section 3 presents an overview of the RTLola specification language. Section 4 gives the main points of the existing RTLola toolchain and its library structure. In Section 5, we present the web-based IDE for RTLola and briefly overview the tool's architecture. Section 6 reviews the RTLola Playground from a users perspective before Section 7 concludes the paper.

2 Writing Specifications is Hard



```
// ##########################################
// use "Time Format: absolute" when executing
// ##########################################

import math
// Battery information. Updates GCS with flight controller battery status. Smart batterie
// https://mavlink.io/en/messages/common.html#BATTERY_STATUS
// input bat: Int64 // %    Remaining battery energy. Values: [0-100], -1: autopilot does
// The filtered global position (e.g. fused GPS and accelerometers). The position is in G
// https://mavlink.io/en/messages/common.html#GLOBAL_POSITION_INT
// input alt_fused: Int64 // mm Altitude (MSL).
// input lat_fused: Int64 // degE7  Latitude, expressed
// input lon_fused: Int64 // degE7  Longitude, expressed
// input vx_fused: Int64 // cm/s    Ground X Speed (Latitude, positive north)
// input vy_fused: Int64 // cm/s    Ground Y Speed (Longitude, positive east)
// input vz_fused: Int64 // cm/s    Ground Z Speed (Altitude, positive down)
// input heading_fused: UInt64 // cdeg  Vehicle heading (yaw angle), 0.0..359.99 degrees.
// The RAW IMU readings for the usual 9DOF sensor setup. This message should contain the
// https://mavlink.io/en/messages/common.html#SCALED_IMU2
input acceleration_x: Int64 // mG     X acceleration
// input acceleration_y: Int64 // mG      Y acceleration
// input acceleration_z: Int64 // mG      Z acceleration
// information about the simulated wind
// input sim_wnd_dir: Float64 // this is in degree 180° is south
```

acceleration_x_periodic
Output Stream
Memory Bound
2
Value Type
Int64
Evaluation Pacing Type
@2Hz
Spawn Pacing Type
@true
Evaluation Layer
1

```
time 00:08:23.500
● acceleration_x_rising      false
● acceleration_x_changes         2
● trigger_0                  false
● gps_missed_beat            false
● acceleration_x_periodic        6
● acceleration_x_sinking     false
● trigger_acc                false
```

Fig. 2: Screenshot of the RTLola playground. The left panel contains the specification editor, the right panel the dependency graph and the trace editor, and the bottom panel contains the output of the interpreter as the CLI output and a plot.

Writing specifications poses similar challenges as programming in general. Large specifications do not fit into the human working memory and small errors such as simple typos or copy&paste errors can creep in. The RTLola Playground tackles these problems on several fronts.

As depicted in the screenshot of the user interface in Figure 2 it is divided into three sections. The left panel features a rich text editor for specifications. The right panel contains the editor for traces and the dependency graph, a static analysis result of the RTLola framework. The visualization of static analysis results is complemented by the integration of an interactive execution of the monitor on a user-provided trace. The bottom panel features either a plot or a textual representation of the resulting stream values allowing for a quick exploration of the specification's behavior. Just like the dependency graph, the plot also allows zooming and hiding uninteresting streams.

RTLola is a language with a rich type system that is already used to check the specification prior to execution but showing the inferred types directly in line with the specification further improves the feedback loop. Some of the simple copy&paste errors such as forgetting to change the accessed stream can already have an influence on the inferred pacing type and therefore more easily spotted as seen in Figure 3a. The RTLola framework already uses another static analysis artifact: the dependency graph. It consists of all streams, sliding windows, and accesses. A more complete definition of the dependency graph is given in Definition 1. The same error as described above would lead to a different edge in the graph which can break the symmetry between copied parts as seen in Figure 3b.

Other simple errors such as accessing a stream with a wrong offset cannot be spotted by checking the inferred types. A wrong offset does not change the inferred type but it changes the thickness of the edge when viewing the dependency graph in *memory view* mode and potentially the buffering requirement of the accessed stream which leads to a different color as seen in Figure 4. Similarly, a mismatch in a periodic pacing type leads to different color in the *pacing view* mode.

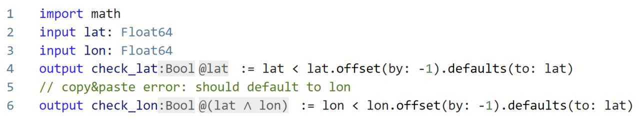
To tackle the aspect of cognitive overload, the RTLola Playground allows for merging connected nodes in the dependency graph to hide currently uninteresting parts as is demonstrated in Figure 5. This could be augmented on the language level by adding a module system. Some preliminary exploration has been done in this direction.

3 The RTLola specification Language

In this section, we give an overview of the RTLola specification language. An RTLola specification consists of input streams, representing sensor reading of the system, output streams, representing internal computations and trigger conditions, constituting an assessment of the system's health. Furthermore, RTLola distinguishes streams by their timing behavior. This timing behavior is part of RTLola's type system and is called the pacing type of a stream. There are two disjunct timing variants: Event-based streams are evaluated in an ad-hock manner whenever the streams they depend on produce a new value. Periodic streams produce values at a fixed frequency. The specification in Listing 1.1 is used as a running example throughout this section and monitors abrupt altitude changes of an autonomous aircraft.

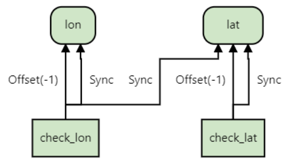
```
input altitude : Float

output avg_altitude @1Hz :=
      altitude.aggregate(over: 1min, using: avg)

output altitude_diff :=
      abs(altitude - avg_altitude.hold(or: altitude))

trigger altitude_diff > 10.0 "Altitude changed too quickly"
```

Listing 1.1: RTLola: A running Example.



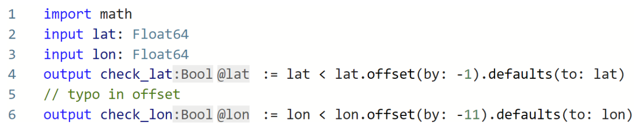
```
import math
input lat: Float64
input lon: Float64
output check_lat:Bool @lat  := lat < lat.offset(by: -1).defaults(to: lat)
// copy&paste error: should default to lon
output check_lon:Bool @(lat ∧ lon)  := lon < lon.offset(by: -1).defaults(to: lat)
```

(a) Specification containing an erroneous access from *check_lon* to *lat*.



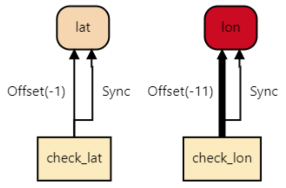
(b) Dependency graph with the additional edge.

Fig. 3: A specification with a typical copy&paste error in the form of an access from *check_lon* to *lat* instead of *lon*. This error can be spotted in the editor due to a change in the inferred pacing type of the accessing stream. Likewise, the dependency graph shows an additional edge width breaks the symmetry and therefore can also be spotted easily.



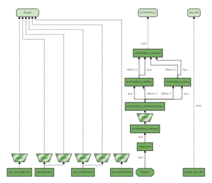
```
import math
input lat: Float64
input lon: Float64
output check_lat:Bool @lat  := lat < lat.offset(by: -1).defaults(to: lat)
// typo in offset
output check_lon:Bool @lon  := lon < lon.offset(by: -11).defaults(to: lon)
```

(a) Specification containing an error in the offset of the access from *check_lon* to *lat*.

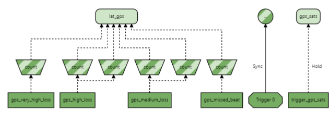


(b) Dependency graph in *memory view* mode.

Fig. 4: A specification with a typical typo in the offset of a stream access. This error does not change the inferred pacing type but the different offset changes the required memory of the accessed stream and therefore its color in the *memory view* mode. In addition, the corresponding edge in the dependency graph is thicker.



(a) Fully expanded dependency graph.



(b) Dependency graph with the parts relevant only for Trigger 0 merged.

Fig. 5: The dependency graph of the same specification. Once fully expanded and once a large part of it merged to better focus on the rest.

In this specification's first line, the input stream `altitude` is declared. As for all input streams, only its value type, `Float` in this case, is known. Distinctly, RTLola does not pose any assumptions on the timing behavior of input streams, i.e. the time when a new sensor reading arrives at the monitor remains unknown till runtime.

As the name suggests, the output stream `avg_altitude` declared in line 3 computes the average as a sliding window over the `altitude` input stream. A sliding window accumulates all values of the target stream in the given time frame. In the example above, this time frame is one minute. Because sliding windows do not imply any timing for their evaluation, the stream has to be annotated with an explicit frequency of `1Hz`, inducing that a new stream value, and therefore for the window, is computed every second. Note that sliding windows must have a periodic timing to have bounded memory. We refer the interested reader to [17] for more details on sliding windows.

The following output stream in line 6 computes the difference between the average altitude and the currently measured one. It highlights an essential part of the RTLola type system: Periodic and event-based streams must not be accessed synchronously in the same expression. The `altitude` access in the stream's expressions constitutes a synchronous access. A synchronous access reads the target stream's current value and additionally binds the accessing stream's timing to the accessed stream's timing. This guarantees that the accessing stream is only evaluated if the accessed value exists. As events can never be assumed to happen with a fixed frequency the type-checking procedure fails if a stream accesses both a periodic and an event-based stream synchronously, as no common timing can be determined in which both accessed values are always guaranteed to exist.

To resolve this, the stream in line 6 of Listing 1.1 uses a `hold` access to the timed average stream. A `hold` access refers to the last computed value of a stream. If no such value exists, a provided default value is substituted. Last, a `trigger` is defined to alert the operator if the current altitude deviates more than ten units from its average.

4 The RTLola Framework

The RTLola framework is split into two purviews. The RTLola Frontend is responsible for parsing and analyzing specifications. A Backend handles the event input, executes the specification, and forwards the output to the user. Executing a specification can follow different paradigms. The specification can be interpreted by the RTLola Interpreter or cross-compiled to a programming or hardware description language by the RTLola Compiler.

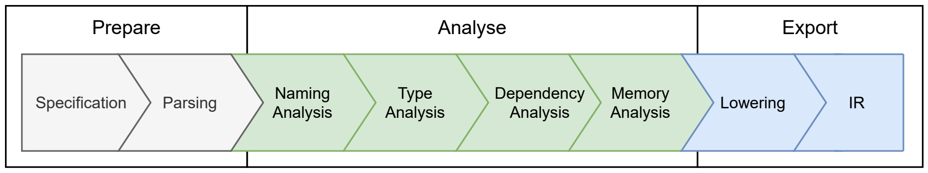


Fig. 6: An overview of the RTLola Frontend

4.1 The RTLola Frontend

The RTLola Frontend[2] is divided into multiple phases consisting of multiple stages. Figure 6 depicts an overview of these stages. In the first phase, the specification is parsed into an abstract syntax tree. The AST is then transformed into its de-sugarized form by representing all syntactic sugar constructs by basic RTLola expressions.

Next, the **naming analysis** checks the AST for duplicated or undefined stream names. For example, this stage rejects all specifications in which the same stream is defined multiple times. Afterward, the AST is transformed into a high-level intermediate representation by replacing stream name occurrences with numerical ids based on the previous analysis.

The **type analysis** infers a value and a pacing type for every stream. The value type of a stream determines the semantics of produced values. The value type system is similar to the one of programming languages. Consequently, RTLola also supports the usual value types, such as signed and unsigned integers, floats, strings, and booleans, including combinations of those types through tuples.

The pacing type of a stream determines the temporal behavior of a stream, i.e., when a new stream value is computed. As described in Section 3, there are two classes of pacing types. An event-based type (e.g. `@(lat ∧ lon)` in Figure 3a) signals that the stream is computed whenever an event occurs. An event is a combination of input streams receiving a new value synchronously. Such a combination is described through a positive boolean formula over input streams. A synchronous access from one event-based stream to another is allowed iff the accessing stream's pacing type implies the pacing of the accessed stream.

A periodic type (`@1Hz`) indicates that a stream is computed at a fixed frequency. A synchronous access from one periodic stream to another is allowed iff the accessing stream's frequency divides the frequency of the accessed stream. A synchronous access from an event-based stream to a periodic stream or vice versa is not allowed.

The **dependency analysis** computes the dependency graph of the specification and checks its well-formedness as presented in [15]. The dependency graph of a specification is defined as follows:

[2] RTLola Frontend: https://crates.io/crates/rtlola-frontend

Definition 1. *The dependency graph of a specification with inputs $i_1, ..., i_m$ and outputs $o_1, .., o_n$ is a directed weighted multi-graph $G = \langle V, E \rangle$ with $V = \{i_1, ..., i_m, o_1, ..., o_n\}$. An edge $e = \langle o_i, o_k, w \rangle$ is in E iff the expression of o_i contains o_k.offset(by: w, or: c) as a sub-expression or $e = \langle o_i, i_k, w \rangle$ if i_k.offset(by: w, or: c) is a sub-expression. Synchronous accesses are reflected as offsets by 0.*

To recap, the dependency graph of a not well-formed specification contains a cycle with an accumulated weight of 0. As a result, specifications such as:

```
output a:= b
output b:= a
```

are rejected.

The **memory analysis** computes a per stream upper bound for the number of values that must be stored as defined in [17]. Intuitively, if the maximal offset a stream is accessed with is 2, then three values must be stored for that stream, including the current value.

After the high-level intermediate representation is validated through the static analyses, it is lowered into the final intermediate representation, dropping information irrelevant to backends.

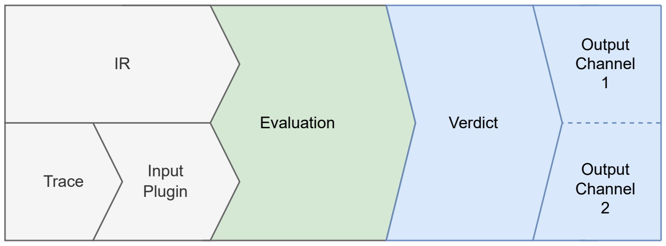


Fig. 7: An overview of the RTLola Interpreter

4.2 The RTLola Interpreter

The RTLola Interpreter[3] is an interpreter for RTLola specifications. Developed for the rapid prototyping of specifications, it forms the basis for evaluating specifications in the RTLOLA PLAYGROUND. Figure 7 shows an overview of the interpreter architecture. The specification can be processed directly in the form of its intermediate representation. To handle a variety of trace formats, the interpreter adds a layer of indirection through input plugins that translate events from their trace representation to an internal representation. This way, the interpreter can accept events in various formats like CSV, network packet capture (PCAP), or serialized as bytes.

[3] RTLola Interpreter: https://crates.io/crates/rtlola-interpreter

Each event starts a new evaluation cycle in which all periodic streams up to the current point are evaluated before all event-based streams are evaluated that were activated by the event. Which information the produced verdict contains is up to configuration and ranges from trigger messages to the current state of all streams. The verdict is forwarded to one or multiple output channels responsible for displaying or forwarding that information.

5 Tool Overview

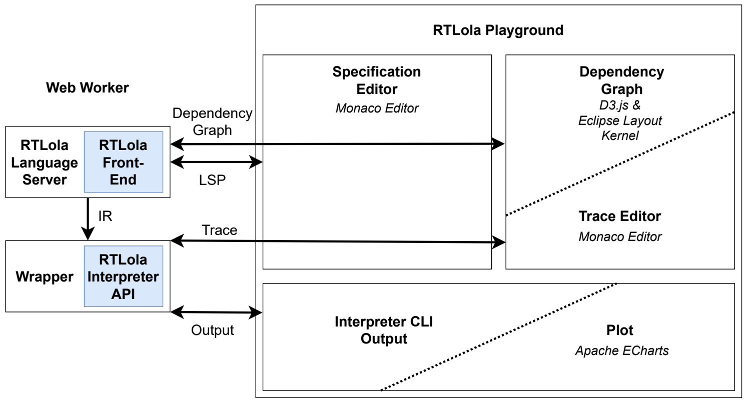


Fig. 8: Simplified overview of the main software components and their interaction. The blue boxes are deployed as WebAssembly compiled from Rust code. The other parts are implemented in TypeScript. Web workers from third-party libraries are not shown.

The RTLola Playground is a progressive web application based on the *Vue* [11] framework and in general written in *TypeScript* [9]. An overview of the main components and their interaction is shown in Figure 8. The components communicate mostly via shared *Pinia* [7] stores.

The RTLola framework is implemented in the *Rust* [8] language. This allows for easy compilation to *WebAssembly* [12] which means, that the code running in the browser matches the code powering the RTLola Interpreter executable.

The text editing is provided by the *Monaco Editor* [5] which is extracted from *Visual Studio Code* [10]. For the specification editor, we also provide a language server for RTLola which is mostly a wrapper written in TypeScript and Rust around the RTLola Frontend. This ensures that the user gets the same errors as if they were using the RTLola interpreter executable. The language server mostly communicates with the specification editor via the *Language Server Protocol* [6] to enable inline hints and diagnostics but it also provides additional artifacts such as the dependency graph and the intermediate representation of the specification.

The dependency graph is mostly based on the *D3.js* [2] library while the layout is handled by the *Eclipse Layout Kernel (elkjs)* [3] guided by information from the static analysis. A thin wrapper written in TypeScript and Rust around the RTLola interpreter API allows for executing monitors directly in the browser. One can inspect the CLI output as if one were to use RTLola interpreter executable but in addition the playground also contains a plot of all scalar numerical and boolean stream values. The plot is based on *Apache ECharts* [1] which provides the typical interactions such as hover, filtering, and zooming.

6 Application Scenarios

This section reviews the benefits of the RTLola Playground by considering two usage scenarios.

Firstly, new users of RTLola can quickly test their mental model about stream-based specification languages. They can run specifications and try them against different traces without interacting with a complicated command line interface or dealing with an installation process. Additionally, we plan to integrate an interactive tutorial directly into the RTLola Playground to lower the entry barrier further. There have been many studies on how and when to give feedback during learning [24,22]. More elaborate feedback than simple right/wrong improves learning and in the case of the RTLola Playground we believe that for type errors, showing the expected and the actual type strikes a good balance between enough information and feedback complexity. For learning basic programming skills immediate feedback seems to work best for beginners. Immediate feedback can be detrimental if it leads to simply gaming the system until a correct answer is found but as we do not have a given task this is not the case for the RTLola Playground.

Secondly, expert users of RTLola can use the RTLola Playground to get better insights into the specification's memory consumption, timing behavior, or locality. Exemplary, expert users can use the dependency graph to identify possible optimizations. In Figure 5, one can see that the `count` aggregation is repeated five times with an identical duration. This can be optimized by outsourcing this aggregation into a separate stream.

These application scenarios show, that the RTLola Playground is not only suitable for users of different knowledge backgrounds but can also be a step towards a wide adoption of runtime verification techniques.

7 Conclusion

This paper presented the RTLola Playground, a web-based integrated development environment for the stream-based specification language RTLola. Built with cutting-edge web technologies like WebAssembly and web workers, it features a rich text editor for specifications, integrated testing and debugging capabilities, and interactive visualizations for static analysis results. We have demonstrated how specific specification errors can be identified using either the editor's feedback or the static analysis results. We elaborated on how different user groups can use the playground to their advantage and hence conclude that the RTLola Playground helps specifiers to write correct specifications faster while keeping the entry barrier for new users low.

In the future, we plan to reuse most of the components of the tool in an extension for the Visual Studio Code editor and integrate an interactive tutorial into the playground.

Lastly, we encourage other community members to port their research tools to the browser. Many modern compiler toolchains support a compilation to WebAssembly, which keeps the overhead feasible. A web-based tool enables easy adoption and makes research easier to reproduce and transfer.

References

1. Apache echarts. https://echarts.apache.org/en/index.html, accessed: 2023-05-08
2. D3.js – data-driven documents. https://d3js.org/, accessed: 2023-05-08
3. Github – kieler/elkjs: Elk's layout algorithms for javascript. https://github.com/kieler/elkjs, accessed: 2023-05-08
4. The go playground. https://go.dev/play/, accessed: 2023-05-08
5. Monaco editor. https://microsoft.github.io/monaco-editor/, accessed: 2023-05-08
6. Official page for language server protocol. https://microsoft.github.io/language-server-protocol/, accessed: 2023-05-08
7. Pinia — the intuitive store for vue.js. https://pinia.vuejs.org/, accessed: 2023-05-08
8. Rust programming language. https://www.rust-lang.org/, accessed: 2023-05-08
9. Typescript: Javascript fith syntax for types. https://www.typescriptlang.org/, accessed: 2023-05-08
10. Visual studio code – code editing. redefined. https://code.visualstudio.com/, accessed: 2023-05-08
11. Vue.js the progressive javascript framework. https://vuejs.org/, accessed: 2023-05-08
12. Webassembly. https://webassembly.org/, accessed: 2023-05-08
13. Alrabbaa, C., Baader, F., Borgwardt, S., Dachselt, R., Koopmann, P., Méndez, J.: Evonne: Interactive proof visualization for description logics (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning. pp. 271–280. Springer International Publishing, Cham (2022)
14. Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., Torens, C.: Rtlola cleared for take-off: monitoring autonomous aircraft. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32. pp. 28–39. Springer (2020)
15. d'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME'05). pp. 166–174. IEEE (2005)
16. Dauer, J.C., Finkbeiner, B., Schirmer, S.: Monitoring with verified guarantees. In: Runtime Verification: 21st International Conference, RV 2021, Virtual Event, October 11–14, 2021, Proceedings 21. pp. 62–80. Springer (2021)

17. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: Streamlab: stream-based monitoring of cyber-physical systems. In: Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. pp. 421–431. Springer (2019)
18. Friese, M.J., Kallwies, H., Leucker, M., Sachenbacher, M., Streichhahn, H., Thoma, D.: Runtime verification of autosar timing extensions. In: Proceedings of the 30th International Conference on Real-Time Networks and Systems. p. 173–183. RTNS 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3534879.3534898, https://doi.org/10.1145/3534879.3534898
19. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: Colombo, C., Leucker, M. (eds.) Runtime Verification. pp. 282–298. Springer International Publishing, Cham (2018)
20. Goulding, J.: The rust playground. https://play.rust-lang.org, accessed: 2023-05-08
21. Kallwies, H., Leucker, M., Schmitz, M., Schulz, A., Thoma, D., Weiss, A.: Tessla–an ecosystem for runtime verification. In: Runtime Verification: 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28–30, 2022, Proceedings. pp. 314–324. Springer (2022)
22. der Kleij, F.M.V., Feskens, R.C.W., Eggen, T.J.H.M.: Effects of feedback in a computer-based learning environment on students' learning outcomes: A meta-analysis. Review of Educational Research **85**(4), 475–511 (2015). https://doi.org/10.3102/0034654314564881, https://doi.org/10.3102/0034654314564881
23. Schirmer, S., Torens, C.: Safe operation monitoring for specific category unmanned aircraft. In: Automated Low-Altitude Air Delivery - Towards Autonomous Cargo Transportation with Drones. Springer (November 2021), https://elib.dlr.de/145080/
24. Shute, V.J.: Focus on formative feedback. Review of Educational Research **78**(1), 153–189 (2008). https://doi.org/10.3102/0034654307313795, https://doi.org/10.3102/0034654307313795