An Intermediate Program Representation for Optimizing Stream-Based Languages

Jan Baumeister(✉), Arthur Correnson, Bernd Finkbeiner, and Frederik Scheerer

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{jan.baumeister, arthur.correnson, finkbeiner, frederik.scheerer}@cispa.de

Abstract. Stream-based runtime monitors are safety assurance tools that check at runtime whether the system's behavior satisfies a formal specification. Specifications consist of stream equations, which relate input streams, containing sensor readings and other incoming information, to output streams, representing filtered and aggregated data. This paper presents a framework for the stream-based specification language RTLola. We introduce a new intermediate representation for stream-based languages, the StreamIR, which, like the specification language, operates on streams of unbounded length; while the stream equations are replaced by imperative programs. We present a set of optimizations based on static analysis of the specification and have implemented an interpreter and a compiler for several target languages. In our evaluation, we measure the performance of several real-world case studies. The results show that the new StreamIR framework reduces the runtime significantly compared to the existing RTLola interpreter. We evaluate the effect of the optimizations and show that significant performance gains are possible beyond the optimizations of the target language's compiler. While our current implementation is limited to RTLola, the StreamIR is designed to accommodate other stream-based languages, enabling their interpretation and compilation into all available target languages.

Keywords: Runtime Verification · Stream-based Monitoring · Real-time Properties · Intermediate Representation

1 Introduction

Frameworks for runtime monitoring must strike a careful balance between safety and performance. On the one hand, runtime monitors are responsible for raising alarms about potential risks and initiating mitigation procedures in real time; their safety and correctness are, therefore, critical. On the other hand, monitors are often run on platforms with very limited resources, such as the onboard computer of an unmanned aircraft, where efficiency is crucial.

To ensure the correctness of the monitor, a common approach is to follow the paradigm of *specification-based monitoring*, where the monitor is not programmed in a programming language but rather based on a formal specification. Many frameworks store the specification in memory and then *interpret* the specification against the incoming data at runtime [15]; this, however, causes substantially sub-optimal performance. The alternative approach is to *translate* the specification into a target programming language, such as Rust, and then rely on a standard compiler for the target language to produce executable code [17,24,9]. Even though translated monitors often perform significantly better than an interpreter, compilers for general-purpose programming languages do not optimize based on the specific properties of the specification language. This approach therefore still leaves potential for optimization on the table.

In this paper, we present an optimization framework for the class of stream-based specification languages [13,12,19]. Such specifications consist of stream equations, which relate input streams, containing sensor readings and other incoming information, to output streams, representing filtered and aggregated data. Stream-based monitoring has been applied to the safety assurance of cyber-physical systems, including monitoring of exhaust emissions in cars [10] and the safety of unmanned aircraft [4,8]. Stream-based languages have properties that are relevant for code optimization but are not found in general-purpose programming languages. For example, the value of a particular expression in a given time step of a stream-based monitor is guaranteed to be the same, regardless of other stream evaluations in between. As a result, if-statements can easily be combined or moved outside or inside other statements.

Our framework introduces a new intermediate representation, the StreamIR, specifically designed to represent the relational stream-based specifications in an imperative form and enable optimizations before the code is translated to the target language. We provide an interpreter and a compiler to Rust and Solidity. The optimizations are implemented as a set of rewrite rules that exploit the specific properties of stream-based languages. As the source language, our framework accepts RTLola [15,5] specifications, which is a famous representative of the class of stream-based specification languages, but could easily be extended to support other stream-based languages like TeSSLa [12] or Striver [19].

We evaluate our approach on a set of specifications from RTLola case studies [4,7,5,31]. Interpreting the optimized StreamIR code significantly outperforms the state-of-the-art RTLola interpreter [15]. This is perhaps not surprising, because the RTLola interpreter does not attempt to optimize the specification before the execution. However, our framework also outperforms the direct compilation of RTLola into Rust, demonstrating that the StreamIR optimizations gain an additional speed-up on top of the optimizations already made by the Rust compiler. The situation is similar in our experiments with monitoring smart contracts in Solidity. Here, we assess our compiler by measuring the gas usage. The StreamIR optimizations again achieve substantial performance gains beyond the optimizations of the Solidity compiler.

The rest of this paper is structured as follows: After giving an overview on RTLola in Section 2, Section 3 introduces the intermediate representation, describes the translation from RTLola specifiations to the StreamIR, and presents the StreamIR optimizations. Section 4 provides details about the implementation and the evaluation of our approach.

Related Work. Runtime monitoring is a dynamic verification approach that has been applied to a variety of domains [22,21], and there are several monitoring tools and case studies [27,8,4]. Monitoring with stream-based specifications was pioneered by Lola [13], which was later extended to Lola2.0 [14] and RTLola [15,5]. Stream-based specification languages are related to synchronous programming languages such as Lustre [20], Esterel [11] and Signal [18]. In contrast to them, stream-based specifications are descriptive languages that use stream equations to describe temporal properties. Other extensions of Lola are TeSSLa [12], Striver [19] or Copilot [28]. Several compilers for stream-based languages exist, including compilers for Lola [17] and TeSSLa [24] to Rust. Other more specific compilers are a compilation [9] from RTLola to the hardware description language VHDL [9] or a compilation from Copilot [29] to the Atom language, a domain-specific language for embedded hard real-time applications. While some of these tools employ an intermediate representation (IR) to support different target languages, all are built for their specific specification language.

The concept of compiler construction through an IR is well-established. The most well-known IR is LLVM, the basis for most modern compilers. However, to the best of our knowledge, a general-purpose IR specifically designed for stream-based specifications does not yet exist. Optimizations of stream-based specifications have been studied before. In both RTLola [6] and TeSSLa [23], there exist approaches that describe optimizations specifically designed for their specification language. This paper follows this idea, but, compared to the other approaches, the optimizations are defined on the more general StreamIR, which describes an imperative program, not a set of equations. This approach is complementary to the optimizations based on the specification language.

2 RTLola

An RTLola specification defines a set of streams, each representing an infinite sequence of values. We differentate between *input streams* which capture external data, and *output streams* that perform computations based on current and past stream values. Consider the specification in Figure 1 which monitors a waypoint mission of an autonomous drone. The drone is provided with new waypoints through an input stream and must reach each waypoint within 10 seconds. We have exemplified an evaluation of the specifiation in Figure 2.

The specification introduces two input streams: `pos`, representing the drone's current position, and `wp`, representing the position of new waypoints. The output stream `moving` determines whether the drone is currently in motion by comparing its current position to the previous one. This is accomplished using an offset lookup – a temporal operator in RTLola that accesses past stream values. The accesses are represented in Figure 2 through a series of dashed arrows. Since previous values don't exist initially, indicated by the question mark in the top

```
input pos : (UInt64, UInt64)
input wp : (UInt64, UInt64)
output moving
  eval @pos with pos != pos.offset(by: -1, or: pos)
output reached(current)
  spawn @wp with wp
  eval @pos with dist(pos, current) < ε
  close @pos when reached(current)
output missed_wp(current)
  spawn @wp with wp
  eval @Local(10s) with ¬reached(current).aggregate(over: 10s, using: ∃)
  close @pos when reached(current)
```

Fig. 1: RTLola specification checking whether a drone reaches new waypoints within 10 seconds.

left of the figure, the offset operator requires a default value. The output stream `reached` checks for each waypoint whether it was reached by the drone. In contrast to the previous output stream, this stream is parameterized to describe a set of streams, called *stream instances* – each tracking a specific waypoint.

Instances are created using the *spawn* clause, where the expression computes the *parameter* of the instance. Similarly, the *close* clause defines when instances are removed. In this example, a new instance is spawned for every new waypoint and closed once the corresponding waypoint is reached. The *evaluation* of parameterized streams is applied to every instance, in our example, if the distance between the current position and the waypoint, represented with the instance's parameter, is smaller than a threshold. The figure depicts two instances of the `reached` stream, each respectively monitoring whether waypoint $(5, 3)$ and $(7, 6)$ have been reached. The output stream `missed_wp` tracks whether a waypoint was missed – that is, not reached within a time bound. This stream is also parameterized over the waypoints, but in contrast to the previous streams, the evaluation is applied at a frequency. More precicely, each stream instance is evaluated every 10 seconds and checks if the waypoint was not reached within the last 10 seconds.

Note that for each clause – spawn, eval, and close – RTLola has two types of filters: pacings and dynamic filter. The *pacing*, specified with an `@`-symbol, describes whether the clause is event-based or periodic. For *event-based* clauses, the clause is evaluated when a new input arrives, such as the evaluation of `moving`, which receives new values for every new value of `pos`. *Periodic* clauses are evaluated with a fixed frequency, such as the evaluation of `missed_wp` that is executed every 10 seconds. *Dynamic filters* are boolean stream expressions and follow after the `when`-keyword. They describe a condition based on stream values; here, we close a stream instance upon reaching the waypoint.

From a specification, we can derive a *Dependency Graph* describing the relation between streams. Analysis on this graph returns the required memory and a partial order, indicating which evaluation steps – called *Tasks* – have to be

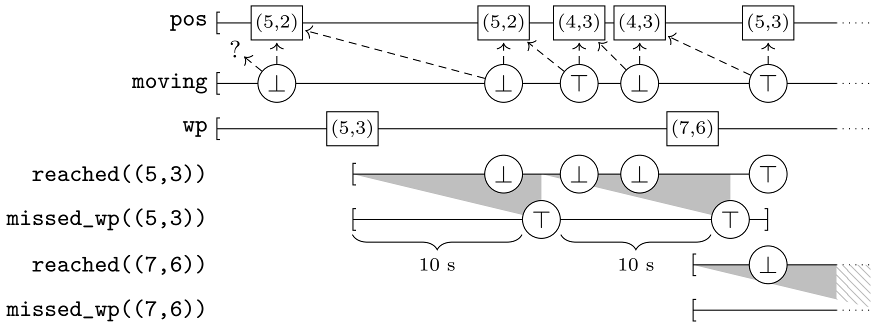


Fig. 2: An evaluation of the RTLola specification from Figure 1.

done in which order. These tasks either spawn, shift, evaluate, or close stream instances and correspond to the clauses in the specification. Here, *shifting* an instance reserves space for a new value. The order can be transformed into a list of *layers*, where the layers are evaluated sequentially and all tasks of one layer in parallel. For a more detailed explanation, we refer to the RTLola tutorial [5].

3 Stream Intermediate Representation (StreamIR)

The StreamIR, a new intermediate representation for stream-based specification languages, captures the imperative semantics of those languages and describes operations specific to real-time stream-based monitors, such as spawning, evaluating, and closing a stream instance. Besides typical imperative statements, such as conditionals in the form of `if`-`then`-`else`, it includes stream-based specific statements or expressions. The *guards* in conditions, for example, extend standard logical operators with constructs to check the presence of inputs to model event-based streams or constructs to reason about deadlines of periodic streams. Using this representation, a StreamIR monitor `loop{`*stmt*`}` describes a statement that is executed for each input event. The complete syntax is defined in Appendix A.

Example 1. The program in Figure 3 describes a monitor for the specification from Figure 1. The program is split into three parts following the three pacings of the specification, `@pos`, `@wp`, and `@Local(10s)`. First, the program handles the input `wp`, and spawns the parameterized output streams. Then, it shifts and evaluates the `pos` input stream and the non-parameterized `moving` output stream before iterating over the instances of the `reached` stream. Since the stream instances of `missed_wp` are closed with the same condition as the `reached` stream instances, the monitor handles the closing of these instances in parallel. The last guard iterates over the `missed_wp` output stream to evaluate its instances. Given that the deadline is local, i.e. relative to the spawn of each instance, the guard checks the deadline of each instance individually.

```
if input? wp then
  shift wp ; input wp ; (spawn reached (syn wp) || spawn missed_wp (syn wp)) ;
if input? pos then
  shift pos ; input pos ; shift moving ;
  eval moving (syn pos ≠ get pos −1 (syn pos)) ;
  iterate reached
    shift reached ; eval reached dist(syn pos, self) < ε ;
    if dynamic syn reached self then
      close reached || close missed_wp ;
iterate missed_wp
  if local 10s missed_wp then
    shift missed_wp ; eval missed_wp ¬(window reached self 10s ∃)
```

Fig. 3: A StreamIR program of the example specification in Figure 1.

StreamIR monitors are defined over a sequence of *Memories*. Each memory contains the already computed values – called *prefix* – of each stream instance and the next *deadline* for periodic stream instances, the timestamp of the next stream evaluation according to the frequency. We differentiate between global frequencies that start with the beginning of the monitor and local frequencies that start with the spawn of a stream instance.

Using the memory, we define inference rules to evaluate expressions and guards, i.e., $(M, \mathit{expr}) \Downarrow_t^{inst} \mathit{val}$, and statements, i.e., $(M, \mathit{stmt}) \Downarrow_{I,D,t}^{inst} M'$. A step of the monitor $((M, D), \mathit{stmt}) \Rightarrow_{I,t} (M', D')$ describes a sequence of statement executions, one execution for each passed deadline and one to process the input. For a given input sequence, *a monitor* then describes the sequence of memories according to the inference rules while starting with an initial memory. The list of all inference rules is given in Appendix B.

Some programs are ill-formed, i.e., the correctness of the program relies on constraints imposed on the input. To avoid such programs, we only consider *well-defined* programs, i.e., each infinite sequence of inputs has a unique output.

3.1 Translation from RTLola to StreamIR

The translation from an RTLola specification to the StreamIR follows the technique illustrated in Figure 4. Here, each stream in the specification is translated into a set of small StreamIR snippets. Input streams are translated to a statement shifting the stream and writing the new value to memory. Output streams are translated into four statements, one for each task – spawn, shift, eval, and close. The translated statements are then sorted according to their layers, where all statements in a layer are evaluated in parallel while the layers are concatenated sequentially. The translation also generates an initial memory consisting of the prefixes and the deadlines. The initial prefix of every input stream is assigned to an empty list whereas every output stream instance is marked as not-spawned. The initial deadline assigns every global frequency to its first deadline. The for-

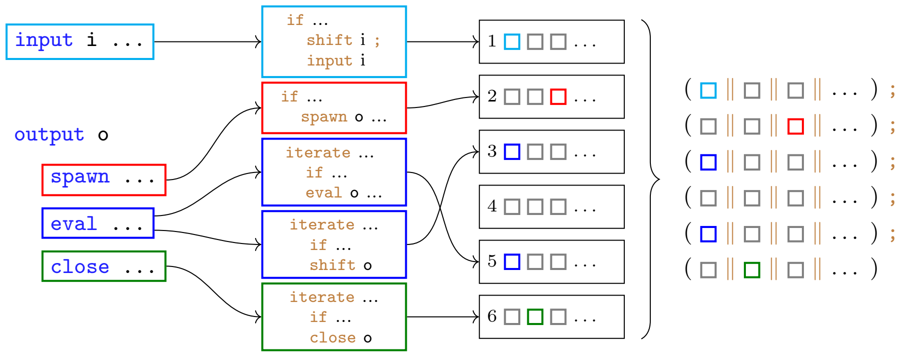


Fig. 4: Overview of the translation from RTLola specifications to StreamIR monitors.

mal definition of the translation and the generation of the initial memory is defined in Appendix C.

3.2 Optimizations

The translation of the monitors shown in Section 3.1 produces specific patterns, allowing for optimizations tailored to these patterns. The StreamIR representation of the monitor, properties of stream-based languages, and the analysis results enable the application of specialized rewriting rules. These rewrite rules follow standard compiler techniques, such as combining if statements or loops. However, since in a correctly constructed StreamIR program, every expression consistently evaluates to the same value regardless of its position, finding these patterns in the StreamIR monitor is simpler than in a general-purpose language. These compilers require complex program analysis which might miss the required assumptions to apply the optimizations.

Example 2. Consider the following snippets. The code on the left is produced by the translation from RTLola and the code on the right is an optimized version:

```
iterate p
    if schedule global 0.5
      ∧ dynamic self = syn a then
        eval p …
iterate q
    if schedule global 0.5
      ∧ dynamic self = syn a then
        eval q …
```

```
if schedule global 0.5 then
    assign (syn a)
        eval p … ;
        eval q …
```

First, we assume that the output streams p and q share the same spawn and close clauses, i.e., each clause has the same pacing, `when`-condition, and `with`-expression. This property ensures that the iteration blocks iterate over the same parameters, so the first rewrite rule combines these iterations. A second rule moves the first part of the guards outside the iteration since it is independent of the specific instance. This optimization is possible because the global deadline

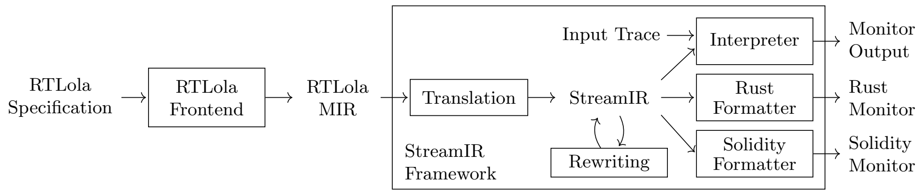


Fig. 5: Overview of the StreamIR framework

is parameter-independent, whereas a local deadline would not permit such optimization. Last, the remaining guard uniquely determines one instance, so the next rule replaces the iteration and the if statement with an assign statement directly calculating the parameter. For the list of all rewriting rules, consider Appendix D.

Memory. Stream-based specification languages operate over an infinite sequence of values. However, an analysis can often determine static memory bounds per stream instance. In RTLola, all temporal accesses are bounded, allowing the dependency analysis to reveal a finite buffer length for all instances. Based on a stream's structure, we can optimize the memory representation using a set of memory rewriting rules analogous to the rewriting rules from the previous paragraph. They include optimizations for non-parameterized streams, instances with a buffer length of 1, and streams that are only accessed synchronously.

Symbolic Execution. The StreamIR enables a partial evaluation of guard conditions. Code blocks surrounded by guards that are never satisfied can be removed, while guard checks can be eliminated if their conditions are never violated. For instance, in RTLola, the evaluation cycle is either completely event- or time-based. By partially evaluating the StreamIR, it is possible to create two versions: one containing event-based streams and one containing time-driven streams.

4 Implementation & Evaluation

Implementation. We extended the RTLola framework [5,15] with a StreamIR-based interpreter and a compiler to Rust and Solidity. We illustrate our setup in Figure 5. Our approach uses the existing RTLola frontend for parsing and analyzing specifications. The RTLolaMIR includes all inferred information, such as pacing types or memory bounds. We then apply the translation steps outlined in Section 3.1 as well as the rewriting rules introduced in Section 3.2. The library outputs the final StreamIR, which is then either interpreted or compiled.

The new *interpretation* uses just-in-time (JIT) compilation for the complete StreamIR while the existing RTLola interpreter [5] only uses JIT compilation for stream expressions. This approach leverages the discussed optimizations, i.e., partial evaluation of the StreamIR to decompose computation in time- and event-driven sections and rewrite rules to optimize control flow and memory.

For the *compiler*, the StreamIR library provides a framework to translate the monitor into different target languages. For each language, a formatter defines the translation of statements and expressions into the corresponding constructs of the target language. Currently, we support the compilation to two programming languages used in different domains. Our Rust formatter generates highly efficient Rust code that can run on microcontrollers. Our second formatter compiles specifications to Solidity, a programming language for smart contracts.

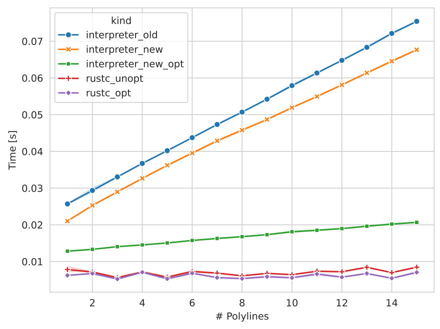
Our artifact is available at Zenodo[1] and our implementation is open-source and available on crates.io[2,3,4,5] and GitHub[6].

Evaluation. The evaluation was performed on a system with a 13th Gen Intel Core i7-1355U processor. Rust code is compiled using rustc version 1.84.0 with the highest optimization level 3, and Solidity using solc version 0.8.24 optimized for 1000 runs. We evaluate our implementation on three different benchmark sets and exclude the setup phases, such as specification analysis, for the runtime measurements. For the optimizations, we apply the same set of rewriting rules for all specifiations.

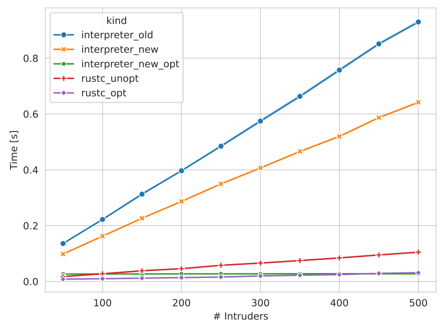
Unmanned Aircraft. First, we evaluate our approach using two specifications from the field of monitoring unmanned aircraft. The first specification is a geofencing application [4], where a drone's position is continuously monitored to ensure it remains within a designated geofence. This geofence is defined by a set of polygon lines in which the number of lines is proportional to the number of output streams. The second specification is based on the RTLola tutorial [5], observing the surrounding airspace of a drone to detect potential interference from other drones – called intruders. Unlike the first specification, which we evaluate across varying numbers of polygon lines, this benchmark varies the number of intruders while keeping the specification unchanged. Here, the number of intruders corresponds to the number of stream instances. For both specifications, we measure the execution time over 20 runs, each processing 10,000 input events. We report the runtime of the old interpreter [15], the runtime of the new StreamIR interpreter with and without the optimizations, and the runtime of the optimized and unoptimized monitors compiled using the Rust formatter.

As seen in Figure 6, the runtime of all monitors increases linearly because the number of stream evaluations increases either with the number of output streams or stream instances. In general, a compilation is much faster than an interpretation. When comparing the interpreter implementation, our new implementation outperforms the old one even without the optimizations by using JIT compilation for the complete StreamIR. After applying the StreamIR rewrite

[1] `https://doi.org/10.5281/zenodo.15222546`
[2] `https://crates.io/crates/rtlola-streamir`
[3] `https://crates.io/crates/rtlola-streamir-interpreter`
[4] `https://crates.io/crates/rtlola2rust`
[5] `https://crates.io/crates/rtlola2solidity`
[6] `https://github.com/reactive-systems/rtlola-streamir`



(a) Geofence specification with increasing number of polygon lines.



(b) Intruder specification with increasing number of intruders.

Fig. 6: Runtime comparison between interpreters and compiled Rust monitors.

rules, the runtime is significantly reduced. In contrast, the rewrite rules mostly do not affect the runtime for the geofence compilation to Rust, since the Rust compiler is capable of applying most optimizations on its own. For the intruder specification however, StreamIR optimizations are able to significantly optimize the runtime of the compiled monitor. The reason for this are optimizations of parameterized streams, which are easy in the StreamIR representation but hard for a Rust program in general. The individual runtimes are presented in Appendix F for the interested reader.

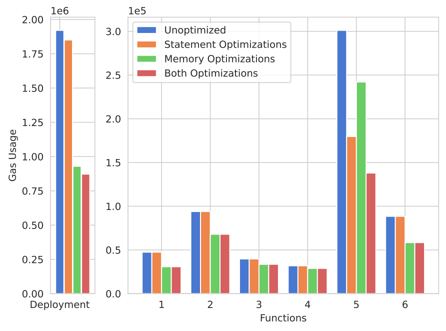
	Unopt	Opt
Existing Interpreter	17.258	-
StreamIR Interpreter	12.889	7.943
Compilation	3.157	1.443

Table 1: Runtime comparison of the fairness specification.

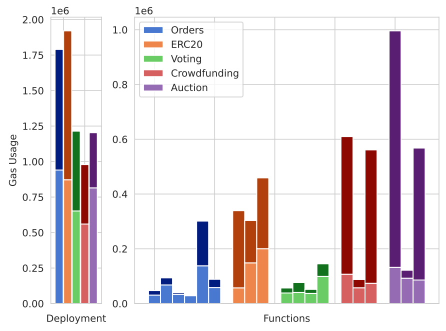
Algorithmic Fairness. The second benchmark utilizes the equalized-odds specification [7] for monitoring algorithmic fairness of the COMPAS tool. COMPAS was developed by Northpointe [26] and predicts the recidivism risk of defendants. A retrospective analysis [2] showed the bias of the tool and our recent paper [7] demonstrates the use of stream-based monitoring to detect such bias. Table 1 depicts the runtime on the COMPAS dataset, using the same evaluation setup described in the previous paragraph. The table reports similar results to the previous benchmark. Optimizations on the StreamIR reduce the runtime of the interpreter and compiler.

Smart Contracts. Efforts have explored monitoring smart contracts [3,1] or expressing them in formal specification languages [16,25,35]. We extend this effort with the compilation to Solidity, so the monitor or smart contract itself can be expressed in RTLola. Since the interpretations can not be executed on the blockchain, our evaluation focuses on the performance benefits of the optimizations when compiling to Solidity.

We extended our framework with a function interface to compile functions describing the contract. The functions set specific input stream values, call the



(a) Order contract



(b) All contracts

Fig. 7: Gas usage comparison of the Solidity monitors.

monitor and return new values of output streams. As not all input streams are set by a function, the compiler adapts the StreamIR using partial evaluation. The evaluation uses RTLola specifications that have previously been employed to monitor smart contracts [3,31,34] as well as existing Solidity contracts that we expressed in RTLola [32,33,30]. A description of the contracts and their functions can be found in Appendix E. For the comparison, we assess the required gas, the transaction fee for calling functions on the blockchain, for contract deployment and the average gas consumption per function call over 1000 calls.

Figure 7a displays the average gas consumption per function call when monitoring an ordering system, along with the gas costs for the contracts deployment. The functions allow the interaction with the contract, for example to place a new order, mark an order as delivered or request termination of the contract. The cost of the six functions and the deployment are each represented with four bars, where each bar represents the contract 1. without any optimizations 2. with statement optimizations 3. with memory optimizations, and 4. with both optimizations. The results demonstrate that the deployment costs mostly are reduced by memory optimizations minimizing the use of global variables. Furthermore, the gas reduction of some function calls relies on memory optimizations. By reducing reliance on global storage, these optimizations yield substantial savings in deployment and runtime. Meanwhile, StreamIR optimizations mainly improve function execution costs, especially for functions that update parameterized streams, such as the fifth function for placing a new order. These functions benefit significantly from the iterate-assign optimization, which eliminates unnecessary iterations and removes the need to store activate parameters.

Figure 7b gives an overview of the gas usage across a number of different contracts. In this figure, each bar corresponds to a function in a contract. The top half of the bar illustrates the gas costs for the StreamIR-unoptimized contract, while the lower half shows the reduction obtained by enabling all StreamIR optimizations. For example, the dark blue bars in Figure 7b correspond with the blue bars in Figure 7a and the light blue corresponds with the red bars. Figure 7b demonstrate that our optimizations reduce gas usage across all specifications.

Beyond gas usage reduction, the StreamIR optimizations address a fundamental issue of contracts with parameterized output streams. Iteration introduces unbounded gas usage – a highly undesirable property for smart contracts. By applying the optimization that replaces iterations with assignments – feasible for all evaluated specifications – gas usage becomes bounded again.

5 Conclusion

We have presented a new framework for the stream-based specification language RTLola, with an interpretation and a compilation to Rust and Solidity. Our framework uses a new intermediate representation designed for stream-based specifications. Further, we introduced rewrite rules for this representation to optimize the resulting monitors. Even without optimizations, our experiments show that the new interpreter and compiler outperform the existing RTLola interpreter. Additionally, we show the impact of the rewriting rules optimizing the runtime of the monitor. Especially in specifications with parameterized streams, our optimizations speed up the compiled monitor significantly. For the interpretation, our optimizations also have a huge impact for specifications without parameterized streams.

Acknowledgments. This work was partially supported by the Aviation Research Program LuFo of the German Federal Ministry for Economic Affairs and Energy as part of "Volocopter Sicherheitstechnologie zur robusten eVTOL Flugzustandsabsicherung durch formales Monitoring" (No. 20Q1963C), by the German Research Foundation (DFG) as part of TRR 248 (No. 389792660), and by the European Research Council (ERC) Grant HYPER (No. 101055412).

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

References

1. Abraham, M., Jevitha, K.: Runtime verification and vulnerability testing of smart contracts. In: Advances in Computing and Data Sciences: Third International Conference, ICACDS 2019, Ghaziabad, India, April 12–13, 2019, Revised Selected Papers, Part II 3. pp. 333–342. Springer (2019)
2. Angwin, J., Larson, J., Mattu, S., Kirchner, L.: Machine bias. there's software used across the country to predict future criminals. and it's biased against blacks. ProPublica (2016), `https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing`
3. Azzopardi, S., Ellul, J., Pace, G.J.: Monitoring smart contracts: Contractlarva and open challenges beyond. In: Runtime Verification: 18th International Conference, RV 2018, Limassol, Cyprus, November 10–13, 2018, Proceedings 18. pp. 113–137. Springer (2018)

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. Lecture Notes in Computer Science, vol. 14682, pp. 207–218. Springer (2024). 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: International Symposium on Formal Methods. pp. 624–648. Springer (2024)
6. Baumeister, J., Finkbeiner, B., Kruse, M., Schwenger, M.: Automatic optimizations for stream-based monitoring languages. In: International Conference on Runtime Verification. pp. 451–461. Springer (2020)
7. 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. pp. 60–81. Springer Nature Switzerland, Cham (2025)
8. 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. Lecture Notes in Computer Science, vol. 12225, pp. 28–39. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_3
9. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: Fpga stream-monitoring of real-time properties. ACM Transactions on Embedded Computing Systems (TECS) **18**(5s), 1–24 (2019)
10. Biewer, S., Finkbeiner, B., Hermanns, H., Köhl, M.A., Schnitzer, Y., Schwenger, M.: Rtlola on board: Testing real driving emissions on your phone. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 365–372. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_20
11. Boussinot, F., De Simone, R.: The esterel language. Proceedings of the IEEE **79**(9), 1293–1304 (1991)
12. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: Tessla: temporal stream-based specification language. In: Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21. pp. 144–162. Springer (2018)
13. 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)
14. 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)
15. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: Streamlab: stream-based monitoring of cyber-physical systems. In: International Conference on Computer Aided Verification. pp. 421–431. Springer (2019)

16. Finkbeiner, B., Hofmann, J., Kohn, F., Passing, N.: Reactive synthesis of smart contract control flows. In: International Symposium on Automated Technology for Verification and Analysis. pp. 248–269. Springer (2023)
17. Finkbeiner, B., Oswald, S., Passing, N., Schwenger, M.: Verified rust monitors for lola specifications. In: International Conference on Runtime Verification. pp. 431–450. Springer (2020)
18. Gautier, T., Le Guernic, P., Besnard, L.: Signal: A declarative language for synchronous programming of real-time systems. Springer (1987)
19. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: International Conference on Runtime Verification. pp. 282–298. Springer (2018)
20. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proceedings of the IEEE **79**(9), 1305–1320 (1991)
21. Henzinger, T.A., Karimi, M., Kueffner, K., Mallik, K.: Monitoring algorithmic fairness. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. pp. 358–382. Springer Nature Switzerland, Cham (2023)
22. Junges, S., Torfah, H., Seshia, S.A.: Runtime monitors for markov decision processes. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 553–576. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_26
23. Kallwies, H., Leucker, M., Prilop, M., Schmitz, M.: Optimizing trans-compilers in runtime verification makes sense–sometimes. In: International Symposium on Theoretical Aspects of Software Engineering. pp. 197–204. Springer (2022)
24. 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)
25. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: A finite state machine based approach. In: Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22. pp. 523–540. Springer (2018)
26. Northpoint Inc. d/b/a equivant: Practitioner's guide to compas core. https://archive.epic.org/algorithmic-transparency/crim-justice/EPIC-16-06-23-WI-FOIA-201600805-COMPASPractionerGuide.pdf (2015), accessed: 2025-01-30
27. Perez, I., Dedden, F., Goodloe, A.: Copilot 3. Tech. rep. (2020), https://ntrs.nasa.gov/citations/20200003164
28. Perez, I., Goodloe, A.E., Dedden, F.: Runtime verification in real-time with the copilot language: A tutorial. 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. Lecture Notes in Computer Science, vol. 14934, pp. 469–491. Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_27
29. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A hard real-time runtime monitor. In: International Conference on Runtime Verification. pp. 345–359. Springer (2010)
30. Program the Blockchain: Coin funding reference contract. https://programtheblockchain.com/posts/2018/01/19/writing-a-crowdfunding-contract-a-la-kickstarter/ (2022), accessed: 2025-01-30

31. Scheerer, F.: Monitoring smart contracts with rtlola. Bachelor's Thesis, Saarland University (2021)
32. Sudi, Andre: Voting reference contract. `https://github.com/andresudi/Voting-Smart-Contract/blob/master/Voting.sol` (2021), accessed: 2025-01-30
33. The Solidity Authors: Simple auction reference contract. `https://docs.soliditylang.org/en/latest/solidity-by-example.html#simple-open-auction`, accessed: 2025-01-30
34. Vogelsteller, Fabian and Buterin, Vitalik: Erc-20: Token standard. `https://eips.ethereum.org/EIPS/eip-20` (2015), accessed: 2025-01-30
35. Zupan, N., Kasinathan, P., Cuellar, J., Sauer, M.: Secure smart contract generation based on petri nets. In: Blockchain Technology for Industry 4.0: Secure, Decentralized, Distributed and Trusted Industry Environment, pp. 73–98. Springer (2020)

A Syntax

Statements	`stmt`	::=	`skip` | `shift sr` | `input sr`
		|	`spawn sr expr` | `eval sr expr` | `close sr`
		|	`stmt ; stmt` | `stmt || stmt`
		|	`if guard then stmt else stmt`
		|	`iterate sr stmt` | `assign expr stmt`
Conditions	`guard`	::=	`input? sr` | `schedule freq` | `dynamic expr`
		|	`guard` ($\vee$ | $\wedge$) `guard`
Expressions	`expr`	::=	`const c` | `get sr expr off expr` | `syn sr expr`
		|	`expr ( + | - | * | ...) expr` | `self`
		|	`window sr expr dur f` | `...`
Frequencies	`freq`	::=	`global dur` | `local dur sr`
Durations	`dur`	::=	$\mathbb{R}^+$
Aggregations	`f`	::=	$\mathbb{V}^* \rightarrow \mathbb{V}$
Constants	`c`	::=	$\mathbb{V}$

Fig. 8: Syntax of StreamIR

Definition 1 (StreamIR Monitors). *A monitor $\phi =$ `loop`$\{stmt\}$ is an infinite loop over the statement stmt.*

Remark 1. For readability, we use `if` c `then` *stmt* if the alternative is followed by the `skip` statement and removed the $\top$ placeholder for non-parameterized stream accesses in the paper.

B Semantics

Definition 2 (Memory). *A memory M is defined as a pair of a PrefixMap and a Deadline.*

$$\begin{aligned}
\mathit{Prefix} &:= (\mathbb{T}, \mathbb{V})^* \\
\mathit{PrefixMap} &:= SR_{in} \uplus SR_{out} \rightarrow \mathbb{V} \rightarrow (\mathit{Prefix} \cup \bot) \\
\mathit{GlobalDeadline} &:= \mathbb{F} \rightarrow \mathbb{R}_{\bot} \\
\mathit{LocalDeadline} &:= (\mathbb{F} \times SR_{in} \uplus SR_{out} \times \mathbb{V}) \rightarrow \mathbb{R}_{\bot} \\
\mathit{Deadline} &:= \mathit{GlobalDeadline} \cup \mathit{LocalDeadline} \\
\mathit{Memory} &:= \mathit{PrefixMap} \times \mathit{Deadline}
\end{aligned}$$

For readability, $M[sr][inst]$ defines the lookup to the prefix of a stream instance in the prefixmap M. The list notation $pre = pre' \cdot v$ splits the non-empty prefix pre into the last element v and the rest of the prefix pre', $|pre|$ returns the length of the prefix, and $pre[n]$ returns the n-th element in the prefix. For the aggregation functions, we use *slice* to get the slice of a prefix:

Definition 3 (Slice). *Given a prefix P and a timestamt t, the* slice *of P returns all values computed later than t:*

$$\text{slice} : \mathit{Prefix} \cup \bot \times \mathbb{R} \to \mathbb{V}^*$$
$$\text{slice}(v, t) = v \qquad \text{if } v \in \{\bot, \epsilon\}$$
$$\text{slice}(xs \cdot (v, t'), t) = \begin{cases} \text{slice}(xs, t) \cdot v & \text{if } t \leq t' \\ \epsilon & \text{otherwise} \end{cases}$$

$$\textsc{eval-self}\quad \frac{\mathit{inst} \neq \bot}{(M, \texttt{self}) \Downarrow_t^{\mathit{inst}} \mathit{inst}} \qquad \textsc{eval-syn}\quad \frac{(M, \mathit{inst}_{exp}) \Downarrow_t^{\mathit{inst}} \mathit{inst}' \qquad M[sr][\mathit{inst}'] = \mathit{pre} \cdot (t, \mathit{val})}{(M, \texttt{syn}\ sr\ \mathit{inst}_{exp}) \Downarrow_t^{\mathit{inst}} \mathit{val}}$$

$$\textsc{eval-get}\quad \frac{\begin{array}{c}(M, \mathit{inst}_{exp}) \Downarrow_t^{\mathit{inst}} \mathit{inst}' \qquad M[sr][\mathit{inst}'] = \mathit{pre} \\ |\mathit{pre}| > \mathit{off} \qquad \mathit{pre}[|\mathit{pre}| - \mathit{off}] = (t', \mathit{val})\end{array}}{(M, \texttt{get}\ sr\ \mathit{inst}_{exp}\ \mathit{off}\ \mathit{dft}) \Downarrow_t^{\mathit{inst}} \mathit{val}} \qquad \textsc{eval-get-dft}\quad \frac{\begin{array}{c}(M, \mathit{inst}_{exp}) \Downarrow_t^{\mathit{inst}} \mathit{inst}' \qquad M[sr][\mathit{inst}'] = \mathit{pre} \\ |\mathit{pre}| \leq \mathit{off} \qquad (M, \mathit{dft}) \Downarrow_t^{\mathit{inst}} \mathit{val}\end{array}}{(M, \texttt{get}\ sr\ \mathit{inst}_{exp}\ \mathit{off}\ \mathit{dft}) \Downarrow_t^{\mathit{inst}} \mathit{val}}$$

$$\textsc{eval-const}\quad \frac{c \in \mathbb{V}}{(M, \texttt{const}\ c) \Downarrow_t^{\mathit{inst}} c} \qquad \textsc{eval-bin-op}\quad \frac{(M, \mathit{exp}_1) \Downarrow_t^{\mathit{inst}} \mathit{val}_1 \qquad (M, \mathit{exp}_2) \Downarrow_t^{\mathit{inst}} \mathit{val}_2 \qquad \circ \in \{+, -, *, \dots\}}{(M, \mathit{exp}_1 \circ \mathit{exp}_2) \Downarrow_t^{\mathit{inst}} \mathit{val}_1 \circ \mathit{val}_2}$$

$$\textsc{eval-window}\quad \frac{(M, \mathit{inst}_{exp}) \Downarrow_t^{\mathit{inst}} \mathit{inst}' \qquad M[sr][\mathit{inst}'] = \mathit{pre} \qquad f(\text{slice}(\mathit{pre}, t - \mathit{dur})) = \mathit{val}}{(M, \texttt{window}\ sr\ \mathit{inst}_{exp}\ \mathit{dur}\ f) \Downarrow_t^{\mathit{inst}} \mathit{val}}$$

Fig. 9: Operational semantics of stream expressions

Definition 4 (Evaluation of expressions). *Given a prefixmap M, a stream expression $expr$, a current instance $inst$ and a timepoint t, the expression evaluation $(M, expr) \Downarrow_t^{inst} val$ is described by the inference rules in Figure 9.*

$$\textsc{eval-input-guard}\quad \frac{I(sr) = v}{(M, \texttt{input?}\ sr) \Downarrow_{I,D,t}^{\mathit{inst}} v \neq \bot} \qquad \textsc{eval-bin-guard}\quad \frac{(M, \mathit{exp}_1) \Downarrow_{I,D,t}^{\mathit{inst}} \mathit{val}_1 \qquad (M, \mathit{exp}_2) \Downarrow_{I,D,t}^{\mathit{inst}} \mathit{val}_2 \qquad \circ \in \{\wedge, \vee\}}{(M, \mathit{exp}_1 \circ \mathit{exp}_2) \Downarrow_{I,D,t}^{\mathit{inst}} \mathit{val}_1 \circ \mathit{val}_2}$$

$$\textsc{eval-schedule-global}\quad \frac{D(\mathit{freq}) = t'}{(M, \texttt{schedule global}\ \mathit{freq}) \Downarrow_{I,D,t}^{\mathit{inst}} t' = t} \qquad \textsc{eval-schedule-local}\quad \frac{D(\mathit{freq}, sr, \mathit{inst}) = t'}{(M, \texttt{schedule local}\ \mathit{freq}\ sr) \Downarrow_{I,D,t}^{\mathit{inst}} t' = t}$$

$$\textsc{eval-dynamic}\quad \frac{(M, \mathit{exp}) \Downarrow_t^{\mathit{inst}} \mathit{val} \qquad \mathit{val} \in \mathbb{B}}{(M, \texttt{dynamic}\ \mathit{exp}) \Downarrow_{I,D,t}^{\mathit{inst}} \mathit{val}}$$

Fig. 10: Operational semantics of guards

Definition 5 (Evaluation of guards). *Given a prefixmap M, a guard g, a current instance $inst$, an input I, a deadline D and a timepoint t, the evaluation of a guard $(M, g) \Downarrow_{I,D,t}^{inst} val$ is described by the inference rules in Figure 10.*

Definition 6 (Update). *Given a prefixmap M, a stream reference sr, and an instance $inst$, an* update *$M[sr][inst \leftarrow v]$ is defined as*

$$\cdot\,[\cdot][\cdot \leftarrow \cdot] : (\mathit{PrefixMap} \times \mathit{SR}_{\mathit{in}} \uplus \mathit{SR}_{\mathit{out}} \times \mathbb{V} \times \mathbb{V}) \to \mathit{PrefixMap}$$

$$M[sr][\mathit{inst} \leftarrow v] = \lambda sr', \mathit{inst}' \begin{cases} v & \text{if } sr = sr' \wedge \mathit{inst} = \mathit{inst}' \\ M[sr'][\mathit{inst}'] & \text{otherwise} \end{cases}$$

EXEC-SKIP
$$\frac{}{(M, \texttt{skip}) \Downarrow^{inst}_{I,D,t} M}$$

EXEC-SHIFT
$$\frac{}{(M, \texttt{shift}\ sr) \Downarrow^{inst}_{I,D,t} M[sr][inst \leftarrow M[sr][inst] \cdot \bot]}$$

EXEC-SEQ
$$\frac{(M_1, p_1) \Downarrow^{inst}_{I,D,t} M_2 \qquad (M_2, p_2) \Downarrow^{inst}_{I,D,t} M_3}{(M_1, p_1\ \texttt{;}\ p_2) \Downarrow^{inst}_{I,D,t} M_3}$$

EXEC-PARALLEL
$$\frac{(M, p_1\ \texttt{;}\ p2) \Downarrow^{inst}_{I,D,t} M' \qquad (M, p_2\ \texttt{;}\ p_1) \Downarrow^{inst}_{I,D,t} M'}{(M, p_1 \parallel p_2) \Downarrow^{inst}_{I,D,t} M'}$$

EXEC-INPUT
$$\frac{I[sr] = val \qquad val \neq \bot \qquad M[sr][\top] = str \cdot \bot}{(M, \texttt{input}\ sr) \Downarrow^{inst}_{I,D,t} M[sr][\top \leftarrow str \cdot (t, val)]}$$

EXEC-EVAL
$$\frac{(M, exp) \Downarrow^{inst}_{t} val \qquad M[sr][inst] = str \cdot \bot}{(M, \texttt{eval}\ sr\ exp) \Downarrow^{inst}_{I,D,t} M[sr][inst \leftarrow str \cdot (t, val)]}$$

EXEC-CLOSE
$$\frac{}{(M, \texttt{close}\ sr) \Downarrow^{inst}_{I,D,t} M[sr][inst \leftarrow \bot]}$$

EXEC-SPAWN-NEW
$$\frac{(M, inst_{exp}) \Downarrow^{inst}_{t} inst' \qquad M[sr][inst'] = \bot}{(M, \texttt{spawn}\ sr\ inst_{exp}) \Downarrow^{inst}_{I,D,t} M[sr][inst' \leftarrow \epsilon]}$$

EXEC-SPAWN-EXISTS
$$\frac{(M, inst_{exp}) \Downarrow^{inst}_{t} inst' \qquad M[sr][inst'] \neq \bot}{(M, \texttt{spawn}\ sr\ inst_{exp}) \Downarrow^{inst}_{I,D,t} M}$$

EXEC-IF-TRUE
$$\frac{(M, stmt) \Downarrow^{inst}_{I,D,t} M' \qquad (M, g) \Downarrow^{inst}_{t} true}{(M, \texttt{if}\ g\ \texttt{then}\ stmt\ \texttt{else}\ stmt') \Downarrow^{inst}_{I,D,t} M'}$$

EXEC-IF-FALSE
$$\frac{(M, stmt') \Downarrow^{inst}_{I,D,t} M' \qquad (M, g) \Downarrow^{inst}_{t} false}{(M, \texttt{if}\ g\ \texttt{then}\ stmt\ \texttt{else}\ stmt') \Downarrow^{inst}_{I,D,t} M'}$$

EXEC-ITERATE
$$\frac{\begin{array}{c}\{p_1, \ldots, p_n\} = \{p \mid M_0[sr][p] \neq \bot\} \\ (M_0, stmt) \Downarrow^{p_1}_{I,D,t} M_1 \quad \ldots \quad (M_{n-1}, stmt) \Downarrow^{p_n}_{I,D,t} M_n\end{array}}{(M_0, \texttt{iterate}\ sr\ stmt) \Downarrow^{\top}_{I,D,t} M_n}$$

EXEC-ASSIGN
$$\frac{(M, inst_{exp}) \Downarrow^{\top}_{t} inst \qquad (M, stmt) \Downarrow^{inst}_{I,D,t} M'}{(M, \texttt{assign}\ inst_{exp}\ stmt) \Downarrow^{\top}_{I,D,t} M'}$$

Fig. 11: Operational semantics of statements

Definition 7 (Evaluation of statements). *Given a prefixmap M, a statement stmt, a current instance inst, an input I, a deadline D and a timepoint t, the evaluation of statement $(M, stmt) \Downarrow^{inst}_{I,D,t} M'$ returns a new prefixmap M' following the inference rules in Figure 11.*

Definition 8 (Next Deadline). *Given a deadline D, a timepoint t, and the prefixmaps M and M', the* deadline update *is defined as*

$$nextDl : Deadline \times \mathbb{R} \times PrefixMap \times PrefixMap \rightarrow Deadline$$
$$nextDl(D, t, M, M') = nextGlobalDl(D, t) \cup nextLocalDl(D, t, M, M')$$
$$nextGlobalDl(D, t) = \lambda f. \begin{cases} t + f & \text{if } D(f) = t \\ D(f) & \text{otherwise} \end{cases} \qquad nextLocalDl(D, t, M, M') =$$
$$\lambda f, sr, inst. \begin{cases} t + f & \text{if } D(f, sr, inst) = t \\ t + f & \text{if } M[sr][inst] = \bot \wedge M'[sr][inst] \neq \bot \\ \bot & \text{if } M[sr][inst] \neq \bot \wedge M'[sr][inst] = \bot \\ D(f, sr, inst) & \text{otherwise} \end{cases}$$

Definition 9 (Step semantics of statements). *Given a memory M, an input I, and a timestamp t, a step of a statement $(M, stmt) \Rightarrow_{I,t} M'$ returns a memory M' using the inference rule in Figure 12.*

$$\frac{\text{EXEC-INPUT} \\ min(D) > t \qquad (M, stmt) \Downarrow^{\top}_{I,D,t} M'}{((M,D), stmt) \Rightarrow_{I,t} (M', D)} \qquad \frac{\text{EXEC-DEADLINE} \\ min(D) \leq t \qquad (M, stmt) \Downarrow^{\top}_{\emptyset,D,min(D)} M' \\ D' = nextDl(D, min(D), M, M') \\ ((M', D'), stmt) \Rightarrow_{I,t} (M'', D'')}{((M,D), stmt) \Rightarrow_{I,t} (M'', D'')}$$

Fig. 12: Semantics of the evaluation of a step in the monitor

Definition 10 (Semantics). *Given an initial memory M_0 and an infinite input sequence $\mathcal{W}_{in}$, a monitor describes a sequence of* memories

$$[\![\texttt{loop}\ \{stmt\}]\!]_{\mathcal{W}_{in},M_0} := \{M_0, M_1, \dots \mid \forall n. M_n \Rightarrow_{\mathcal{W}_{in}[n]} M_{n+1}\}$$

Definition 11 (Well-definedness). *A program stmt with an initial memory M is well-defined if, for any infinite sequence of inputs $\mathcal{W}_{in}$ with monotone timestamps, the program has exactly one evaluation model:*

$$\forall \mathcal{W}_{in}.\forall n. Time(\mathcal{W}_{in}[n]) < Time(\mathcal{W}_{in}[n+1]) \to [\![\texttt{loop}\ \{stmt\}]\!]_{\mathcal{W}_{in},M} = \{\mathcal{W}_{out}\}$$

C Translation from RTLola to StreamIR

$$\begin{aligned}
\text{translate}_{\varphi}(\epsilon) &:= \texttt{skip} \\
\text{translate}_{\varphi}(layer \cdot xs) &:= \text{translate}_{\varphi}(layer)\ \texttt{;}\ \text{translate}_{\varphi}(xs) \\
\text{translate}_{\varphi}(\{task_0, ...task_n\}) &:= \text{translate}_{\varphi}(task_0) \parallel \dots \parallel \text{translate}_{\varphi}(task_n) \\
\text{translate}_{\varphi}(Input(i)) &:= \texttt{if inputs?}\ i\ \texttt{then shift}\ i\ \texttt{; input}\ i \\
\text{translate}_{\varphi}(Spawn(o)) &:= \texttt{if}\ p_s \land \texttt{dynamic}\ c_s\ \texttt{then spawn}\ o\ e_s \\
\text{translate}_{\varphi}(Shift(o)) &:= \begin{array}{l}\texttt{iterate}\ o \\ \quad \texttt{if}\ p_e \land \texttt{dynamic}\ c_e\ \texttt{then shift}\ o\end{array} \\
\text{translate}_{\varphi}(Eval(o)) &:= \begin{array}{l}\texttt{iterate}\ o \\ \quad \texttt{if}\ p_e \land \texttt{dynamic}\ c_e\ \texttt{then eval}\ o\ e\end{array} \\
\text{translate}_{\varphi}(Close(o)) &:= \begin{array}{l}\texttt{iterate}\ o \\ \quad \texttt{if}\ p_c \land \texttt{dynamic}\ c_c\ \texttt{then close}\ o\end{array}
\end{aligned}$$

Fig. 13: Translation of an RTLola specification to a monitor

Definition 12 (Translation). *Given an RTLola specification φ with fully annotated output streams*

```
output o(inst)
    spawn @ps when cs with es eval @pe when ce with ee close @pc
        when cc
```

and a partial order Layers generated from the Dependency Graph of φ, the translation translate$_{\varphi}$(Layers) defined in Figure 13 returns a statement stmt where `loop { stmt }` *is a monitor ψ for φ.*

Definition 13 (Initial Memory). *Given an RTLola specification, the initial memory is defined as the pair (M_0, D_0) with*

$$\begin{aligned} \forall sr \in SR_{in}. M_0[sr][\top] &= \epsilon \\ \forall sr \in SR_{out}. \forall inst \in \mathbb{V}. M_0[sr][inst] &= \bot \\ \forall Global(freq) \in \varphi. D_0(freq) &= freq \\ \forall Local(freq) \in \varphi. sr \in SR_{out}. inst \in \mathbb{V}. D_0(freq, sr, inst) &= \bot \end{aligned}$$

Theorem 1 (Well-Definedness). *For an RTLola specification φ and a partial order derived from the dependency graph of φ, the translation returns a well-defined monitor ψ with initial memory M_0.*

Proof. The proof iterates over all failable inference rules for statements, expressions, and guards, and contradicts these rules to the well-definedness or well-typedness of φ, the list of layers or the initial memory to show the existence of an output. To prove that this output is unique, the proof uses the well-definedness of φ and shows that the order of the instance iterations does not affect the computation, i.e., otherwise, φ would not be well-typed.

D Rewriting Rules

D.1 General

Common-If: $\circ \in \{ \texttt{;}\, , \parallel \}$	
$\texttt{if}\ c \wedge c_1\ \texttt{then}\ A \circ \texttt{if}\ c \wedge c_2\ \texttt{then}\ B$	$\texttt{if}\ c\ \texttt{then}$ $\quad\texttt{if}\ c_1\ \texttt{then}\ A \circ \texttt{if}\ c_2\ \texttt{then}\ B$
Nested-If: $\circ \in \{ \texttt{;}\, , \parallel \}$	
$(\texttt{if}\ c\ \texttt{then}\ A) \circ (\texttt{if}\ c\ \texttt{then}\ B)$	$\texttt{if}\ c\ \texttt{then}\ (A \circ B)$
Split-If	
$\texttt{if}\ c_1 \wedge c_2\ \texttt{then}\ A$	$\texttt{if}\ c_1\ \texttt{then}\ \texttt{if}\ c_2\ \texttt{then}\ A$
Combine-If	
$\texttt{if}\ c_1\ \texttt{then}\ \texttt{if}\ c_2\ \texttt{then}\ A$	$\texttt{if}\ c_1 \wedge c_2\ \texttt{then}\ A$
Implied-If: If $c_1 \Rightarrow c_2$	
$\texttt{if}\ c_1\ \texttt{then}\ \ldots \texttt{if}\ c_2\ \texttt{then}\ A\ \ldots$	$\texttt{if}\ c_1\ \texttt{then}\ \ldots A \ldots$
Associativity: $\circ \in \{ \texttt{;}\, , \parallel \}$	
$A \circ (B \circ C)$	$(A \circ B) \circ C$
Swap-Par	
$A \parallel B$	$B \parallel A$
Iterate-Inside: if c does not contain `self` and is not `dynamic`	
$\texttt{iterate}\ o\ (\ \texttt{if}\ c\ \texttt{then}\ A\ )$	$\texttt{if}\ c\ (\ \texttt{iterate}\ o\ A\ )$
Unique-Assign: if c uniquely determines a parameter p	
$\texttt{iterate}\ o\ (\ \texttt{if}\ c\ \texttt{then}\ A\ )$	$\texttt{assign}\ p\ (\ \texttt{if}\ c\ \texttt{then}\ A\ )$
Combine-Iterate-Seq: if $o1$ and $o2$ share spawn and close condition	
$(\ \texttt{iterate}\ o1\ A\ )\ \texttt{;}\ (\texttt{iterate}\ o2\ B)$	$\texttt{iterate}\ o1\ (A\ \texttt{;}\ B\ )$

COMBINE-ITERATE-PAR: if $o1$ and $o2$ share spawn and close condition	
(iterate $o1$ A) ‖ (iterate $o2$ B)	iterate $o1$ (A ‖ B)
SINGLE-INSTANCE: if o has only a single instance	
(iterate o A)	A
IF-TRUE	
if $true$ then A else B	A
IF-FALSE	
if $false$ then A else B	B
REMOVE-SKIP-SEQ: $\circ \in \{$; , ‖ $\}$	
A ∘ skip | skip ∘ A	A
REMOVE-SKIP-IF	
if c skip	skip
REMOVE-SKIP-ITERATE	
iterate o skip	skip
REMOVE-SKIP-ASSIGN	
assign p skip	skip

D.2 Implementation Specific

The following rewrite rules are based on assumptions about the implementation, such as the bounded memory being realized as a ring buffer:

UNECESSARY-SHIFT: if o has a memory bound ≤ 1	
shift o	skip
UNECESSARY-SPAWN: if o has a single instance without spawn condition	
spawn o	skip

D.3 Guards

AND-TRUE	
$c \wedge true$ | $true \wedge c$	c
AND-FALSE	
$c \wedge false$ | $false \wedge c$	$false$
OR-TRUE	
$c \vee true$ | $true \vee c$	$true$
OR-FALSE	
$c \vee false$ | $false \vee c$	c
DYNAMIC-OP: $\circ \in \{ \wedge , \vee \}$	
dynamic $(c_1 \circ c_2)$	dynamic c_1 $\circ$ dynamic c_2

E Contracts

For the evaluation, we only list functions updating the state of the contract. Getter functions are ommited.

Order Contract [3,31]

1. *acceptContract:* Both parties, the buyer and seller, need to accept the contract before the placement of the first order.
2. *deliveryMade:* Marks an order as delivered by the seller.
3. *escrow:* Before accepting the contract, the buyer has to pay an escrow of a minimum number of items, while the seller has to pay an performance guarantee.
4. *paymentReceived:* The seller marks the payment of an order as received.
5. *placeOrder:* The buyer places a new order.
6. *requestTermination:* Either party can request termination of the contract.

ERC20 Contract [34]

1. *approve:* Permit another account to transfer tokens from this account.
2. *transfer:* Transfer tokens from your account to someone elses.
3. *transferFrom:* Transfer tokens from another account, if it was approved.

Voting Contract [32]

1. *endVoting:* End the voting phase.
2. *giveRightToVote:* Assign someone the right to vote.
3. *startVoting:* Start the voting phase.
4. *voteFor:* Cast your vote.

Crowdfunding Contract [30]

1. *bid:* Contribute money to the fund.
2. *claim:* If goal was reached, claim the funded money.
3. *refund:* If goal was not reached, refund your contribution.

Auction Contract [33]

1. *bid:* Bit to the auction.
2. *end:* Mark the auction as ended.
3. *withdraw:* Withdraw money if previous bid was overbid.

F Evaluation Results

The following tables displays the mean runtime over 20 runs across the different parameters. They correspond to the runtimes depicted in figures Figure 6a and Figure 6b respectively.

F.1 Geofence Specification

parameter	Runtime [ms]				
	ip_old	ip_new	ip_new_opt	rustc	rustc_opt
1	25.7	21.0	12.8	7.8	6.2
2	29.4	25.3	13.3	7.2	6.7
3	33.1	29.0	14.1	5.6	5.2
4	36.7	32.6	14.5	7.1	7.1
5	40.2	36.2	15.1	5.8	5.3
6	43.7	39.5	15.7	7.3	6.8
7	47.3	42.9	16.3	6.8	5.6
8	50.7	45.8	16.8	6.1	5.3
9	54.2	48.7	17.3	6.8	5.9
10	57.9	51.9	18.1	6.4	5.6
11	61.3	54.9	18.5	7.4	6.6
12	64.8	58.1	19.0	7.2	5.7
13	68.3	61.4	19.6	8.4	6.8
14	72.1	64.5	20.2	7.0	5.4
15	75.4	67.7	20.7	8.5	7.1

F.2 Intruder Specification

parameter	Runtime [ms]				
	ip_old	ip_new	ip_new_opt	rustc	rustc_opt
50	135.9	98.1	26.6	17.5	8.2
100	222.2	162.4	26.9	27.5	9.9
150	312.8	226.4	27.0	38.3	11.7
200	396.7	286.7	27.2	45.8	13.9
250	484.6	349.5	27.2	58.0	15.8
300	574.5	406.5	27.3	65.9	19.9
350	663.2	465.8	27.7	74.9	22.3
400	757.1	519.6	27.8	84.3	24.5
450	851.2	587.0	27.4	95.0	29.0
500	929.6	641.6	27.4	104.8	31.3