Automatic Optimizations for Stream-based Monitoring Languages*

Jan Baumeister[1][0000−0002−8891−7483], Bernd Finkbeiner[1][0000−0002−4280−8441], Matthis Kruse[2][0000−0003−4062−9666], and Maximilian Schwenger[1][0000−0002−2091−7575]

[1] CISPA Helmholtz Center for Information Security, Saarland Informatics Campus
66123 Saarbrücken, Germany
{jan.baumeister, finkbeiner, maximilian.schwenger}@cispa.saarland
[2] Saarland University, Saarland Informatics Campus
66123 Saarbrücken, Germany
matthis.kruse@cs.uni-saarland.de

Abstract. Runtime monitors that are specified in a stream-based monitoring language tend to be easier to understand, maintain, and reuse than those written in a standard programming language. Because of their formal semantics, such specification languages are also a natural choice for safety-critical applications. Unlike for standard programming languages, there is, however, so far very little support for automatic code optimization. In this paper, we present the first collection of code transformations for the stream-based monitoring language RTLola. We show that classic compiler optimizations, such as Sparse Conditional Constant Propagation and Common Subexpression Elimination, can be adapted to monitoring specifications. We also develop new transformations — Pacing Type Refinement and Filter Refinement — which exploit the specific modular structure of RTLola as well as the implementation freedom afforded by a declarative specification language. We demonstrate the significant impact of the code transformations on benchmarks from the monitoring of unmanned aircraft systems (UAS).

Keywords: Runtime Verification · Stream Monitoring · Compiler Optimizations · Specification Languages

1 Introduction

The spectrum of languages for the develpment of monitors ranges from standard programming languages, like Java and C++, to formal logics like LTL and its many variations. The advantage of programming languages is the universal expressiveness and the availability of modern compiler technology; programming languages lack, however, the precise semantics and compile-time guarantees

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

needed for safety-critical applications. Formal logics, on the other hand, are sufficiently precise, but have limited expressiveness. A good trade-off between the two extremes is provided by stream-based monitoring languages like RTLola. Stream-based languages have the expressiveness of a programming language, and, at the same time, the formal semantics and compile-time guarantees of a formal specification language.

For standard programming languages, the development of effective code optimizations is one of the most fundamental research questions. By contrast, there is, so far, very little support for the automatic optimization of monitoring specifications. In this paper, we present the first collection of code transformations for the stream-based monitoring language RTLola [7].

Our starting point are compiler optimizations known from imperative programming languages like *Sparse Conditional Constant Propagation* and *Common Subexpression Elimination*. Adapted to stream-based specifications, such transformations allow the user to write code that is easy to read and maintain, without the performance penalty resulting, for example, from unnecessarily recomputing the value of subexpressions.

We also develop optimizations that are specific to stream-based monitoring. Stream-based languages have several features that make them a particularly promising target for code optimization. Stream-based languages are *declarative* in the sense that it is only the correct computation of the trigger conditions that matters for the soundness of the monitor, not the specific order in which intermediate data is produced. This leaves much more freedom for code transformation than in an imperative language. Another feature of stream-based languages that is beneficial for code transformation is that the write-access to memory is inherently *local*: the computation of a stream only writes once in its local memory while potentially reading multiple times from other streams[3]. This means that expressions used for the computation of one stream can be modified without affecting the other streams. Finally, our code optimization exploits the clear *dependency structure* of stream-based specifications, which allows us to efficiently propagate type changes made in one stream to all affected streams in the remainder of the specification. We present two transformations that specifically exploit these advantages. *Pacing Type Refinement* optimizes the points in time when a stream value is calculated, eliminating the computation of stream values that are never used. *Filter Refinement* avoids the unncecessary computation of expressions that appear in the scope of an *if* statement, ensuring that the expression is only evaluated if the condition is actually true.

RTLola specifications are used both in interpretation-based monitors [7] and as the source language for compilers, for example to VHDL [3]. Our code transformations are applicable in both approaches, because the transformations are applied already on the level of intermediate representations (AST, IR). In transpilation backends, the optimized code is compiled one more time, and thus

[3] This is related to the functional programming paradigm where function calls are *pure*, i.e., free of side effects.

additionally benefits from the standard compiler optimizations for the target platform.

A prime application area for our optimizations is the monitoring of unmanned aircraft systems (UAS) [2]. Monitoring aircraft involves complex computations, such as the crossvalidation of different sensor modules. The performance of the monitor implementation is critical, because the on-board monitor is executed on a platform with limited computing power. Our experience with the code transformations (for details see Section 5) is very encouraging.

1.1 Related Work

This paper presents the first collection of code transformations for the stream-based monitoring language RTLola. There is, of course, a vast literature on compiler optimization. For an introduction, we refer the reader to the standard textbooks on compiler design and implementation (cf. [1, 14, 17]). Kildall [13] gives a comprehensive overview on the classic code transformations. The foundation for the code transformations is provided by methods from program analysis such as abstract interpretation [6].

The programming paradigm that most closely resembles stream-based monitoring languages like RTLola is *synchronous programming*. Examples of synchronous programming languages are Lustre [12], Esterel [4], and Signal [9]. These languages are supported by optimization techniques like the annotation-based memory optimization of Lustre [10] and the low-level elimination of redundant gates and latches in Esterel [15]. There are, however, important differences to the transformations presented in this paper. Our transformations work on the level of intermediate representations, which makes them uniformly applicable to interpretation and compilation. The new *Pacing Type* and *Filter Refinements* furthermore exploit the specific modular structure of RTLola as well as the much greater implementation freedom afforded by a declarative specification language.

Our focus on RTLola is motivated by recent work on RTLola-based monitoring for UAS [2] and other cyber-physical systems [3, 7]. It should be possible, however, to develop similar optimizations for other stream-based monitoring languages like TeSSLa [5] and Striver [11].

2 RTLola

RTLola [7, 8] is a runtime monitoring framework. In its core, it takes a specification in the eponymous specification language and analyzes whether and when input data violates the specification. To this end, it interprets sequences of incoming data points as input streams. The RTLola stream engine then transforms these values according to stream expressions in the specification to obtain output streams. The specification also contains trigger conditions, i.e., boolean expressions indicating whether a certain property is violated or not. Stream expressions and trigger conditions depend either on input or output stream values.

Consider the following RTLola specification.

```
input gps: (Float64, Float64)
output gps_readings: Bool@1Hz := gps.aggregate(over:2s,using:count)
trigger gps_glitch < 10 "GPS sensor frequency < 5Hz"
```

The specification first declares an input stream with the name `gps`. The output stream `gps_readings` analyzes the input stream by counting how many readings the monitor received within the last 2s. This computation is a sliding window, so when the `gps_readings` stream computes a new value at point in time t, RTLola takes all data points of the `gps` stream into account, which were received in the interval $[t - 2\text{s}, t]$. The trigger then checks whether the number of GPS readings in such a 2s interval falls below 10. If so, it raises an alarm such that the observed system can react accordingly e.g. by initiating mitigation procedures.

2.1 Type System

Types in RTLola are two-dimensional consisting of the value type and the pacing type. The former is drawn from a set of types representable with a static amount of bits. The pacing type consists of two components: an evaluation trigger and a filter condition. The monitor will compute a new value for a stream as soon as the evaluation trigger occurred *unless* the filter condition is false. Let us ignore filter conditions for now. The evaluation trigger can be a real-time frequency as was the case for `gps_readings`. In this case, the stream is a *periodic* stream. Otherwise, the evaluation trigger is a positive boolean formula φ over the set of input streams, in which case the stream is *event-based*. The reason behind this lies within the input model of RTLola. RTLola assumes input values to arrive asynchronously, i.e., if a specification declares several input streams $\mathcal{I}$, an incoming data point $\mathcal{I}'$ can cover an arbitrary non-empty subset $\emptyset \neq \mathcal{I}' \subseteq \mathcal{I}$. Only streams in $\mathcal{I}'$ receive a new value. Thus, the monitor evaluates event-based streams with evaluation trigger ι iff $\mathcal{I}' \implies \iota$. I.e., it replaces all occurrence of the input stream name i in ι by true if $\iota \in \mathcal{I}'$ and false otherwise. Consequently, any input stream i has evaluation trigger $\{i\}$ intuitively meaning "i will be extended when the system provides a new value for it." For event-based streams, the evaluation trigger is called the *activation condition*.

Note that the type annotation of `gps` in the previous example does not contain information about the pacing type at all. In many cases, RTLola infers the types of streams automatically based on the stream expression rendering type annotations largely optional. While the type inference for value types is straight-forward because RTLola requires input streams to have type annotations, the inference for pacing types is mainly based on stream accesses. There are three kinds of stream accesses: synchronous, asynchronous, and aggregations. If a stream x accesses a stream y synchronously, then the evaluation of x demands the nth-to-latest value of y where n is the *offset* of the access. This ties the evaluation of both streams together, so if y has an evaluation frequency of 5Hz, x cannot be evaluated more frequently, nor can x be event-based. Asynchronous accesses refer to the last value of a stream, no matter how old it may be. Here, the pacing of x and y remain decoupled. Aggregating accesses — such as the one in `gps_readings` — decouple the pacing as well.

Lastly, filter conditions are regular RTLola expressions. Assume stream x has the evaluation condition π with filter ϕ. Whenever π is true, the monitor evaluates the filter ϕ. Only if the filter is true as well, the monitor evaluates the stream expression and extends x.

2.2 Evaluation

An RTLola specification consists of input streams, output streams, and triggers. The monitor for a specification computes a static schedule containing information on which a periodic stream needs to be computed at which point in time. When such a point in time is reached or the monitor receives new input values, it starts an evaluation cycle. Here, the monitor first determines which streams could be affected by checking their frequencies or activation conditions. It then orders them according to an *evaluation order* $\prec$. Following this order, the monitor checks the filter condition of each stream. If it evaluates to true, the monitor extends the stream by evaluating the stream expression to obtain a new value.

This process only works correctly if the evaluation order complies with the *dependency graph* of the specification. The annotated dependency graph is a directed multigraph consisting of one node for each trigger, stream, and filter condition. Each edge in the graph represents a stream access in the specification. For the evaluation order, only synchronous lookups matter: if node s access node s' synchronously, s' needs to be evaluated before s.

After the evaluation, the monitor checks whether a trigger conditions was true. If so, passes the information on to the system under scrutiny. This constitutes the *observable behavior* of the monitor, any other computation is considered internal behavior. Consequently, any computation that does not impact a trigger condition is completely irrelevant.

This is just a rough outline of RTLola. For more information refer to [16].

Remark 1 (Transformations Preserve Observable Behavior). The point behind the compiler transformations presented in this paper is to improve the running time and thus decrease the latency between the occurrence and report of a violation. Yet, the correctness, i.e., the observable behavior of the monitor needs to remain unchanged. Thus, the transformations may alter the behavior of the monitor arbitrarily granted the observable behavior remains the same.

3 Classical Compiler Optimizations

In this section, we explain the adaption of classical compiler optimization techniques to the specification language RTLola. These techniques focuses on the expression of a stream under consideration of the pacing type. We exemplarily introduce transformations for the *Sparse Conditional Constant Propagation* and the *Common Expression Elimination*.

3.1 Sparse Conditional Constant Propagation

Sparse Conditional Constant Propagation (SCCP) allows the programmer to write maintainable specifications without a performance penalty of constant streams. It inlines them, pre-evaluates constant expressions, and deletes never accessed streams that includes a simple dead-code elimination. This procedure works transitively, i.e., a stream that turns constant due to the inlining will again be subject to the same transformation. Note that evaluating a constant expression might change the activation condition of a stream. Thus, the transformation annotates types explicitly before changing expressions.

3.2 Common Subexpression Elimination

The Common Subexpression Elimination (CSE) identifies subexpressions that appear multiple times and assigns the subexpressions to new streams. These new streams might increase the required memory but save computation time by eliminating repeated computations.

In RTLola, finding common subexpressions is simple compared to imperative programing languages for several reasons. First, RTLola as a declarative language is agnostic to the syntactic order in which streams are declared; the evaluation order only depends on the dependency graph. Secondly, expression evaluations are *pure*, i.e., free of side effects. As a result, the common subexpression elimination becomes a syntactic task except that it requires access to the inferred types. Here, two subexpressions are only considered common, if their pacing is of the same kind: periodic or event-based. This is necessary because RTLola strictly separates the evaluation of expressions with different pacing type kinds.

After identifying a common subexpression, the transformation creates a new stream and replaces occurrences of the expression by stream accesses. The pacing type of the newly created stream is either the disjunction of the activation conditions of accessing streams, or the least common multiple of their evaluation frequencies. The latter case is an over-approximation that introduces additional, irrelevant evaluations of the common subexpression. This might decrease the performance of the monitor, so CSE is only applied if the least common multiple coincides with one of the accessing frequencies. In this case, the transformation is always beneficial.

4 RTLola Specific Optimizations

This section introduces transformations around the concept of pacing types. Since these types are specific to the specification language RTLola, the transformations are as well. The concepts, however, apply to similar languages as well. We introduce the *Pacing Type Refinement* and the *Filter Refinement* as such transformations.

4.1 Pacing Type Refinement

In this subsection, we describe a transformation refining the pacing type of output streams. Consider the following specification as an example. Note, the inferred pacing types are marked gray, whereas the black ones are annotated explicitly.

```
input alt, lat: Float64
output check_alt @{alt}
  := alt < b0
output check_lat @{lat}
  := lat ∈ [b1, b2]
trigger @{alt ∧ lat}
  ¬(check_alt ∧ check_lat)
```

```
input alt, lat: Float64
output check_alt @{alt ∧ lat}
  := alt < b0
output check_lat @{alt ∧ lat}
  := lat ∈ [b1, b2]
trigger @{alt ∧ lat}
  ¬(check_alt∧check_lat)
```

The specification shows a simple geofence, i.e., it checks if the altitude and latitude values are in the specified bounds. Each expression only accesses one input stream, so the specification infers the pacing types `@{alt}` and `@{lat}` for the output streams. The trigger then accesses all output stream values and notifies the user if a bound is violated. Transitively, the trigger accesses all input streams, so its inferred pacing type is `@{alt ∧ lat}`. With this type, the monitor evaluates the trigger iff all input streams receive a new value at the same time. Consequently, whenever an event arrives that does not cover both input streams, the output stream computations are in vain. This justifies refining the pacing types of the output streams to mirror the pacing type of the trigger, which is exactly what the Pacing Type Refinement transformation does.

For event-based streams, the transformation finds the most specific activation condition that does not change the observable behavior. This goal is achieved by annotating a stream with a pacing type that is the disjunction of all pacing types accessing it. For periodic streams, the transformation proceeds similarly. Here, the explicit type annotation is the slowest frequency such that each stream access is still valid, i.e., the least common multiple of each accessing frequency, similar to Section 3.2.

Note that the pacing type transformation of a stream s is only possible if all accesses to s are synchronous, i.e., $(s_j^-, 0, s^\uparrow) \in E$. Otherwise, the transformation might change the observable behavior, as illustrated with the following example. Consider a sliding window in a trigger condition targeting a stream $s^\uparrow$. Assume further that the transformation changes the pacing type $s^\uparrow.pt$ from 2Hz to 1Hz. As a result, $s^\uparrow$ produces fewer values, changing the result of the sliding window and thus the trigger as well.

The transformation resolves transitive dependencies by applying a fix-point iteration.

4.2 Filter Refinement

RTLola is free of side effects and thanks to its evaluation order, it has a static program flow. The static program flow, however, also has a drawback: if a stream

s conditionally accesses a stream s', s' will always be evaluated before the condition is resolved. This problem can be circumvented by integrating the condition occurring in the expression of s into the filter of s'.

Consider the following specifications:

```
input pilots : Float64
input emergency : Bool
output check_1
 @{emergency ∧ pilots}
  := num_pilots > 0

output check_2
 @{emergency ∧ pilots}
  := num_pilots == 2

trigger @{emergency ∧ pilots}
if !emergency then check_1 else
    check_2
```

```
input pilots : Float64
input emergency : Bool
output check_1
 @{emergency ∧ pilots}
 { filter !emergency }
  := pilots > 0
output check_2
@{emergency ∧ pilots}
 { filter emergency }
  := num_pilots == 2
trigger @{emergency ∧ pilots}
if !emergency then
 check_1.hold(or: true)
else check_2.hold(or: true)
```

Both specifications check the number of pilots in the cockpit. Depending on whether or not the plane is in emergency mode, one or two pilots are adequate. Because of the static evaluation order, the monitor with the specification on the left always computes the values of both output streams. However, the final trigger only uses one of the streams, depending on the `emergency` input. Thus, the monitor can avoid half of the output computations. The specification on the right show how this can be achieved using Filter Refinement. The transformation adds filters to all streams accessed in the consequence or alternative of a conditional expression. Additionally, it replaces the synchronous lookups to these streams with asynchronous lookups and adds explicit type annotations. The former prevents the type inference from adding the filter to the trigger as well. The latter is necessary because the type of the trigger can no longer be inferred without the synchronous lookups. Similar to previous transformations, Filter Refinement takes direct and transitive dependencies into account.

The algorithm for this transformation consists of four parts: In the first step, it identifies conditional expressions. Afterward, it constructs the filter condition for the synchronously accessed streams based on the condition following four rules. If a stream is accesses in a) the condition, it does not add any filter condition. b) the consequence, it adds a filter containing the if-condition. c) the alternative, it adds a filter containing the negation of the if-condition. d) a nested conditional, it builds the conjunction of the conditions. e) the consequence and the alternative of a nested conditional, it combines the filter conditions with a disjunction. f) the consequence and the alternative of a non-nested conditional, it does not add a filter. After building the filter conditions for the synchronously accessed streams, the transformation adds the filter to the stream. If the stream already had one, the transformation builds the conjunction of both. It then changes the affected synchronous lookups to asynchronous ones to prevent the type inference from adapting its own filter. This process is repeated until a fix-

point is reached. Note that the transformation is only possible for synchronous lookups, otherwise the transformation alters the observable behavior.

5 Evaluation

We evaluate our transformations using the interpreter of the RTLola framework [7].[4] We compare the monitor executions with enabled and disabled compiler transformations for a specification checking whether an aircraft remains within a geofence [2]. The traces for the evaluation consists of 10,000 randomly generated events. Each execution was performed ten times on a 2.9GHz Dual-Core Intel Core i5 processor.

The geofence specification was selected due to its high practical relevance. It checks if the monitored aircraft leaves a polygonal area, i.e., the zone for which the aircraft has a flight permission. If the monitor raises a trigger, the vehicle has to start an emergency landing to prevent further damage. The specification computes the approximated trajectory of the vehicle to decide whether a face of the fence was crossed.

The shape of the fence is determined statically, so the gradient and y-intercept of the faces are constants in the original specification. We generalized the specification slightly for our case study. This makes the specification more maintainable without forsaking performance thanks to the SCCP transformation. In a geofence with five faces, the SCCP propagates and eliminates 48 constants streams. This roughly halves the execution time of the monitor as can be seen in the first graph of Fig. 1.

In the second evaluation, we extended the specification by a third dimension, also taking the altitude of the aircraft into account. The altitude of the aircraft is independent of the longitude and latitude, rendering computations of the output streams unobservable for events not covering all three dimensions. Here, the Pacing Type Refinement places explicit type annotations on 32 streams in the specification with five faces. The new trace contains a new reading for the altitude every 100ms and for the longitude and latitude every 10ms. The impact of the Pacing Type Refinement can be seen in the second graph of Fig. 1: the monitor for the transformed specification is roughly three times faster.

To evaluate the impact of the Filter Refinement, we adapt the specification to perform a violation check for an under-approximation of the geo-fence. The more costly precise geo-fence check is only performed if the under-approximation reports a violation. This specification shows the potential impact of the Filter Refinement transformation, which adds filters to 27 output streams for a geofence with five faces. The first two columns in the third graph of Fig. 1 illustrate the results of the executions with a trace that is most of the time within the under-approximated fence. Surprisingly, the specification after the transformation is about three times slower than the original specification.

The reason lies within the evaluation process of the monitor. Filters increase the number of nodes in the dependency graph, thus triggering new evaluation

[4] http://rtlola.org

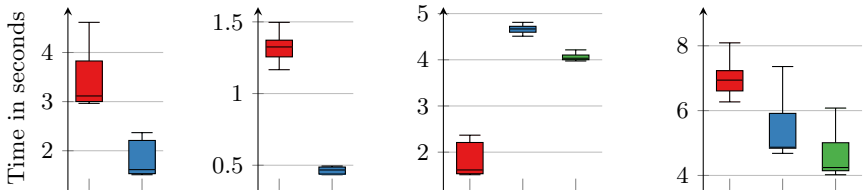

Fig. 1: From left to right, the graphs show the impact of SCCP, Pacing Type Refinement, Filter Refinement without, and with pre-existing filters. Red boxes are the running time before applying the respective transformation, blue after, and green by additionally applying CSE.

steps. In our example, this produces an overhead that is higher than the performance benefits gained by adding filters. The last graph in Fig. 1 shows the results for a specification like that for the same input trace. Here, the transformation reduces the execution time by about 30%.

When now also applying the CSE as well, 27 filter conditions and one if condition can be summarized in a common subexpression. This yields another 5% performance gain as can be seen in the last two graphs in Fig. 1.

6 Conclusion

Since the safety of the monitored system rests on the quality of the monitoring specification, it is crucially important that specifications are easy to understand and maintain. The code transformations presented in this paper contribute towards this goal. By taking care of performance considerations, the transformations help the user to focus on writing clear specifications.

Monitoring languages are, in many ways, similar to programming languages. It is therefore not surprising that classic compiler optimization techniques like Sparse Conditional Constant Propagation and Common Subexpression Elimination are also useful for monitoring. Especially encouraging, however, is the effect of our new Pacing Type and Filter Refinements. In our experiments, the transformations improved the performance of the monitor as much as threefold. This could be a starting point for a new branch of runtime verification research that, similar to the area of compiler optimization in programming language theory, focusses on the automatic transformation and optimization of monitoring specifications.

In future work, our immediate next step is to integrate further common code transformations into our framework. We will also investigate the interplay between the different transformations and develop heuristics that choose the best transformations for a specific specification. A careful understanding of the impact on the monitoring performance is especially needed for transformations that prolong the evaluation order, such as Common Subexpression Elimination and Filter Refinement.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (August 2006)
2. 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.) CAV 2020. LNCS, vol. 12225, pp. 28–39. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_3
3. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: FPGA stream-monitoring of real-time properties. ACM Trans. Embedded Comput. Syst. **18**(5s), 88:1–88:24 (2019). https://doi.org/10.1145/3358220
4. Berry, G.: Proof, language, and interaction: essays in honour of Robin Milner, chap. The foundations of Esterel, pp. 425–454. MIT Press (2000)
5. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: Tessla: Temporal stream-based specification language. In: Massoni, T., Mousavi, M.R. (eds.) SBMF 2018. LNCS, vol. 11254, pp. 144–162. Springer (2018). https://doi.org/10.1007/978-3-030-03044-5_10
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL '77. p. 238–252. Association for Computing Machinery, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973
7. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: Streamlab: Stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 421–431. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_24
8. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. CoRR **abs/1711.03829** (2017), http://arxiv.org/abs/1711.03829
9. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: A declarative language for synchronous programming of real-time systems. In: Proc. Conference on Functional Programming Languages and Computer Architecture. pp. 257–277. Springer (1987)
10. Gérard, L., Guatto, A., Pasteur, C., Pouzet, M.: A modular memory optimization for synchronous data-flow languages: Application to arrays in a lustre compiler. SIGPLAN Not. **47**(5), 51–60 (Jun 2012). https://doi.org/10.1145/2345141.2248426
11. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNSC, vol. 11237, pp. 282–298. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_16
12. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proc. of IEEE **79**(9), 1305–1320 (1991)
13. Kildall, G.A.: A unified approach to global program optimization. In: POPL '73. p. 194–206. Association for Computing Machinery, New York, NY, USA (1973). https://doi.org/10.1145/512927.512945
14. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1998)
15. Potop-Butucaru, D.: Fast Redundancy Elimination Using High-Level Structural Information from Esterel. Tech. Rep. RR-4330, INRIA (Nov 2001), https://hal.inria.fr/inria-00072257
16. Schwenger, M.: Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS. Master thesis, Saarland University (2019)
17. Seidl, H., Wilhelm, R., Hack, S.: Compiler Design - Analysis and Transformation. Springer (2012). https://doi.org/http://dx.doi.org/10.1007/978-3-642-17548-0