Monitoring Cyber-Physical Systems: From Design to Integration*

Maximilian Schwenger

CISPA Helmholtz Center for Information Security
maximilian.schwenger@cispa.saarland

Abstract. Cyber-physical systems are inherently safety-critical. The deployment of a runtime monitor significantly increases confidence in their safety. The effectiveness of the monitor can be maximized by considering it an integral component during its development. Thus, in this paper, I given an overview over recent work regarding a development process for runtime monitors alongside a cyber-physical system. This process includes the transformation of desirable safety properties into the formal specification language RTLola. A compiler then generates an executable artifact for monitoring the specification. This artifact can then be integrated into the system.

1 Introduction

Cyber-physical systems (CPS) directly interact with the physical world, rendering them inherently safety-critical. Integrating a runtime monitor into the CPS greatly increases confidence in its safety. The monitor assesses the health status of the system based on available data sources such as sensors. It detects a deterioration of the health and alerts the system such that it can e.g. initiate mitigation procedures. In this paper I will provide an overview regarding the development process of a runtime monitor for CPS based on recent work. For this, I will use the RTLola [13, 14] monitoring framework.

The process ranges from designing specifications to integrating the executable monitor. It starts by identifying relevant properties and translating them into a formal specification language. The resulting specification is type-checked and validated to increase confidence in its correctness. Afterwards, it is compiled into an executable artifact, either based on software or hardware. Lastly, the artifact is integrated into the full system. This step takes the existing system architecture of the CPS into account and enables the monitor to support a post-mortem analysis. The full process is illustrated in Figure 1.

The first step of the process concerns the specification. It captures a detailed analysis of the system behavior, which entails computationally challenging arith-

* 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).

metic. Yet, since the monitor for the specification will be realized as an embedded component, its resource consumption must be statically bounded. Thus, the specification language has to provide sufficient expressiveness while allowing the monitor to retain a predictable and low resource footprint. In particular, an ideal specification language provides formal guarantees on the runtime behavior of its monitors such as worst case execution time or memory consumption. In general, however, expressiveness, formal guarantees, and predictably low resource consumption cannot be achieved at the same time. Desirable properties like "every request must be granted within a second" might come at the cost that the memory consumption of the monitor depends on the unpredictable input frequency of requests. Consequently, specification languages providing input-independent formal guarantees on the required memory must impose restrictions to prohibit such properties. These restriction can be direct, i.e., the syntax of the language renders the property inexpressible, or indirect, so the property can be expressed but falls into a language fragment unsuitable for monitoring. RTLola falls into the former category.

During the design phase of the CPS, the specifier defines properties spanning from validation of low-level input sensor readings to high-level mission-specific control decisions. The former are real-time critical, i.e., they demand a timely response from the monitor, whereas the latter include long-term checks and statistical analysis [6] where slight delays and mild inaccuracies are unsubstantial. Just like code is not a perfect reflection of what the programmer had in mind, the specification might deviate from the specifiers intention. To reduce the amount of undetected bugs, the specification needs to be validated. This increases confidence in it and — by proxy — in the monitor. The validation consists of two parts: type checks and validation based on log data. The former relies solely on the specification itself and checks for type errors or undefined behavior. The latter requires access to recorded or simulated traces of the system and interprets the specification over the given traces. The output of the monitor can then be compared against the expected result.

After successfully validating the specification, a compiler for the specification language generates an executable artifact. This artifact is either a hardware or a software solution, depending on the requirements of the system architecture. If, for example, the architecture does not allow for adding additional components, a software solution is preferable as it does not require dedicated hardware; the monitor can be part of the control computer. Hardware solutions, on the other hand, are more resource efficient and allow for parallelization with nearly-0 cost. In any case, the compiler can inject additional annotations for static code-level verification [15] or traceability [5] to further increase confidence in the correctness of the monitor.

Finally, deploying the monitor into the CPS harbors additional pitfalls. As an external safety component, the monitor should not influence the regular operation of the system unless upon detection of a safety violation. As a result, the system architecture needs to enable non-intrusive data flow from the system to the monitor and intrusive data flow from the monitor to the controller.

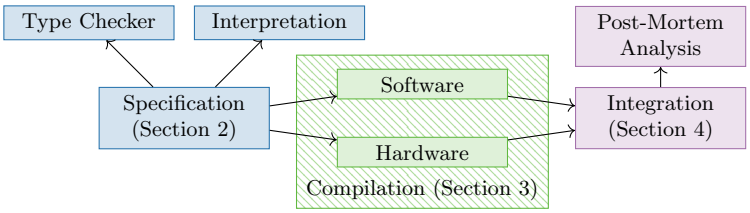


Fig. 1: Illustration of the paper structure. It is divided into three phases: specification, compilation, and integration.

The controller then has to react on an alarm appropriately. Such a reaction can e.g. be a switch from the regular controller to a formally verified controller with significantly reduced complexity responsible for a graceful shutdown of the system [16], as suggested in the Simplex Architecture [37].

After terminating a mission, the output of the monitor provides valuable data for the post-mortem analysis. Regular system logs might be insufficient as they do not contain the entire periphery data due to resource constraints. The monitor, however, filters and aggregates the data specifically to assess information regarding the system's status w.r.t. safety, thus providing valuable feedback.

2 Specifications: From Sensors to Missions

When designing specifications for CPS, the specifier has to keep in mind that not all properties are equal. They fall into a spectrum from low-level properties concerned with concrete sensor readings to high-level properties validating mission-specific criteria. Properties on the least end of the spectrum work on raw data points of single sensors. Most common are simple bounds checks (*the altitude may not be negative*) or frequency checks (*the barometer must provide between 9 and 11 readings per second*). Less low-level properties work on refined data points, e.g. to check whether several sensors contradict each other (*the altitude measured by the sonic altimeter must not deviate more than ϵ from the altitude based on the air pressure*). Such a sensor cross-validation requires refinement of raw values as they cannot be compared without further ado. While a barometer provides the air pressure, it needs further information such as pressure and temperature at sea level to accurately estimate the current altitude. Similarly, validating the position provided by the *global navigation satellite system* (GNSS) module against the position estimated by the *inertial measurement unit* (IMU) requires double integration of the measured acceleration. On the highest end of the spectrum reside mission-level properties. When checking such properties, the source of information is mostly discarded and the values are assumed to be correct. For example, consider an aircraft that traverses a set of dynamically received waypoints. Mission-level properties could demand that a waypoint is reached in time or that the traveled distance does deviate more than a factor from the actual distance between two points.

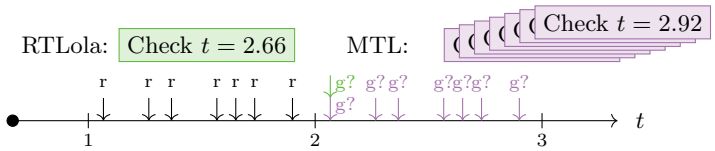


Fig. 2: Illustration of monitor obligations for checking if a request ("r") is granted ("g?") within a second. While the MTL interpretation is more precise, it requires statically unbounded memory. The RTLola under-approximation requires constant memory.

Properties are usually expressed in natural language as above and translated into a specification language. Consider the first property: *the altitude may not be negative*. Evidently, the properties harbors little challenge in terms of arithmetic. However, timeliness is critical. If the altimeter reports a negative altitude, clearly something is wrong and the system needs to be informed near-instantaneously. In RTLola, the property translates to the following specification:

```
input altitude: Float32
input orientation: Float32
trigger altitude < 0 "Altimeter reports negative altitude."
trigger orientation > 2 * π "Orientation exceeds 2π."
```

The first two lines declare input streams of name `altitude` and `orientation`, both with type `Float32`. The remaining lines contain trigger conditions with message to be sent to the system for the case a condition turns true. Whenever the monitor receives a new value from the altimeter or gyroscope, the respective condition is checked immediately. Note that RTLola allows for *asynchronous* input behavior, i.e., one input stream can produce a new value while the other does not. Thus, when the gyroscope produces a value, the respective trigger condition is checked regardless of the altimeter. This timing dependency from inputs to expressions is part of RTLola's type system.

The type system is two-dimensional: every stream and expression has a *value type* and a *pacing type*. Value types are common within programming languages, they indicate the shape and interpretation of data. The input streams, for example, are of value type `Float32`, so storing a single value requires 32 bits and the bits should be interpreted as a floating point number. The pacing type, however, states *when* expressions are evaluated and thus when streams produce new values. In case of the trigger expressions, the pacing types are *event-based*, i.e., they are coupled to the reception of new values from the altimeter or gyroscope.

The pacing type can also be *periodic*, effectively decoupling the evaluation of expressions from input streams in terms of timing. As an example, consider the second low-level property: *the barometer must provide between 9 and 11 readings per second*. An RTLola specification for this property is:

```
input pressure: Float32
output readings_per_sec @ 1Hz := pressure.aggregate(over: 1s, using:
    count)
```

```
trigger readings_per_sec < 9 "Barometer produces too few readings."
trigger readings_per_sec > 11 "Barometer produces too many readings."
```

Here, `readings_per_sec` is an output stream with a timing annotation `@ 1Hz` prompting the monitor to only evaluate the expression of the stream once per second. Thus, the timing of the evaluation is decoupled from the reception of new input values.

The expression itself is a sliding window aggregation, i.e., whenever evaluated, the expression counts how many data points the barometer generated in the last second. If the count falls below 9 or exceeds 11, the monitor raises an alarm. While the logic behind a an efficient sliding window implementation is rather complex and requires a great deal of bookkeeping, RTLola provides a simple primitive for it. This alleviates the need for the specifier to manually take care of the implementation details.

Note that the specification does not precisely represent the property. Assume the system alternates between receiving 7 readings in the first half of a second and receiving 3 readings in the second half. Then, every other second, the system receives a total of 14 readings per second — unbeknownst to the monitor. In RTLola, it is impossible to specify the property correctly as it lacks the necessary expressiveness by design: sliding window expressions can only occur in streams with a timing annotation. This annotation renders the stream *isochronous*, i.e., the point in time when its expression will be evaluated are determined statically. The reason behind it is that the original properties lies in a category of properties that generally need a statically unbounded amount of memory to be monitored. To understand this, consider the property *Every request must be granted within one second*. A sound monitor for the property needs to check whether a request was granted exactly one second after receiving a request. However, there is no static bound on the amount of requests the monitor receives within this frame of time. Since it has to store their arrival times, the memory consumption might exceed any bound. The problem is illustrated in Figure 2. There are specification logics such as *metric temporal logic* (MTL) [20], in which the property can be expressed. In such a case, the memory consumption of the monitor is linear in the number of events receives within the second. Since RTLola only provides constant memory monitors, it rejects specifications for such properties and instead enables constant-memory under-approximations. This design decision is a requirement to guarantee that the monitor cannot possible run out of memory during the execution.

RTLola provides primitives for more abstract constraints such as sensor cross-validations as well:

```
input velocity_1: Int64
input velocity_2: Int64
output deviation := abs(velocity_1 - velocity_2)
output lasting_dev := deviation > 5 ∧ deviation.offset(by: -1,
    default: 0) > 5 ∧ deviation.offset(by: -2, default: 0) > 5
trigger lasting_deviation "Lasting deviation in measured velocities."
```

The specification declares two input streams providing different readings for the velocity of the system, and two output streams `deviation` and `lasting_dev` that computes the absolut deviation of the readings and checks whether the deviation exceeds a threshold three consecutive times. The first conjunct of the stream expression accesses the current, i.e., the latest value of the `deviation` stream, whereas the `offset(by: -n, default: v)` function allows for accessing the nth-to-latest value of the stream for $n \in \mathbb{N}$[1]. This value does not exist at the beginning of the monitor execution, so the specifier has to supply a default value v. Here, the specification refers to the abstract notion of "the last value" rather than considering the real-time behavior, assuming that low-level validation already took place.

Note that `deviation` accesses both `velocity` streams without supplying a default value. This indicates a *synchronous* access and prompts the monitor to only evaluate `deviation` when both input receive a new value. This is not necessarily the case since RTLola considers inputs to be asynchronous. The pacing type of `deviation` captures the information that the stream is only evaluated when the two inputs happen to arrive at the same time: it is event-based and the conjunction of both input streams. In contrast to that, a different definition of `deviation` could look like:

```
output deviation_disj @ velocity_1 ∨ velocity_2 :=
    abs(velocity_1.hold(or: 0) - velocity_2.hold(or: 0)
```

Here, the output stream has a disjunctive type, so when it is extended, at least one of the two inputs received a new value, not necessarily both. In such a case, RTLola forces the specifier to declare precisely how it should handle potentially old values. The specifier can, as in the example of `deviation_disj`, turn the synchronous accesses into sample and hold accesses. When evaluating the expression, the monitor will access the latest — yet potentially old — value of the input stream with a 0-order hold. If the specifier attempted to access either stream synchronously, RTLola would reject the specification because it contains an inner contradiction. These kinds of type checks greatly increase confidence in the correctness of the specification as they point out imprecise and potentially flawed parts.

Lastly, consider two mission-level properties for an aircraft flying a dynamic list of waypoints: the monitor should issue a warning if the aircraft deviated from the straight-line distance by at least ε, and it should issue an error if such a deviation occurred more than 10 times.

```
input wp: (Float64, Float64)
input pos: (Float64, Float64)
output start: (Float64, Float64) := (0, 0)
output exp_dist @ wp := wp - wp.offset(by: -1, default: start)
output dist_since_wp @ pos := pos - pos.offset(by: -1, default: start)
  + dist_since_wp.offset(by: -1, default: 0)
output distance_deviation @ wp :=
```

[1] As a result, RTLola does not allow for accessing future values.

```
  abs(exp_dist.offset(by: -1, default: 0) - dist_since_wp.hold(or: 0))
trigger distance_deviation > ε "Warn: Path deviation detected."
output deviations := deviations.offset(by: -1, default: 0)
  + if distance_deviation > ε then 1 else 0
trigger deviations > 10 "Err: Too many path deviations!"
```

The specification declares two clearly asynchronous input streams: the current waypoint `wp` and the current position `pos`. The `start` stream is a constant stream only containing the initial position of the aircraft. `exp_dist` contains the distance between the current and the last waypoint whereas `dist_since_wp` aggregates the distance the aircraft has traveled since reaching the last waypoint/starting the mission. The deviation in distance is then the absolut difference between these two distances. Note that this value is only valid when the aircraft just reached a new waypoint, hence the `@ wp` annotation. This prompts the monitor to only evaluate the stream when it receives a new waypoint. Lastly, the `deviations` stream counts the number of times the deviation exceeded its threshold.

The specification contains several pacing type annotations. This, however, is for illustration, as most of the time, RTLola can infer both types from the stream expression. Yet, the specifier always has the option to annotate types for clarity or if the timing of a stream should deviate from the standard behavior, e.g. for disjunctive event-based types.

Note that this was only a brief overview of RTLola. For more details on the theory, refer to [36], and for the implementation, refer to [13].

2.1 Specification Validation by Interpretation

RTLola's type system already rules out several sources for incorrect behavior of the monitor. Yet, a validation of the specification is crucial to increase confidence in the *correctness* of the specification. The validation requires access to records of previous runs of the system. These can be simulated, collected during test runs, or logs from sufficiently similar systems. Just like when testing software, developers annotate the trace data with points in time when they expect the monitor to raise an alarm. Then, they execute a monitor for the given specification on the trace and compare the result with their annotations. Deviations mainly root from either an error when designing the specification, or a discrepancy in the mental image of different people regarding the correct interpretation of a property.

A key point for the specification validation is that the process should incur as little cost as possible to enable rapid prototyping. Hence, interpreting the specification rather than compiling it is preferable — especially when the target platform is hardware-based. After all, realizing a specification on an FPGA usually takes upwards of 30 min [7]. While interpretation is considerably less performant than compiled solutions, the RTLola interpreter manages to process a single event in 1.5 µs. This enables a reasonably fast validation of specifications even against large traces.

2.2 Static Analysis for RTLola Specifications

After type checking the specification and validating its correctness based on test traces, RTLola provides static checks to further analyze it. For this, RTLola generates a dependency graph where each stream is a node and each stream access is an edge. This information suffices to perform a memory analysis and a running time analysis. The analysis identifies the resource consumption — both spatial and temporal — of each stream, granting fine-grained control to the specifier.

Memory For the memory consumption of stream s, the analysis identifies the access of another stream to s with the greatest offset n_s^*. Evidently, the monitor only has to retain n_s^* values of s to successfully resolve all accesses. Moreover, note that all types in RTLola have a fixed size. Let T_s be the type of s with bit-size $|T_s|$. Then, the memory consumption of s induced by accesses through other streams amounts to $n_s^* \cdot |T_s|$.

Sliding window expressions within the stream expression of s incur additional memory overhead. Suppose $w_1, \ldots, w_k$ are the windows occurring in s where for $w_i = (\gamma_i, d_i)$, γ_i is the aggregation function and d_i is the length of the window. If $k > 0$, RTLola demands s to be periodic with frequency π_s. The memory consumption of s induced by sliding windows consists of the number of *panes* required. Here, a pane represents the time interval between two consecutive evaluations of the window. The pane consists of a single value which contains the aggregated information of all values that arrived in the respective time interval. This implementation of sliding windows is inspired by Li et al. [23] and only works for list homomorphisms [25]. A window thus has $d_i \cdot \pi_s$ panes, which has to be multiplied by the size of the value stored within a pane: T_{γ_i}. This value is statically determined and depends on the aggregation function: for summation, it is merely the sum of the values, for the average it is the intermediate average plus the number of values that occurred within the pane.

The overall memory consumption of s is therefore

$$\mu(s) = n_s^* \, |T_s| + \mathbb{1}_{k>0} \sum_{i=1}^{k} d_i \pi_i \, |T_{\gamma_i}|$$

Here, $\mathbb{1}_\varphi$ is the indicator function evaluating to 1 if φ is true and 0 otherwise.

Running Time The running time cannot be fully determined based on the specification alone as it depends on the hardware of the CPS. For this reason, RTLola provides a preliminary analysis that can be concretized given the concrete target platform. The preliminary analysis computes a) the complexity of each evaluation cycle given a certain event or point in time, and b) the parallelizability of the specification.

For the former metric, note that the monitor starts evaluation cycles either when it receives an event, or at predetermined points in time (*deadlines*). An event always updates a set of input streams and a statically determined set of

output streams. Recall the mission-level specification computing the deviation from the flight path including the `deviation_disj` stream. The specification declares two input streams, thus allowing for three possible non-empty events. An event covering either `velocity` stream but not the other only triggers an evaluation of `deviation_disj`. Only if the event covers both inputs, `deviation` and the trigger are evaluated as well.

Consider a specification containing periodic streams and suppose the monitor has a deadline at time t. It then evaluates all periodic streams due at t, i.e., all streams with frequency π where $\pi \cdot t$ is a natural number. Thus, the set of streams affected by an evaluation cycle is pre-determined.

The next step in the analysis is concerned with *evaluation layers* They are closely related to the parallelizability of the monitor as they indicate how many stream evaluations can take place at once. The analysis yields a partition of the set of streams where all streams within an element of the partition are *independent*, enabling a parallel evaluation. The (in-)dependence relation is based on the dependency graph. If a stream accesses another *synchronously*, i.e., without an offset, than the target stream needs to be evaluated before the accessing stream. This definition entails an *evaluation order* on output streams. The aforementioned partition is then the coarsest partition such that any two streams in the same set are incomparable with respect to the transitive closure of the evaluation order. Each element of the partition is an evaluation layer. By construction, streams within the same layer can be evaluated in an arbitrary order — in particular also in parallel. The order in which layers are evaluated, however, still needs to follow the evaluation order. In the example specification before, the partition would be `{{wp, pos, start} < {exp_dist, dist_since_wp} < {distance_deviation} < {deviations, trigger_warn} < {trigger_err}}`.

The evaluation layer analysis immediately provides information regarding the parallelizability of the monitor. The running time analysis takes the number of evaluations into account as well as how many streams are affected by an evaluation cycle, and how complex their expressions are. Intuitively, if an event or deadline affects streams in a multitude of layers, then the evaluation is slow as computations depend on each other and thus require a sequential order. Conversely, if an event only affects few streams, all within the first layer, the evaluations are independent and thus highly parallelizable. As a result, the running time of the monitor is low.

Note, however, that for software monitors the degree to which computations should run in parallel requires careful consideration, since spawning threads incurs a constant overhead. For hardware monitors, the overhead does not apply.

3 Compilation: Generating Monitors

While interpretation of specifications enables rapid prototyping, its logic is far more complex than a compiled monitor, at the same time resulting in subpar performance. This renders compilation preferable. Additionally, the compiler can inject additional information into the generated code. Such annotations can

benefit the certification process of the CPS either by providing a notion of traceability, or by outright enabling the static verification of the monitor. The target platform of the compilation can either be hardware or software, both coming with advantages and drawbacks.

In this section, I will present and discuss a hardware compilation for RTLola specifications, and a software compilation for Lola, i.e., a subset of RTLola, with verification annotations.

3.1 RTLola on FPGA

Realizing an RTLola specification on hardware has several advantages. For one, the hardware monitor does not share resources with the controller of the system apart from power, eliminating potential negative interference. Moreover, special purpose hardware tends to be smaller, lighter, and require less energy than their general purpose counterparts. Secondly, hardware enables parallel computations at minimal cost. This synergizes well with RTLola, where output streams within the same evaluation layer can be evaluated in parallel.

The realization of RTLola on hardware [7] works in two steps: an RTLola specification is translated into VHDL code, out of which an FPGA (field-programmable gate array) implementation is synthesized. The synthesis provides additional static information regarding the required board size in terms of memory[2] and lookup-up tables. This allows for validating whether the available hardware suffices to host the monitor. Moreover, the synthesis indicates the idle and peak power consumption of the monitor, information that is invaluable when integrating the monitor into the system.

The aforementioned advantages are not only valid for FPGA but also for other hardware realization such as application-specific integrated circuits (ASIC) and complex programmable logic device (CPLD). While ASIC have significant advantages over FPGA when it comes to mass-producibility, power consumption and performance, FPGA are preferable during the development phase as they are orders of magnitude cheaper, have a lower entry barrier, and allow for rapid development. CPLD, on the other hand, are just too small to host realistic/non-trivial specifications.

Hardware Realization Managing periodic and event-based streams under a common hardware clock poses the key challenging when realizing an RTLola monitor in hardware. Yet, this distinction only affects the part of the monitor logic deciding *when* to evaluate stream expressions; the evaluation itself is agnostic to it. For this reason, the monitor is split into two modules. The *high-level controller* (HLC) is responsible for scheduling evaluations, i.e., to decide when and which streams to evaluation. It passes the information down to the second module, the *low-level controller* (LLC), which is responsible for managing the

[2] The hardware realization might require temporary registers and working memory. This can slightly increase the computed memory consumption.

evaluation. A FIFO queue between the controllers buffers information sent from the HLC to the LLC.

Recall the specification from Section 2 checking for strong deviations in readings of two velocimeters. As a running example for the hardware compilation, we extend the specification by the following declarations.

```
output avg_dev @10mHz := dev.aggregate(over: 10min, using: avg)
trigger avg_dev > 4 "High average deviation."
```

The specification checks whether two velocimeters disagree strongly over three consecutive measurements and whether the average disagreement is close to the disagreement-threshold. Note that all streams are event-based except for `avg_dev` and the second trigger.

Figure 3 illustrates the overall structure of the monitor. As can be seen, the HLC accepts input events from the monitored system. Such an event has a fixed size of $2 \cdot (32 + 1)$ bits, i.e., 32 due to the input types and an additional bit per stream to indicate whether the stream received an update. For periodic streams, the HLC has access to the system clock. Based on the current time and arrival of events, the HLC triggers evaluation cycles by sending relevant information to the queue while rising the *push* signal. Such a data package consists of $2 \cdot (32 + 1) + 64 + 5$ bits. The first component of the sum represents the potential input event. If the evaluation only serves to update periodic streams, these bits will all be 0. The following 64 bits contain the timestamp of the evaluation, crucial information for the computation of sliding window expressions. The last 5 bits each represent an output stream or trigger and indicate whether the respective entity is affected by the evaluation cycle. As a result, the LLC does not have to distinguish between event-based and periodic streams; it merely has to evaluate all streams the HLC marked as affected.

The communication between the queue and the LLC consists of three data lines: the *pop* bit is set by the LLC and triggers the queue to send another data packet down to it — provided the *empty* bit is 0. In this case, the queue puts the oldest evaluation information on the *dout* wires.

Internally, the LLC consists of two state machines. The first one handles the communication with the queue. While the first machines resides in the *eval* state, the second state machine manages the evaluation. To this end, it cycles through different states, each representing an evaluation layer. The first state ("1") copies the information about input stream updates into the respective memory region. In each consecutive state, the monitor enables the modules responsible for evaluating the respective stream expression by raising the *enable* bit. It then waits on the *done* bits. Upon receiving all of them, the monitor proceeds to the next state. During this process, the outputs of trigger expressions are not persisted locally, but directly piped down to the monitored system.

Resource Consumption When compiling the specification into VHDL and realizing it on a ZYNQ-7 ZC702 Evaluation Board, using the Vivado Design Suite[3],

[3] https://www.xilinx.com/products/design-tools/vivado.html

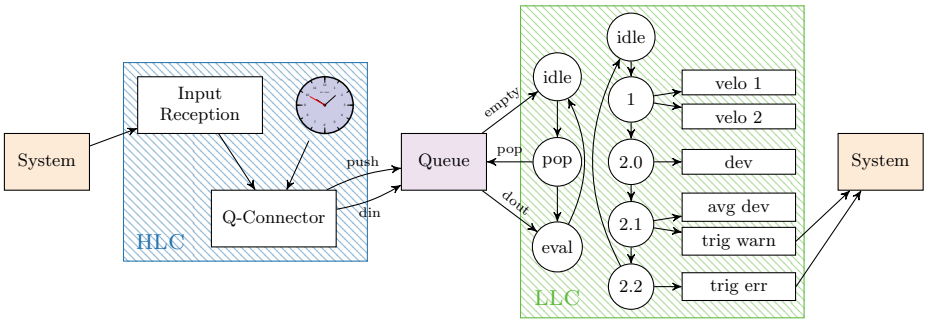


Fig. 3: Illustration of the hardware realization of an RTLola monitor. It is composed of two components connected by a queue. The HLC receives inputs from the system and manages the timing of periodic and event-based streams. The LLC controls the evaluation process with a state machine where each state represents an evaluation layer of the underlying specification. The LLC passes violations of safety properties on to the system.

the hardware synthesizer provides information regarding the overall resource consumption. In this case, the monitor requires 10,700 lookup tables and 1735 bits of memory. The energy consumption amounts to 144 µW when idle, and 1.699 W under peak pressure. Even though the specification is rather small, it gives a glimpse at how low the resource consumption actually is. Baumeister et al. [6] successfully synthesized larger specifications designed for autonomous aircraft on the same FPGA.

3.2 Lola to Rust

While a compilation of RTLola specifications into software is a topic for future work, a compilation for Lola does exist, presented by Finkbeiner et al. [15]. Lola [10] is a synchronous and discrete-time subset of RTLola. As such, it does not have a notion of real-time, thus neither sliding windows nor periodic streams are an issue. Moreover, Lola assumes all input streams to receive new values at the same time, prompting all output streams to be extended as well. This renders sample and hold accesses obsolete. Lola does, however, allow for future lookups, i.e., a stream may refer to the *next* value of another stream.

The example specification for the software compilation is another mild variation of the velocimeter cross-validation from Section 2. The modification replaces the `lasting_dev` stream by the following:

```
output lasting_dev := dev > 5 ∧ dev.offset(by: +1, default: 0) > 5 ∧
    dev.offset(by: -1, default: 0) > 5
```

Here, `lasting_dev` access the last, current and *next* value of `deviation`.

The compilation presented in this section translates the Lola specification into Rust[4] code that enables a static verification. Rust as a target language comes with several advantages. First, as a system language with an LLVM[5] backend, it is compatible with a wide array of platforms. Secondly, a key paradigm of the language is to enforce static checks on the code and thus reduce dynamic failures. This goes hand in hand with the goal of verifying the functional correctness and absence of dynamic failures of the generated monitor. Lastly, Rust allows for fine-grained control low-level constructs such as memory management, enabling the programmer — or in the case, the Lola compiler — to write highly performant code.

The compiler injects verification annotations into the code as well. This enables the static verifier Viper to prove functional correctness of the monitor in two steps. First, it relates the working memory of the monitor to the semantic model of Lola. The key challenge here is that the semantics of Lola argues about infinite data sequences while the monitor operates on a finite working memory. Next, the verification relates the verdict of the monitor, i.e., the boolean indicator of whether a trigger should go off is correct given the current state of the working memory. In combination we can conclude that the monitor only emits an alarm if the semantic model demands so.

Dealing with Fallible Accesses While future offsets provide a natural way to specify temporal dependencies, the monitor has to compensate for them by delaying the evaluation of the accessing streams. Thus, the evaluation of `lasting_dev` needs to be delayed by one step since they access a future value of `dev`. This delay is propagated through the dependency graph: the trigger transitively accesses a future value, so its evaluation needs to be delayed, too.

With the delay operation in place, accesses via a future offset will always succeed up until the system terminates, and thus no longer produces new inputs. In this case, the monitor continues to evaluate delayed streams until they have the same length as the input streams. This phase is the *postfix* phase of the monitor execution. Here, future offsets fail because the accesses values do not exist and never will. Similarly, past offsets fail at the beginning of the monitor execution, the *prefix*.

In the very first iteration of the monitor, only the inputs and `dev` can be evaluated, the other output stream and the trigger are delayed. In the next iteration, the input is updated and all output streams and the trigger are evaluated. Evaluating `lasting_dev` accesses both values of `dev`. In addition to that, the past lookup refers to the -1st value of `altitude`, a value, that will never exist. Thus, the monitor statically substituted the access with the default value.

Clearly, the monitor goes through three phases: a prefix, in which past offsets fail unconditionally, a loop phase, in which both past and future offsets succeed unconditionally, and a postfix phase, in which future offsets fail unconditionally. In light of this, the compiler faces a trade-off: it can generate a general-purpose

[4] https://www.rust-lang.org/
[5] https://llvm.org/

```
Prelude
struct Memory { ... }
impl Memory { ... }
[[ Evaluation Functions ]]
fn get_input() ->
      Option<(T_{s_1}, ..., T_{s_ℓ})> {
  [[ Communicate with system ]]
}
fn emit(output: &(T_{s_1}, ..., T_{s_n})) {
  [[ Communicate with system ]]
}
fn main() {
  let mut memory = Memory::new();
  let early_exit  = prefix(&mem);
  if !early_exit  {
    while let Some(input) = get_input() {
      mem.add_input(&input1);
      [[ Evaluation Logic ]]
    }                          Monitor Loop
  }
  postfix(&mem);
}
```

```
fn prefix(mem: &mut Memory) -> bool {
 if let Some(input) = get_input() {
   mem.add_input(&input);
     [[ Evaluation Logic ]]
   } else {
     return true // Jump to Postfix.
   }
   [[ Repeat η_φ^← times. ]]
   false  // Continue with Monitor Loop.
}                              Execution Prefix
```

```
fn postfix(mem &Memory) {
 [[ Evaluation Logic ]]
 [[ Repeat η_φ^→ times. ]]
}                              Execution Postfix
```

Listing 1.1: Structure of the generated Rust code. The prelude is highlighted in orange, the monitor loop in blue, the execution prefix in green, and the execution postfix in violet.

loop containing conditional statements resolving offsets dynamically, or it can take the three phases into account by generating code specific to them. The former option contains conditional statements not found in the original specification, resulting in far less performant code. The latter option, however, requires more code, resulting in a larger executable file. The compiler outlined in this section opts for the latter option.

Listing 1.1 illustrates the abstract structure of the generated Rust code. The first portion of the code is the *prelude* containing declaration for data structures and I/O functions. Most notably, the `Memory` struct represents the monitor's working memory and is of a fixed size. For this, it utilizes the memory analysis from Section 2.2. Note also, that the `get_input` function returns an optional value: either it contains new input data or it indicates that the system terminated.

The `main` function is the entry point of the monitor. It allocates the working memory and transitions to the prefix. Here, the monitor code contains a static repetition of code checking for a new input, and evaluating all streams. In the evaluation, stream access are either translated into immediate access to memory or substituted by constant default values. The `prefix` function returns a boolean flag indicating whether the system terminated before the prefix was completed. This prompts the `main` function to jump to the postfix immediately. Otherwise, the main monitor loop begins following the same scheme of the prefix: retrieve new input values, commit them to memory, evaluate streams, repeat until the system terminates. Note that all stream accesses during the monitor loop translate to accesses to the working memory. Lastly, the `main` function triggers the

computation of the postfix. The structure is similar to the prefix except that it does not check for new input values.

The evaluation logic for streams is a straight-forward translation of the Lola specification as conditional statements, constants, and arithmetic functions are syntactically and semantically almost identical in Lola and Rust. Only stream accesses requires special attention as they boil down to accesses to the `Memory` struct. Lastly, the compilation has to order the evaluation of streams to comply with the evaluation order from Section 2.2. Streams in the same evaluation can be ordered arbitrarily or parallelized. The latter leads to a significant runtime overhead and only pays off if computational complexity of the stream expressions is sufficiently high.

Verification The compilation injects verification annotations in the Rust code to enable the Prusti [1] plugin of the Viper framework [28] to verify functional correctness of the monitor. The major challenge here is to verify that the finite memory available to the monitor suffices to accurately represent the infinite evaluation model of Lola. The compilation achieves this by introducing a dynamically growing list of values, the ghost memory. With this, the correctness proof proceeds in two steps. First, whenever the monitor receives or computes a new value, it commits it both to the ghost memory and the working memory. Here, the working memory acts as a ring buffer: as soon as its capacity is reached, the addition of a new value overwrites and thus evicts the oldest value. Therefore, the working memory is an excerpt of the ghost memory and thus the evaluation model. Ergo, the computation of new values is valid with respect to the evaluation model because memory accesses yield the same values as the evaluation model would. The newly computed, correct values, are then added into the memory, concluding the inductive proof of memory compliance.

Secondly, the verdict of a trigger in the theoretical model needs to be equivalent to the concrete monitor realization. This amounts to proving that the trigger condition was translated properly. Here, memory accesses are particularly interesting, because the theoretical computation uses entries of the ghost memory whereas the realization accesses the working memory only. The agreement of the ghost memory and the working memory for the respective excerpts conclude this proof.

Note that for the monitor the ghost memory is write-only, whereas the verification procedure "reads" it, i.e., it refers to its theoretical values. The evaluation logic of the monitor uses data from the system to compute output values. Different components of the verification than access these values either directly or over the ghost memory (GM). Clearly, the information flow is unidirectional: information flows from the monitor to the verification but not vice versa. As a result, upon successful verification, the ghost memory can safely be dissected from the realization.

Conclusion Not only does the compilation from Lola to Rust produce performant runtime monitors, the injection of verification annotations answers the

question *"Quis custodiet ipsos custodes?*[6]" rendering it an important step into the direction of verified runtime monitor. The applicability of Lola for CPS is limited to high-level properties where neither asynchrony, nor real-time play a significant role. Further research in this direction especially relating to RTLola can significantly increase the value and practical relevance of the compilation.

4 Integration and Post-Mortem

A key task when integrating a monitor into a CPS is finding a suitable spot in the system architecture. Improper placement can lead to ineffective monitoring or negative interference, jeopardizing the safety of the entire system.

4.1 Integration

The integration is considerably easier when the architecture is not yet fully determined. When adding the monitor retroactively, only minimal changes to the architecture are possible. Larger changes would render previous tests void since the additional component might physically change the system, e.g. in terms of weight distribution and power consumption, or logically offset the timing of other components. Consider, for example, a monitor that relies on dedicated messages from the controller as input data source. If the processor of the controller was already almost fully utilized, the additional communication leads to the controller missing deadlines. This can lead to safety hazards, as timing is critical for the safe operation of a CPS. Taking the placement of the monitor into account early on increases the degree of freedom, which helps avoid such problems.

The amount of interference the monitor imposes on the system also depends on the method of instrumentation. Non-intrusive instrumentation such as bus snooping grants the monitor access to data without affecting other modules. The effectiveness of this approach hinges on the amount of data available on the bus.

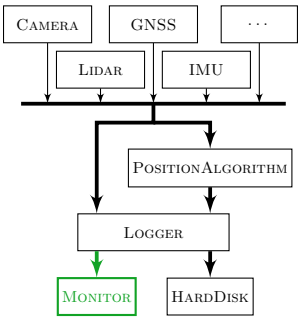
Consider, for example, the system architecture of the autonomous aircraft superARTIS of the German Aerospace Center (DLR) depicted in Figure 4. When integrating a monitor for the optical navigation rail into the aircraft [6], the monitor was placed near the logging component. By design, the logger had access to all relevant data. This enabled monitoring of properties from the entire spectrum: The specification contained single-sensor validation, cross-validation, and geo-fence compliance checks. Note that in this particular case, the utilization of the logger was low. This allowed it to forward the information from the bus to the monitor. In a scenario where performance is critical, snooping is the preferable option.

4.2 Post-Mortem Analysis

After termination of a flight, the post-mortem analysis allows for assessing the performance of the system and finding culprits for errors. The analysis relies on

[6] *"Who will guard the guards themselves?"*

(a) An image of the superARTIS aircraft.



(b) A schematic of the superARTIS system architecture.

Fig. 4: Information on the superARTIS aircraft of the German Aerospace Center.

data recorded during the mission; a full record enables a perfect reconstruction of the execution from the perspective of the system. Resource restrictions, however, limit the amount of data, so full records are often not an option. Thus, data needs to be filtered and aggregated rigorously.

A main task of the monitor is exactly this: refining input data by filtering and aggregation to obtain an accurate assessment of the system state. Based on this assessment, the monitor determines whether a property is violated. While this binary verdict is the major output of the monitor, the intermediate results provide valuable insight into the evolution of the system over time. Hence, logging this data can improve the post-mortem analysis and alleviates the need to filter and aggregate data in another component as well.

5 Bibliographic Remarks

Early work on runtime monitoring was mainly based on temporal logics [12, 17, 18, 21, 22, 33]. Their notion of time was limited to discrete time, leading to the development of real-time logics like STL [24] and MTL [20]. Not only do algorithms for runtime monitoring exist for these logics [3, 4, 11, 29], there is also work realizing it on an FPGA [19]. The R2U2 [27, 35] tool in particular implements MTL monitors on FPGA while allowing for future-time specifications. Further, there are approaches for generating verified monitors for logics [2, 34].

Apart from these temporal logics, there are other specification languages specifically for CPS such as differential dynamic logic [32]. The ModelPlex [26] framework translates such a specification into several verified components monitoring both the environment w.r.t. the assumed model and the controller decisions.

Other approaches — such as the one presented in this paper — completely forgo logics. Similar to the compiler from Lola to Rust, there is a verified compiler

for synchronous Lustre [8] programs to C code. Moreover, the Copilot [30, 31] toolchain is based on a functional, declarative, stream language with real-time capabilities. Copilot enables the verification of generated monitors using the CBMC model checker [9]. As opposed to the verification with Viper, their verification is limited to the absence of various arithmetic errors, lacking functional correctness.

In terms of integration, both the R2U2 [27] and the Copilot [31] tool were successfully integrated into an aircraft.

6 Conclusion

In this paper, I provided an overview of recent work on the development of a runtime monitor for cyber-physical systems from design to integration. The process can roughly be divided into three phases. In the specification phase, specifiers transform natural language descriptions of properties into a suitable formal specification language. Such properties range from low-level properties validating a single sensor to high-level properties overseen the quality of the entire mission. Type checking and validation based on log data from previous or simulated missions increase confidence in the specification. The compilation phase transforms the specification into an executable software or hardware artifact, potentially injecting annotations to enable static verification of the monitor. This process can help increase the effectiveness of the monitor, which directly translates into safer systems.

Acknowledgements

This paper is based on a tutorial at the 20th International Conference on Runtime Verification. The work summarized in this paper is based on several earlier publications [6, 7, 13–15] and I am grateful to all my co-authors. I especially would also like to thank Jan Baumeister and Bernd Finkbeiner for providing valuable feedback and comments.

References

1. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust Types for Modular Specification and Verification. Proc. ACM Program. Lang. **3**(OOPSLA), 147:1–147:30 (2019). https://doi.org/10.1145/3360573
2. Basin, D.A., Dardinier, T., Heimes, L., Krstic, S., Raszyk, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12166, pp. 432–453. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_25
3. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015). https://doi.org/10.1145/2699444

4. Basin, D.A., Krstic, S., Traytel, D.: AERIAL: almost event-rate independent algorithms for monitoring metric regular properties. In: RV-CuBES 2017. pp. 29–36 (2017)
5. Baumeister: Tracing Correctness: A Practical Approach to Traceable Runtime Monitoring. Master thesis, Saarland University (2020)
6. Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., Torens, C.: RTLola cleared for take-off: Monitoring autonomous aircraft. In: CAV 2020. LNCS, vol. 12225, pp. 28–39. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_3
7. 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
8. Bourke, T., Brun, L., Dagand, P., Leroy, X., Pouzet, M., Rieg, L.: A formally verified compiler for lustre. In: Cohen, A., Vechev, M.T. (eds.) PLDI 2017. pp. 586–601. ACM (2017). https://doi.org/10.1145/3062341.3062358
9. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
10. 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: TIME 2005. pp. 166–174. IEEE Computer Society Press (June 2005)
11. Deshmukh, J.V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., Seshia, S.A.: Robust online monitoring of signal temporal logic. Formal Methods in System Design **51**(1), 5–30 (2017). https://doi.org/10.1007/s10703-017-0286-7
12. Drusinsky, D.: The temporal rover and the ATG rover. In: SPIN Model Checking and Software Verification. pp. 323–330 (2000). https://doi.org/10.1007/10722468_19
13. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: StreamLAB: Stream-based Monitoring of Cyber-Physical Systems. In: CAV 2019. LNCS, vol. 11561, pp. 421–431. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_24
14. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time Stream-based Monitoring. CoRR **abs/1711.03829** (2017), `http://arxiv.org/abs/1711.03829`
15. Finkbeiner, B., Oswald, S., Passing, N., Schwenger, M.: Verified Rust Monitors for Lola Specifications. In: RV 2020. LNCS, Springer (2020)
16. Finkbeiner, B., Schmidt, J., Schwenger, M.: Simplex architecture meets RT-Lola. In: MT@CPSWeek 2020 (2020), `https://www.react.uni-saarland.de/publications/FSS20.pdf`
17. Finkbeiner, B., Sipma, H.: Checking finite traces using alternating automata. Formal Methods in System Design **24**(2), 101–127 (2004). https://doi.org/10.1023/B:FORM.0000017718.28096.48
18. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: TACAS 2002. pp. 342–356 (2002). https://doi.org/10.1007/3-540-46002-0_24
19. Jaksic, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Nickovic, D.: From signal temporal logic to FPGA monitors. In: MEMOCODE 2015. pp. 218–227 (2015). https://doi.org/10.1109/MEMCOD.2015.7340489
20. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems **2**(4), 255–299 (1990). https://doi.org/10.1007/BF01995674
21. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design **19**(3), 291–314 (2001). https://doi.org/10.1023/A:1011254632723

22. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: PDPTA 1999. pp. 279–287 (1999)
23. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Rec. **34**(1), 39–44 (2005). https://doi.org/10.1145/1058150.1058158
24. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: FORMATS 2004 and FTRTFT 2004. pp. 152–166 (2004). https://doi.org/10.1007/978-3-540-30206-3_12
25. Meertens, L.: Algorithmics: Towards programming as a mathematical activity (1986)
26. Mitsch, S., Platzer, A.: Modelplex: verified runtime validation of verified cyber-physical system models. Formal Methods Syst. Des. **49**(1-2), 33–74 (2016). https://doi.org/10.1007/s10703-016-0241-z
27. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. Formal Methods Syst. Des. **51**(1), 31–61 (2017). https://doi.org/10.1007/s10703-017-0275-x
28. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2
29. Nickovic, D., Maler, O.: AMT: A property-based monitoring tool for analog systems. In: FORMATS 2007. pp. 304–319 (2007). https://doi.org/10.1007/978-3-540-75454-1_22
30. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A Hard Real-Time Runtime Monitor. In: RV 2010. LNCS, vol. 6418, pp. 345–359. Springer (2010). https://doi.org/10.1007/978-3-642-16612-9_26
31. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Copilot: Monitoring Embedded Systems. ISSE **9**(4), 235–255 (2013). https://doi.org/10.1007/s11334-013-0223-x
32. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reasoning **41**(2), 143–189 (2008). https://doi.org/10.1007/s10817-008-9103-8
33. Pnueli, A.: The Temporal Logic of Programs. In: FOCS 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
34. Schneider, J., Basin, D.A., Krstic, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 310–328. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_18
35. Schumann, J., Moosbrugger, P., Rozier, K.Y.: R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. In: RV 2015. LNCS, vol. 9333, pp. 233–249. Springer (2015). https://doi.org/10.1007/978-3-319-23820-3_15
36. Schwenger, M.: Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS. Master thesis, Saarland University (2019)
37. Sha, L.: Using simplicity to control complexity. IEEE Software **18**(4), 20–28 (2001), https://doi.org/10.1109/MS.2001.936213