Monitoring with Verified Guarantees

Jan Baumeister[1], Johann C. Dauer[2], Bernd Finkbeiner[1] and Sebastian Schirmer[2*]

[1]Helmholtz Center for Information Security (CISPA), Saarbrücken, Germany.
[2*]German Aerospace Center (DLR), Braunschweig, Germany.

*Corresponding author(s). E-mail(s): sebastian.schirmer@dlr.de;
Contributing authors: jan.baumeister@cispa.de; johann.dauer@dlr.de; finkbeiner@cispa.de;

Abstract

Runtime monitoring is generally considered a light-weight alternative to formal verification. In safety-critical systems, however, the monitor itself is a critical component. For example, if the monitor is responsible for initiating emergency protocols, as proposed in a recent aviation standard, then the safety of the entire system critically depends on the correctness of the monitor. In this paper, we present a verification extension to the Lola monitoring language that extends the efficient specification of the monitor with Hoare-style annotations that guarantee the correctness of the monitor specification. We add two new operators, assume and assert, which specify assumptions of the monitor and expectations on its output, respectively. The validity of the annotations is established by an integrated SMT solver. We report on experience in applying the approach to specifications from the avionics domain, where the annotation with assumptions and assertions has lead to the discovery of safety-critical errors in specifications. The errors range from incorrect default values in offset computations to complex algorithmic errors that result in unexpected temporal patterns. We also report how verified specifications can be monitored efficiently at runtime.

Keywords: Formal methods, Cyber-physical systems, Runtime Verification, Hoare Logic

1 Introduction

Cyber-physical systems are inherently safety-critical due to their direct interaction with the physical environment – failures are unacceptable. A means of protection against failures is the integration of reliable monitoring capabilities. A *monitor* is a system component that has access to a wide range of system information, e.g., sensor readings and control decisions. When the monitor detects a failure, i.e., a violation of the behavior stated in its *specification*, it notifies the system or activates recoveries to prevent failure propagation.

The task of the monitor is critical to the safety of the system, and its correctness is therefore of utmost importance. Runtime monitoring approaches like Lola [8, 11] address this by describing the monitor in a formal specification language, and then generating a monitor implementation that is provably correct and has strong runtime guarantees, for example on memory consumption. Formal monitoring languages typically feature temporal [24] and sometimes spatial [21] operators that simplify the specification of complex monitoring behaviors. However, the specification itself, the central part of runtime monitoring, is still prone to human errors during specification development. Hence, how can we check that the monitor specification itself is correct?

In this paper, we introduce a verification feature to the Lola framework. Specifically, we

extend the specification language with *assumptions* and *assertions*. The framework statically verifies that the assertions are guaranteed to hold if the input to the monitor satisfies the assumptions. This verification feature was previously introduced in [9]. Here, we extend this work by providing a proof of soundness and presenting an online monitoring approach with experimental results that checks the satisfaction of assumptions during runtime to activate assertion checks.

The prime application area of Lola is unmanned aviation. Lola is increasingly used for the development and operation monitoring of unmanned aircraft; for example, the Lola monitoring framework has been integrated into the DLR unmanned aircraft superARTIS[1] [2] using an FPGA realization [3]. The verification extension presented in this paper is motivated by this work. In practice, system engineers report that support for specification development is necessary, e.g., sanity checks and proofs of correctness. Additionally, recent developments in unmanned aviation regulations and standards indicate a similar necessity. One such development is the industry standard F3269-21 (Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions) by ASTM International [2]. ASTM F3269-21 introduces a certification strategy based on a Run-Time Assurance (RTA) architecture that bounds the behavior of a complex function by a safety monitor [20], similar to the well-known Simplex architecture [27]. This complex function could be a Deep Neural Network as proposed in [7]. A simplified version of the architecture[3] of ASTM F3269-21 is shown in Figure 1.

At the core of the architecture is a safety monitor that takes the inputs and outputs of the complex function, and decides whether the complex function behaves as expected. If not, the monitor switches the control from the complex function to a matching recovery function. For instance, the flight of an unmanned aircraft could be separated into different phases: e.g., take-off, cruise flight, and landing. For each of these phases,

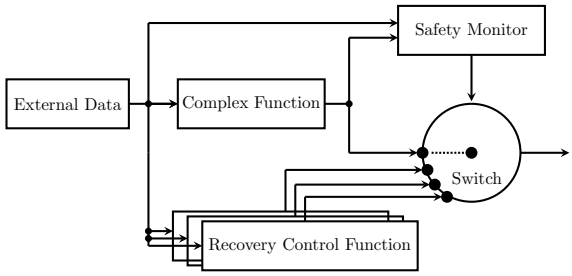


Fig. 1: Run-Time Assurance architecture proposed by ASTM F3269-21 to safely bound a complex function using a safety monitor.

a dedicated recovery could be defined, e.g., braking during take-off, the activation of a parachute during cruise flight, or a go-around maneuver during landing. Further, it is crucial that recoveries are only activated under certain conditions and that only one recovery is activated at a time. For instance, a parachute activation during a landing approach is considered safety-critical. The verification extension of Lola introduced in this paper can be used to guarantee statically that such decisions are avoided within the monitor specification. Consider the simplified Lola specification

```
input event_a, event_b, value: Bool, Bool, Float32
assume <a1> !(event_a and event_b)
output braking  : Bool := ...computation...
output parachute : Bool := ...computation...
output go_around : Bool := ...computation...
assert <a1> !(braking and parachute)
```

that declares an assumption on the system input `events` and asserts that `braking` and `parachute` never evaluate to *true* simultaneously.

In the following, we first give a brief introduction to the stream-based specification language Lola, then present the verification approach, and give details on the tool implementation and our tool experience with specifications that were written based on interviews with aviation experts. Last, we consider the case where assumptions might not be satisfied during runtime. Our results show that standard Lola specifications are indeed prone to error, and that these errors can be caught with the formal verification introduced by our extension.

[1] https://www.dlr.de/content/en/research-facilities/superartis-en.html

[2] https://www.astm.org/

[3] In its original version, data is separated into assured and unassured data and data preparation components are added.

Related Work

Most work on the verification of monitors focuses on the correct transformation into a general programming language. For example, Copilot [22] specifications can be compiled into C code with constant time and memory requirements. Similarly, there is a translation validation toolkit for Lola monitors implemented in Rust [11], which is based on the Viper verification tool [19]. Translation validation of this type is orthogonal to the verification approach of this paper. Instead of verifying the correctness of a transformation, our focus is to verify the specification itself. Both activities complement each other and facilitate safer future cyber-physical systems.

Our verification approach is based on classic ideas of inductive program verification [12, 16], and is closely related to the techniques used in static program verifiers like KeY [4], Why3 [6], and Dafny [18]. In a verification approach like Dafny, we are interested in functional properties of procedures, specified as post-conditions that relate the values upon the termination of the procedure with those at the time of entry to the procedure, e.g., *ensure y = old(y)*. By contrast, a stream-based language like Lola allows arbitrary access to past and future stream values. This makes it necessary to *unfold* the Lola specification in order to properly relate the assumptions and assertions in time.

Most closely related to stream-based monitoring languages are synchronous programming languages like LUSTRE [15], ESTEREL [5], and SIGNAL [13]. For these languages, the compiler is typically used for verification – a program representing the negation of desired properties is compiled with the target program and a check for emptiness decides whether the properties are satisfied. Furthermore, a translation from past linear-time temporal logic to ESTEREL was proposed to simplify the specification of more complex temporal properties [17]. Other verification techniques also exist like SMT-based k-Induction for LUSTRE [14] or a term rewriting system on synced effects [28]. A key difference in our approach is that we do not rely on compilation. Our verification works at the level of an intermediate representation. Furthermore, synchronous programming languages are limited to past references, while the stream unfolding for the inductive correctness proof of the Lola specification includes both past and future temporal operators. Similar to k-Induction, our approach is sound but not complete.

2 Runtime Monitoring with Lola

Lola is a stream-based language that describes the translation from input to output streams:

$$\begin{aligned}
&\textbf{input } t_1 : T_1\\
&\qquad\vdots\\
&\textbf{input } t_m : T_m\\
&\textbf{output } s_1 : T_{m+1} := e_1(t_1, \ldots, t_m, s_1, \ldots, s_n)\\
&\qquad\vdots\\
&\textbf{output } s_n : T_{m+n} := e_n(t_1, \ldots, t_m, s_1, \ldots, s_n)\\
&\textbf{trigger } \varphi \ \mathit{message}
\end{aligned}$$

where input streams carry synchronous arriving data from the system under scrutiny, output streams represent calculations, and triggers generate notification *message*s at instants where their condition φ becomes *true*. Input streams $t_1, \ldots, t_m$ and output streams $s_1, \ldots, s_n$ are called *independent* and *dependent variables*, respectively. Each variable is typed: independent variables t_i are typed T_i and dependent variables s_i are typed T_{m+i}. Dependent variables are computed based on *stream expressions* $e_1, \ldots, e_n$ over dependent and independent stream variables. A (stream) expression is one of the following:

- an atomic expression c of type T if c is a constant of type T;
- an atomic expression s of type T if s is a stream variable of type T;
- an expression $\mathit{ite}(b, e_1, e_2)$ of type T if b is a Boolean expression and e_1, e_2 are expressions of type T. Note that *ite* abbreviates the control construct *if-then-else*;
- an expression $f(e_1, \ldots, e_k)$ of type T if $f : T_1 \times \cdots \times T_k \mapsto T$ is a k-ary operator and $e_1, \ldots, e_k$ are expressions of type $T_1, \ldots, T_k$;
- an expression $o.\mathit{offset}(\mathit{by} : i).\mathit{defaults}(\mathit{to} : d)$ of type T if o is a stream variable of type T, i is an Integer, and d is of type T.

For example, consider the Lola specification

```
input altitude: Float32 // in m
output altitude_bound := altitude > 200.0
trigger altitude_bound "Warning: Decrease altitude!"
```

that notifies the system if the current `altitude` is above its operating limits, i.e., `200.0` meters. Note that stream types are inferred, i.e., `altitude_bound` is of type `Bool`.

Lola uses temporal operators that allow output streams to access its and others previous and future stream values. The stream

```
output alt_count := if altitude ≤ 200.0 then 0
    else alt_count.offset(by: -1).defaults(to: 0) + 1
```

represents a count of consecutive altitude violations by accessing its own previous value, i.e., `offset(by: x)` where a negative and positive integer `x` represents past and future stream accesses, respectively. Since temporal accesses are not always guaranteed to exist, the default operator defines values which are used instead, i.e., `defaults(to: d)` where `d` has to be of the same type as the used stream. Here, at the first position of `alt_count` the default value zero is taken. As abbreviations for the temporal operators, `alt_count[x, d]` is used. Further, `s[x..y, d, ○]` for $x < y$ abbreviates `s[x,d]` ○ `s[x+1,d]` ○ ... ○ `s[y,d]` where ○ is a binary operator. Using `alt_count > 10` as a trigger condition is preferable if only persistent violations should be reported.

In general, Lola is a specification language that allows to specify complex temporal properties in a precise, concise, and less error-prone way. The focus is on *what* properties should be monitored instead of *how* a monitor should be executed. Therefore, the Lola monitor synthesis automatically infers and optimizes implementation details like evaluation order and memory management. The evaluation order [11] of Lola streams is automatically derived by analysis of the *dependency graph* [8] of the specification. This allows to ignore the order when taking advantage of the modular structure of Lola output streams, e.g.,:

```
output alt_avg := alt_count / (position+1)
output alt_count := if altitude ≤ 200.0 then 0
    else alt_count.offset(by: -1).defaults(to: 0) + 1
output pos := pos.offset(by: -1).defaults(to: 0)
```

where `pos` and `alt_count` are used before their definition. Further, the graph allows to detect all invalid cyclic stream dependencies, e.g., `output a := a.offset(by:0).defaults(to:0)`.

3 Assumptions and Assertions

In this section, we present the verification extension for the Lola specification language. The extension allows the developer to annotate the Lola specification with *assumptions* and *assertions* in order to verify the desired guarantees on the computed streams. As an example, consider the simplified specification in Listing 1, which is structured into stream computations in Lines 1 to 28, and assumptions and assertions from Line 30 onwards.

The computation part specifies a safety monitor within a RTA architecture that triggers recovery functions for three different flight phases. First, the take-off recovery function is triggered (Line 24) when the targeted take-off speed was not achieved on a runway up to a predefined point (Lines 14-15). The distance between the current position and the end of the runway with local coordinates $(0, 0)$ is computed in Line 9. Second, in-flight a parachute is activated (Line 26) when virtual barriers for the aircraft, i.e., a geofence, are exceeded (Line 17) [26]. Last, during landing, up to a point of no return (`alt < 10.0`), a new landing attempt is initiated (Line 27) if the aircraft's speed is too fast or its landing gear is not yet ready. To be more robust, the current and the previous value of the `landing_gear_ready` is taken into account (Lines 19-21).

With the verification extension, the specification assures that recoveries are not activated simultaneously (Lines 34-36), i.e., for instance there is no possibility that a parachute is activated during a landing approach. The first two conjunctions in Lines 34 and 35 evaluate to *false* because relevant outputs use a disjoint altitude condition. The last conjunction requires an assumption. Here, two assumptions are linked by the identifier `a1` to the assertion. The assumptions specify: the known bound of received speed data (Line 31) as well as operational information (Line 30), e.g., given by the concept of operation a nominal landing is only foreseen within the predefined operational airspace. Note that assumptions are provided by the user and are *assumed* to be valid. Further, a second assertion is stated in Line 38 that guarantees that *the parachute should only be activated when the aircraft is 100 meters above ground*. In this case, the property can be shown assumption-free. Assertions help engineers

```
input alt : Float32 // Height above ground
input x, y : Float32, Float32 // Position
input speed : Float32 // Velocity of aircraft
input landing : Bool // Indicates landing mode
// Status of landing gear
input lg_status : (Float32,Float32,FLoat32)

// Complex computations
output dst_on_runway      : Float32 := √(x² + y²)
output geofence_violation : Bool    := ...
output landing_gear_ready : Bool    := ...

// Take-off contingency
output decelerate := alt < 1.0 ∧ speed < 10.0
                   ∧ dst_on_runway > 20.0
// In-flight contingency
output parachute := geofence_violation ∧ alt > 100.0
// Landing contingency
output gain_alt := landing ∧ alt ≥ 10.0
                ∧ (landing_gear_ready[-4..0, true, ∧]
                → speed > 10.0)

// Notifications to the system
trigger decelerate "RECOVERY: Stop take-off by
                    decelerating aircraft."
trigger parachute  "RECOVERY: Activate parachute."
trigger gain_alt   "RECOVERY: Gain altitude for next
                    landing attempt."

assume <a1> ¬(landing ∧ geofence_violation)
assume <a1> abs(speed) ≤ 80.0

// Only one contingency is activated at once.
assert <a1> ¬( (decelerate ∧ parachute)
             ∨ (decelerate ∧ gain_alt)
             ∨ (parachute ∧ gain_alt) )
// Parachute only activated 100 m above ground.
assert <a2> parachute → alt > 100.0
```

Listing 1: A simplified Run-Time Assurance Lola specification with three recovery functions for three different flight phases. Assumptions and assertions are used to show that only one recovery function is activated at once.

to show that certain properties are *true*. The given assertions indicate how specification debugging and management can benefit from the extension – it avoids digging into complex computations.

The extension and its verification approach are presented in the following. In general, the verification extension is used if a Lola specification is annotated in the following way:

$$\begin{aligned} \textbf{assume } \langle \alpha_1 \rangle \quad & \theta_1 \\ & \vdots \\ \textbf{assume } \langle \alpha_m \rangle \quad & \theta_m \\ \textbf{assert } \langle \alpha_{m+1} \rangle \quad & \psi_1 \\ & \vdots \\ \textbf{assert } \langle \alpha_{m+n} \rangle \quad & \psi_n \end{aligned}$$

where $\alpha_1, \ldots, \alpha_{m+n} \in \Gamma$ are identifiers for $\theta_1, \ldots, \theta_m, \psi_1, \ldots, \psi_n$, which are Boolean stream expressions with possibly temporal operators. For convenience, we define functions which return all θ and ψ that are linked to a given α identifier: $assume(\alpha) = \{\theta_j \mid \forall \alpha_j \in \Gamma, \alpha = \alpha_j\}$ and $assert(\alpha) = \{\psi_j \mid \forall \alpha_j \in \Gamma, \alpha = \alpha_j\}$. The set of assertion $\psi_1, \ldots, \psi_n$ is *correct* for all input streams if and only if whenever an assumption is satisfied, its corresponding assertion is satisfied as well.

The verification of assertions relies on the encoding of the Lola execution in Satisfiability Modulo Theory (SMT). We define the *smt* function that encodes a stream expression in Def. 1. It can be used to encode independent and dependent variables as well as expressions of assumptions and assertions.

Definition 1 (SMT-Encoding of Stream Expressions) Let Φ be a Lola specification over independent stream variables $t_1, \ldots, t_m$ and dependent stream variables $s_1, \ldots, s_n$. Further, let the natural number $N + 1$ be the length of the input streams, c be an SMT constant symbol, and $\tau_1^0, \ldots, \tau_1^N, \ldots, \tau_m^0, \ldots, \tau_m^N, \sigma_1^0, \ldots, \sigma_1^N, \ldots, \sigma_n^0, \ldots, \sigma_n^N$ be SMT variables. Then, the function *smt* recursively encodes a stream expression e at position j with $0 \leq j \leq N$ in the following way:

- Base cases:
  - $smt(c)(j) = \texttt{c}$
  - $smt(t_i)(j) = \tau_i^j$
  - $smt(s_i)(j) = \sigma_i^j$
- Recursive cases:
  - $smt(f(e_1, \ldots, e_n))(j) = \texttt{f}(smt(e_1)(j), \ldots, smt(e_n)(j))$
  - $smt(ite(e_b, e_1, e_2))(j) = \texttt{ite}(smt(e_b)(j), smt(e_1)(j), smt(e_2)(j))$
  - $smt(e[k, c])(j) = \begin{cases} smt(e)(j+k) & \text{if } 0 \leq j+k \leq N, \\ c & \text{otherwise} \end{cases}$

where `ite` is an SMT encoding of *if-then-else*; `f` is an interpreted function if f is from a theory supported by the SMT solver and an uninterpreted function otherwise.

Next, Proposition 1 proves the correctness of asserted stream properties for finite input streams. If the set of assertions is correct, asserted stream

properties are guaranteed to be valid in each step of the monitor execution. In practice, such specifications are preferable. In the following, let Φ be a Lola specification with verification annotations. Further, we refer to the set of input streams and computed output streams as stream execution.

Proposition 1 (Assertion Verification of a Finite Stream Execution)
Let Φ be a Lola specification and let $s_1, \ldots, s_n$ be dependent stream variables used in Φ. The set of assertions is correct for a finite stream execution with length $N+1$ under given assumptions, if the following formula is valid:

$$\bigwedge_{0 \leq i \leq N} \Big(\bigwedge_{\alpha \in \Gamma} \Big(\bigwedge_{\theta \,\in\, assume(\alpha)} smt(\theta)(i) \wedge \bigwedge_{s_k \in \Phi} \sigma_k^i = smt(e_k)(i) \rightarrow \bigwedge_{\psi \,\in\, assert(\alpha)} smt(\psi)(i) \Big) \Big)$$

The formula in Proposition 1 unfolds the complete stream execution and informally expresses that an assertion must hold in each stream position whenever its corresponding assumption and implementation are satisfied.

To avoid the complete unfolding and allow arbitrary stream lengths, an inductive argument is given in Proposition 2 that defines proof obligations for an annotated Lola specification. Next, we present a template for the stream unfolding that helps to define the proof obligation at the *Begin*ning (Definition 3), during *Run* (Definition 4), and at the *End* (Definition 5) of a stream execution.

Definition 2 (Template Stream Unfolding)
We define the template formula ϕ_t that states proof obligations as:

$$\begin{aligned} &\bigwedge_{\alpha \in \Gamma} \Big(\bigwedge_{i \in p\_asm} \Big(\bigwedge_{\theta \in assume(\alpha)} smt(\theta)(i) \Big) \\ &\wedge \bigwedge_{i \in p\_asserted} \Big(\bigwedge_{\psi \in assert(\alpha)} smt(\psi)(i) \Big) \\ &\wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0 < k \leq n} \sigma_k = smt(e_k)(i) \Big) \\ &\rightarrow \bigwedge_{i \in p\_assert} \Big(\bigwedge_{\psi \in assert(\alpha)} smt(\psi)(i) \Big) \Big) \end{aligned}$$

where *p_asm*, *p_asserted*, *p_streams*, and *p_assert* are template parameters. They are sets of positions for the unfolding of assumptions, previously proven assertions, output streams, and assertions, respectively.

The template formula in Definition 2 uses template parameters for the stream unfolding. For instance, the parameter assignment $p\_asm := \{i \mid 0 \leq i < 10\}$ adds assumptions at the first ten positions of the stream execution. Further, the parameter *p_asserted* allows to incorporate the induction hypothesis.

In the following, we will use the Lola specification in Listing 2 as a running example for the template stream unfolding.

Here, the input *reset* represents a reset command for the output stream *o1* that counts how long no *reset* occurred. Output *o1* is used by output *o2* which aggregates over the previous, the current, and the next outcome of *o1*. This is achieved by using the offset operator, e.g., `o1[-1, 0]` accesses the previous value of `o1` if it exists, otherwise it takes the default value `0`. As assertion, we show that *o2* is always positive and never larger than three given the assumption that in each execution step either the previous or the next *reset* is *true*. The assumption ensures that at most two consecutive *resets* are *false*. Given the *reset* sequence of input values $\langle true; false; false \rangle$ that satisfies the assumption, the resulting *o1* stream evaluates to $\langle 0; 1; 2 \rangle$. Here, at the second position of the sequence, *o2* evaluates to three. To show that the assertion also holds at the first and the last position of the sequence, out-of-bounds values must be considered.

We show how the template ϕ_t can be used at the beginning of a stream execution. Here, default values due to past stream accesses beyond the beginning of a stream need to be captured by the obligation to guarantee that the assertions hold

```
assume<a1> reset[-1, false] ∨ reset[1, false]
input reset : Bool
output o1 := if reset then 0 else o1[-1, 0] + 1
output o2 := o1[-1, 0] + o1 + o1[1, 0]
assert<a1> 0 ≤ o2 and o2 ≤ 3
```

Listing 2: Lola specification with assumptions on a reset that guarantees that an output remains within bounds.

in these cases. The combination of past out-of-bounds and future out-of-bounds default values must also be covered by the obligations in case the stream is stopped early. These scenarios are depicted for the running example in Figure 2.

The figure shows four finite stream executions with different lengths. All stream positions are colored gray, while only some positions contain a single red dot. These features indicate the unfolding of stream variables and annotations using the template ϕ_t. A gray-colored position means that the assumptions have been unfolded and a dotted position means the assertion has been unfolded. Further, arrows indicate temporal stream accesses where solid lines correspond to accesses by outputs and dashed lines correspond to accesses by annotations, i.e., assumptions and assertions. For each stream execution, only the arrows for a single position are depicted – the arrows for other positions have been omitted for the sake of clarity. For example, for $N = 0$, the accesses of output $o2$ are both out-of-bounds, i.e., the default value zero is used. While for $N = 3$, the accesses at the second position are shown where only the past access of the assumption leads to an out-of-bounds access, i.e., only the dotted line towards the beginning of the stream execution. The figure depicts all necessary stream executions that cover all combinations of past out-of-bounds accesses, i.e., with and without future bound violations. The described unfoldings of Figure 2 are formalized as proof obligations in Definition 3.

Definition 3 (Proof Obligations ϕ_{Begin} for Past Out-of-bounds Accesses)
Let $w_p = \sup(\{0\} \cup \{|k| \mid e[k, c] \in \Phi \text{ where } k < 0\})$ be the most negative offset and $w_f = \sup(\{0\} \cup \{k \mid e[k, c] \in \Phi \text{ where } k > 0\})$ be the greatest positive offset. The proof obligations ϕ_{Begin} for past out-of-bounds accesses are defined as the conjunction of template formulas:

$$\bigwedge_{0 \le N < \max(1,\ 2\cdot(w_p+w_f))} \phi_t(^{p\_asm,\ p\_asserted,}_{p\_streams,\ p\_assert})$$

with template parameters:

- $p\_asm \quad := \{i \mid 0 \le i \le N\}$,
- $p\_asserted \quad := \emptyset$,
- $p\_streams \quad := \{i \mid 0 \le i \le N\}$,
- $p\_assert \quad := \{i \mid 0 \le i < \max(1,\ \min(N+1,\ 2 \cdot w_p))\}$.

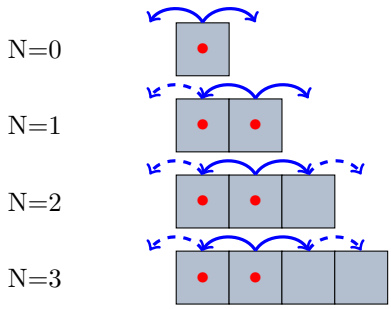


Fig. 2: Four stream executions of different length $N + 1$ with the respective template unfolding are depicted. The stream executions consider all cases with past out-of-bound accesses. A gray-colored box indicates that an assumption has been unfolded at this position, while a red dotted box indicates that an assertion has been unfolded at this position. Solid and dashed arrows indicate accesses by streams and annotations, respectively.

Next, the case where no out-of-bounds access occurs is considered. Hence, the obligations capture the nominal case where no default value is used. Since we have shown that past out-of-bounds accesses are valid we can use these proven assertions as assumptions. Figure 3 depicts a stream execution with a single dotted position, i.e., the position where the assertion must be proven. As can be seen, all accesses from this position are within bounds. Further, note that the accesses of the first and the last unfolded assumption, i.e., the first and the last gray-colored position, are also within bounds. The described unfolding is formalized as proof obligations in Definition 4.

Definition 4 (Proof Obligations ϕ_{Run} for No Out-of-bounds Accesses)
The proof obligations ϕ_{Run} without out-of-bounds accesses are defined as
$\phi_t(p\_asm,\ p\_asserted,\ p\_streams,\ p\_assert)$ with template parameters:

- $p\_asm \quad := \{i \mid w_p \le i \le N - w_f\}$,
- $p\_asserted \quad := \{i \mid 2 \cdot w_p \le i \le N - 2 \cdot w_f \wedge i \ne 3 \cdot w_p\}$,
- $p\_streams \quad := \{i \mid 2 \cdot w_p \le i \le N - 2 \cdot w_f\}$,
- $p\_assert \quad := \{i \mid i = 3 \cdot w_p\}$,

where $N = 3 \cdot (w_p + w_f)$.

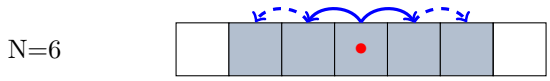


Fig. 3: A stream execution of length $N + 1$ with the corresponding template unfolding is depicted. The stream execution considers the case where no out-of-bound access occurs. Gray-colored and red dotted positions represent unfolded assumptions and assertions, respectively. Solid and dashed arrows indicate accesses by streams and annotations, respectively.

Last, we consider the case where only future out-of-bounds accesses occur. Hence, the respective obligations need to incorporate default values of future out-of-bounds accesses. As before, we can use the previously proven assertions as assumptions. Figure 4 depicts a stream execution with two dotted positions, i.e., positions where the assertion must be proven. The position where arrows are given represents the case where only the assumption results in a future out-of-bounds access. The last position of the stream execution represents the case in which both the assumption and the stream result in future out-of-bounds accesses. The presented unfolding is formalized as proof obligations in Definition 5.

Definition 5 (Proof Obligations ϕ_{End} for Future Out-of-bounds Accesses)
The proof obligations ϕ_{End} for future out-of-bounds accesses are defined as
$\phi_t(p\_asm,\ p\_asserted,\ p\_streams,\ p\_assert)$ with template parameters:

- $p\_asm \quad := \{i \mid w_p \leq i \leq N\}$,
- $p\_asserted \quad := \{i \mid 2 \cdot w_p \leq i < 3 \cdot w_p\}$,
- $p\_streams \quad := \{i \mid 2 \cdot w_p \leq i \leq N\}$,
- $p\_assert \quad := \{i \mid 3 \cdot w_p \leq i \leq N\}$

where $N = 3 \cdot w_p + w_f$.

So far, we have defined proof obligations for certain positions in the stream execution with and without out-of-bounds accesses. Together, the proof obligations constitute an inductive argument for the correctness of the assertions, see Proposition 2. Here, the base case is given by Definition 3 and induction steps are given by Definitions 4 and 5. The induction steps use the induction hypothesis, i.e., valid assertions, due to the template parameter $p\_asserted$.

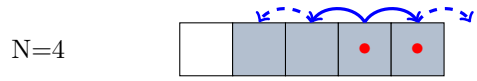


Fig. 4: A stream execution of length $N + 1$ with the corresponding template unfolding is depicted. The stream execution covers all cases where future out-of-accesses occur. Gray-colored and red dotted positions represent unfolded assumptions and assertions, respectively. Solid and dashed arrows indicate accesses by streams and annotations, respectively.

Proposition 2 (Assertion Verification by Lola Unfolding)
The set of assertions is correct if the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ *is valid.*

To prove that Proposition 2 holds, we distinguish exhaustively four specification cases: no temporal accesses (Proposition 3), past temporal accesses only (Proposition 4), future temporal accesses only (Proposition 5), and past and future temporal accesses (Proposition 6). First, in the case without temporal accesses, we show that all the necessary obligations for a Hoare triple are encoded in the formula and that this formula only evaluates to *false* if the assertions are not satisfied while all assumptions are.

Proposition 3 (Assertion Verification for Zero Offsets)
For a Lola specification with $w_p = 0$ *and* $w_f = 0$*, the set of assertions is correct if the formula* $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ *is valid.*

Proof The set of assertions $\psi_1, \ldots, \psi_n$ is correct for all input streams if and only if whenever an assumption is satisfied, its corresponding assertion is satisfied as well. Without loss of generality, let $\Gamma = \{\alpha\}$, $assume(\alpha) = \{\theta\}$, and $assert(\alpha) = \{\psi\}$. We prove the proposition by showing that the formula encodes an argument for the correctness of the assertions. We consider the case that $w_p = 0$, $w_f = 0$, and the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ is valid.

In this case, we do not have past or future out-of-bounds accesses. Hence, the obligations consider a single *Run* step with $N = 0$ and template parameters:

- $p\_asm \quad := \{i \mid i = 0\}$,
- $p\_asserted \quad := \emptyset$,
- $p\_streams \quad := \{i \mid i = 0\}$,
- $p\_assert \quad := \{i \mid i = 0\}$.

By instantiating the template for *Run*, we encode the following obligations:

$$
\begin{aligned}
\phi_{Run} &\overset{\text{Def. 3}}{=} \phi_t(\underset{p\_streams}{p\_asm}, \underset{p\_assert}{p\_asserted},) \\
&\overset{\text{Def. 2}}{=} \bigwedge_{i \in \{0\}} smt(\theta)(i) \ \wedge \ \bigwedge_{i \in \emptyset} smt(\psi)(i) \\
&\quad \wedge \ \bigwedge_{i \in \{0\}} \Big(\bigwedge_{0<k\leq n} \sigma_k = smt(e_k)(i) \Big) \\
&\quad \rightarrow \ \bigwedge_{i \in \{0\}} smt(\psi)(i) \\
&= \ smt(\theta)(0) \ \wedge \ \bigwedge_{0<k\leq n} \sigma_k = smt(e_k)(0) \\
&\quad \rightarrow \ smt(\psi)(0)
\end{aligned}
$$

As can be seen, the formula encodes "*assume* $\wedge$ *program* $\rightarrow$ *assertion*" for a single position. Note that a single position suffices since no temporal dependencies exist. Table 1 shows that the formula evaluates to $false$ only if an assertion evaluates to false, although the assumptions are valid and the output computations behave as expected. Conversely, if the formula is valid, then the set of assertions is correct or assumptions violated.
Note that no further obligations are added by ϕ_{Begin} and ϕ_{End} since the same template instances are created.

$$
\begin{aligned}
\phi_{Begin} &\overset{\text{Def. 3}}{=} \bigwedge_{0\leq N<1} \phi_t(\underset{\{i|0\leq i\leq N\},}{\{i|0\leq i\leq N\},} \underset{\{i|0\leq i<1\}}{\emptyset,}) \\
&= \ \phi_t(\{0\}, false, \{0\}, \{0\}) = \phi_{Run} \\
\phi_{End} &\overset{\text{Def. 5}}{=} \phi_t(\underset{\{i|0\leq i\leq 0\},}{\{i|0\leq i\leq 0\},} \underset{\{i|0\leq i\leq 0\}}{\emptyset,}) \\
&= \ \phi_t(\{0\}, false, \{0\}, \{0\}) = \phi_{Run}
\end{aligned}
$$

□

Next, we show that all necessary obligations for temporal accesses to previous stream values are encoded. Further, we show that an encoding of an inductive argument is provided that considers all possible combinations of out-of-bounds accesses at the beginning of stream execution as the base case and a monitoring step with no out-of-bounds accesses as the inductive step.

θ	σ	ψ	$\theta \wedge \sigma \rightarrow \psi$	
0	0	0	1	assumption invalid
0	0	1	1	assumption invalid
0	1	0	1	assumption invalid
0	1	1	1	assumption invalid
1	0	0	1	outputs invalid
1	0	1	1	outputs invalid
1	1	0	0	incorrect
1	**1**	**1**	**1**	correct

Table 1: Truth table for the encoding of the Hoare triple.

Proposition 4 (Assertion Verification for Past Offsets Only)
For a Lola *specification with* $w_p > 0$ *and* $w_f = 0$*, the set of assertions is correct if the formula* $\phi_{Begin} \ \wedge \ \phi_{Run} \ \wedge \ \phi_{End}$ *is valid.*

Example Consider the Lola specification

```
assume<a1> reset[-1, false] ∨ reset                          1
input reset : Bool                                           2
output o1 := if reset then 0 else o1[-1, 0] + 1              3
output o2 := o1[-1, 0] + o1                                  4
assert<a1> 0 ≤ o2 and o2 ≤ 3                                 5
```

that simplifies Listing 2, e.g., only past offset accesses are used. Here, w_p is 1 and w_f is 0. The unfolding of ϕ_{Begin} checks all possible combinations of out-of-bounds accesses of annotations, i.e., `reset[-1, false]`, and outputs, i.e., `o1[-1, 0]`. In comparison to Figure 2, ϕ_{Begin} would only produce $N = 0$ and $N = 1$ without future accesses. ϕ_{Run} represents the induction step where no default values are taken.

Proof The set of assertions $\psi_1, \ldots, \psi_n$ is correct for all input streams if and only if whenever an assumption is satisfied, its corresponding assertion is satisfied as well. Without loss of generality, let $\Gamma = \{\alpha\}$, $assume(\alpha) = \{\theta\}$, and $assert(\alpha) = \{\psi\}$. We prove the proposition by showing that the formula encodes an inductive argument for the correctness of the assertions. We consider the case that $w_p > 0$, $w_f = 0$, and the formula $\phi_{Begin} \ \wedge \ \phi_{Run} \ \wedge \ \phi_{End}$ is valid.

In this case, we consider only accesses to the past. The formula encodes a k-induction where ϕ_{Begin} encodes the base cases and ϕ_{Run} the step case.
The template parameter for ϕ_{Begin} are:

- p_asm $:= \{i \mid 0 \leq i \leq N\}$,
- $p\_asserted$ $:= \emptyset$,
- $p\_streams$ $:= \{i \mid 0 \leq i \leq N\}$,
- p_assert $:= \{i \mid 0 \leq i < \min(N+1, 2 \cdot w_p)\}$.

$$\phi_{Begin} \overset{\text{Def. 3}}{=} \bigwedge_{0 \leq N < 2 \cdot w_p} \phi_t(\substack{p\_asm, p\_asserted, \\ p\_streams, p\_assert})$$

$$\overset{\text{Def. 2}}{=} \bigwedge_{0 \leq N < 2 \cdot w_p} \Big(\bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0 < k \leq n} \sigma_k = smt(e_k)(i) \Big) \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i) \Big)$$

ϕ_{Begin} encodes stream executions that handles all possible out-of-bounds accesses. The range of stream execution N from 0 to $2 \cdot w_p$ covers all combinations of stream and assumption out-of-bounds scenarios. Two times the w_p is required since it is required in case that an output accesses available past values while at the same position the access of an assumption is out-of-bounds. Figure 5 depicts an example for $w_p = 1$. The sub-figure on the top shows a stream execution where both the output computation and the assumption have out-ouf-bounds accesses. The sub-figure on the bottom shows a stream execution where the output access to the past at the last position is in-bound but at the accessed position the access of the used assumption is not.

ϕ_{Run} unfolds a stream execution and uses the inductive argument. The unfolding is depicted in Fig. 6. It shows that a stream execution is unfolded with an assertion at position $3 \cdot w_p$. The inductive argument is depicted as *asserted*. The *asserted* unfolding is based on the base-cases of the k-induction where $k = w_p$. Further, our encoding ensures that no out-of-bounds accesses occur by sufficiently unfolding the *assumptions* and *outputs*.

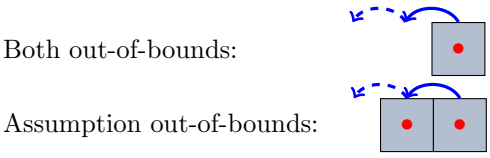


Fig. 5: All stream executions of ϕ_{Begin} for $w_p = 1$ are depicted. It shows the case where possibly both accesses are out-of-bounds (upper) and the case where only an assumption is out-of-bounds (lower). A gray-colored box indicates that an assumption has been unfolded at this position, while a red dotted box indicates that an assertion has been unfolded at this position. Solid and dashed arrows indicate accesses by streams and annotations, respectively.

The template parameter for ϕ_{Run} are:

- p_asm $:= \{i \mid w_p \leq i \leq 3 \cdot w_p\}$,
- $p\_asserted$ $:= \{i \mid 2 \cdot w_p \leq i < 3 \cdot w_p\}$,
- $p\_streams$ $:= \{i \mid 2 \cdot w_p \leq i \leq 3 \cdot w_p\}$,
- p_assert $:= \{i \mid i = 3 \cdot w_p\}$.

and $N = 3 \cdot w_p$.

$$\phi_{Run} \overset{\text{Def. 4}}{=} \phi_t(\substack{p\_asm, p\_asserted, \\ p\_streams, p\_assert})$$

$$\overset{\text{Def. 2}}{=} \bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in p\_asserted} smt(\psi)(i) \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0 < k \leq n} \sigma_k = smt(e_k)(i) \Big) \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i)$$

Since $w_f = 0$, no future out-of-bounds access can occur and, therefore, no obligations are added by ϕ_{End}.

$$\phi_{End} \overset{\text{Def. 5}}{=} \phi_t(\substack{\{i \mid w_p \leq i \leq 3 \cdot w_p\}, \ \{i \mid 2 \cdot w_p \leq i < 3 \cdot w_p\}, \\ \{i \mid 2 \cdot w_p \leq i \leq 3 \cdot w_p\}, \ \{i \mid 3 \cdot w_p \leq i \leq 3 \cdot w_p\}})$$

$$= \phi_t(p\_asm, p\_asserted, p\_streams, p\_assert)$$

$$= \phi_{Run}$$

The formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ intuitively encodes a k-induction where $k = w_p$:

$$\phi_{Begin} \begin{cases} assume(0) \wedge program(0) \rightarrow assert(0) \\ \dots \\ assume(k) \wedge program(k) \rightarrow assert(k) \end{cases}$$

$$\wedge \phi_{Run} \begin{cases} assume(n) \wedge program(n) \rightarrow assert(n) \end{cases}$$

Similar to the first case, for each base case and step case, if the formula is valid then the assertions must be correct. □

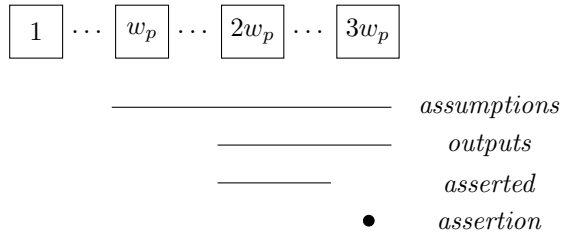


Fig. 6: The unfolding of a stream execution is depicted. The assertion is proven at Position $3 \cdot w_p$. *Assumptions* and *outputs* are unfolded such that all accesses can be resolved. The inductive argument that is shown in the induction base is represented by *asserted*.

After specifications with past offsets only, we now consider future-only specifications. Similar to before, we show that an encoding of an inductive argument is provided. In contrast to before, the encoding must also prove that default values do not violate assertions at the end of a stopped stream execution.

Proposition 5 (Assertion Verification for Future Offsets Only)
For a Lola *specification with* $w_p = 0$ *and* $w_f > 0$*, the set of assertions is correct if the formula* $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ *is valid.*

Example Consider the Lola specification

```
assume<a1> reset ∨ reset[1, false]
input reset : Bool
output o1 := if reset then 0 else o1[1, 0] + 1
output o2 := o1[1, 0] + o1
assert<a1> 0 ≤ o2 and o2 ≤ 3
```

that simplifies Listing 2, e.g., only future offset accesses are used. Here, w_p is 0 and w_f is 1. The unfolding of ϕ_{Begin} checks all possible combinations of out-of-bounds accesses of annotations, i.e., `reset[1, false]`, and outputs, i.e., `o1[1, 0]`. ϕ_{Run} represents the induction step where no default values are taken.

Proof The set of assertions $\psi_1, \ldots, \psi_n$ is correct for all input streams if and only if whenever an assumption is satisfied, its corresponding assertion is satisfied as well. Without loss of generality, let $\Gamma = \{\alpha\}$, $assume(\alpha) = \{\theta\}$, and $assert(\alpha) = \{\psi\}$. We prove the proposition by showing that the formula encodes an inductive argument for the correctness of the assertions. We consider the case that $w_p = 0$, $w_f > 0$, and the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ is valid.

In this case, only future stream accesses are considered. The formula encodes a k-induction where ϕ_{End} represent the base cases and ϕ_{Run} the step case. By unfolding a stream execution of length w_f, ϕ_{End} covers all possible future out-of-bounds combinations for stream as well as annotation accesses. Figure 7 depicts ϕ_{End} where $N = w_f$. The cases are similar to the base cases in the second case but this time for the future accesses. The template parameter for ϕ_{End} are:

- p_asm $:= \{i \mid 0 \leq i \leq w_f\}$,
- $p\_asserted$ $:= \emptyset$,
- $p\_streams$ $:= \{i \mid 0 \leq i \leq w_f\}$,
- p_assert $:= \{i \mid 0 \leq i \leq w_f\}$.

and $N = w_f$.

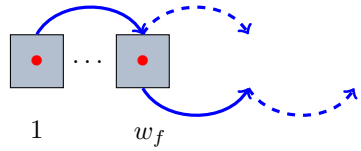


Fig. 7: A stream executions of length w_f is depicted. It shows the case where only the last assumption access is out-of-bounds. A gray-colored box indicates that an assumption has been unfolded at this position, while a red dotted box indicates that an assertion has been unfolded at this position. Solid and dashed arrows indicate accesses by streams and annotations, respectively.

$$\phi_{End} \overset{\text{Def. 5}}{=} \phi_t(p\_asm, p\_asserted, p\_streams, p\_assert)$$
$$\overset{\text{Def. 2}}{=} \bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in \emptyset} smt(\psi)(i) \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0 < k \leq n} \sigma_k = smt(e_k)(i) \Big) \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i)$$

The induction step ϕ_{Run} unfolds a stream execution and uses the inductive argument. The unfolding is depicted in Fig. 8. In this case, the *assertion* is at the first position and the k base cases hold for possible future out-of-bounds accesses, i.e., at the next w_f positions. Further, *outputs* and *assumptions* are unfolded to consider all necessary *output* computations and *assertion* accesses.
The template parameter for ϕ_{Run} are:

- p_asm $:= \{i \mid 0 \leq i \leq 2 \cdot w_f\}$,
- $p\_asserted$ $:= \{i \mid 0 \leq i \leq w_f \wedge i \neq 0\}$,
- $p\_streams$ $:= \{i \mid 0 \leq i \leq w_f\}$,
- p_assert $:= \{i \mid i = 0\}$.

and $N = 3 \cdot w_f$.

$$\phi_{Run} \overset{\text{Def. 4}}{=} \phi_t(p\_asm, p\_asserted, p\_streams, p\_assert)$$
$$\overset{\text{Def. 2}}{=} \bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in p\_asserted} smt(\psi)(i) \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0 < k \leq n} \sigma_k = smt(e_k)(i) \Big) \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i)$$

Note that ϕ_{Begin} adds no further obligations to the formula since a single execution suffices to include all possible out-of-bounds values.

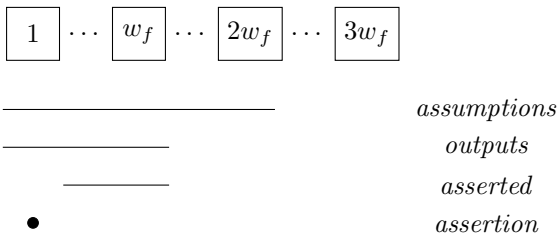


Fig. 8: The unfolding of a stream execution is depicted. The assertion is proven at the first position. *Assumptions* and *outputs* are unfolded such that all accesses can be resolved. The inductive argument that is shown in the induction base is represented by *asserted*.

The formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ intuitively encodes a k-induction where $k = w_f$:

$$\phi_{End} \begin{cases} assume(0) \wedge program(0) \rightarrow assert(0) \\ \dots \\ assume(k) \wedge program(k) \rightarrow assert(k) \end{cases}$$
$$\wedge\, \phi_{Run} \begin{cases} assume(n) \wedge program(n) \rightarrow assert(n) \end{cases}$$

Similar to the past-only case, for each base case and step case, if the formula is valid then the assertions must be correct. □

Finally, the next case provides an inductive argument for Lola specification with past and future temporal accesses. The inductive argument incorporates default values in the base case at the beginning and at the end of stream executions. Further, it handles all combinations of past and future out-of-bounds accesses.

Proposition 6 (Assertion Verification for Non-Zero Offsets)
For a Lola *specification with* $w_p > 0$ *and* $w_f > 0$*, the set of assertions is correct if the formula* $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ *is valid.*

Example Consider the Lola specification in Listing 2, i.e., $w_p = 1$ and $w_f = 1$. The unfolding of ϕ_{Begin} checks all possible combinations of out-of-bounds accesses of annotations, i.e., `reset[-1, false]` and `reset[1, false]` , and outputs, i.e., `o1[-1, 0]` and `o1[1, 0]`. ϕ_{Run} represents the induction step where no default values are taken.

Proof The set of assertions $\psi_1, \dots, \psi_n$ is correct for all input streams if and only if whenever an assumption is satisfied, its corresponding assertion is satisfied as well. Without loss of generality, let $\Gamma = \{\alpha\}$, $assume(\alpha) = \{\theta\}$, and $assert(\alpha) = \{\psi\}$. We prove the proposition by showing that the formula encodes an inductive argument for the correctness of the assertions. We consider the case that $w_p > 0$, $w_f > 0$, and the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ is valid.

Further, we consider combinations of past and future accesses. Hence, all combinations of out-of-bounds accesses need to be covered. The formula ϕ_{Begin} covers all possible combinations of past out-of-bounds accesses with future accesses. The combinations are depicted in Fig. 9: $N = 0$ represents the case that all accesses are out-of-bounds, $N = w_p$ represents the case that except of the past output stream access at the last position all other accesses are out-of-bounds and vice versa for the future stream access at the first position, $N = 2 \cdot w_p$ represents the case that only accesses of annotations are out-of-bounds, $N = 2 \cdot w_p + w_f$ represents both cases that only one annotation is out-of-bounds, and $N = 2 \cdot (w_p + w_f) - 1$ represents the case that only the past access of an annotation is out-of-bounds.
The template parameter for ϕ_{Begin} are:

- p_asm $:= \{i \mid 0 \leq i \leq N\}$,
- $p\_asserted$ $:= \emptyset$,
- $p\_streams$ $:= \{i \mid 0 \leq i \leq N\}$,
- p_assert $:= \{i \mid 0 \leq i < \min(N + 1, 2 \cdot w_p)\}$.

$$\phi_{Begin} \overset{\text{Def. 3}}{=} \bigwedge_{0 \leq N < 2 \cdot (w_p + w_f)} \phi_t(\substack{p\_asm, p\_asserted \\ p\_streams, p\_assert})$$
$$\overset{\text{Def. 2}}{=} \bigwedge_{0 \leq N < 2 \cdot (w_p + w_f)} \Big(\bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in \emptyset} smt(\psi)(i) \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0 < k \leq n} \sigma_k = smt(e_k)(i) \Big) \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i) \Big)$$

Since ϕ_{Begin} covers only combinations of past out-of-bounds accesses with future accesses, ϕ_{End} covers future out-of-bounds accesses only. The unfolding is depicted in Fig. 10. The *assertions* are unfolded such that all combinations of future out-of-bounds accesses are covered: annotation only and stream and annotation combined. Since the first k base cases are already covered, they can be used here as *asserted*. Further, *outputs* and *assumptions* are unfolded such that all required accesses are available.

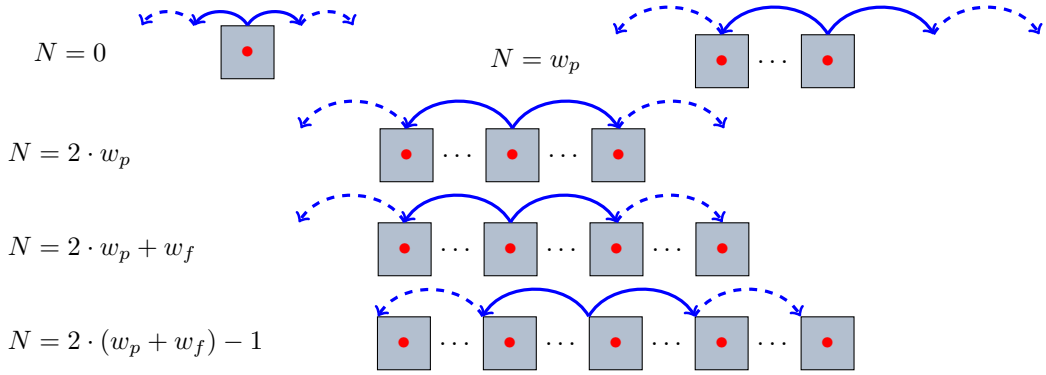


Fig. 9: All combinations of past out-of-bounds accesses with future accesses are depicted.

The template parameter for ϕ_{End} are:

- $p\_asm := \{i \mid w_p \leq i \leq 3 \cdot w_p + w_f\}$,
- $p\_asserted := \{i \mid 2 \cdot w_p \leq i < 3 \cdot w_p\}$,
- $p\_streams := \{i \mid 2 \cdot w_p \leq i \leq 3 \cdot w_p + w_f\}$,
- $p\_assert := \{i \mid 3 \cdot w_p \leq i \leq 3 \cdot w_p + w_f\}$.

and $N = 3 \cdot w_p + w_f$.

$$\begin{aligned}\phi_{End} &\overset{\text{Def. 4}}{=} \phi_t(p\_asm,\ p\_asserted, p\_streams, p\_assert)\\ &\overset{\text{Def. 2}}{=} \bigwedge_{i \in p\_asm} smt(\theta)(i) \ \wedge \bigwedge_{i \in p\_asserted} smt(\psi)(i)\\ &\quad \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0<k\leq n} \sigma_k = smt(e_k)(i) \Big)\\ &\quad \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i)\end{aligned}$$

Last, we need to show that the assertions also hold for the case that no out-of-bounds access exists. Hence, the unfolding is encoded as depicted in Fig. 11. The *assertion* is proven at position $3 \cdot w_p$ and already shown assertions of past and future accesses are incorporated by *asserted*. Further, all required accesses of *outputs* and *assumptions* are unfolded such that no out-of-bounds access exists.

The template parameter for ϕ_{Run} are:

- $p\_asm := \{i \mid w_p \leq i \leq 3 \cdot w_p + 2 \cdot w_f\}$,
- $p\_asserted := \{i \mid 2 \cdot w_p \leq i \leq 3 \cdot w_p + w_f \wedge i \neq 3 \cdot w_p\}$,
- $p\_streams := \{i \mid 2 \cdot w_p \leq i \leq 3 \cdot w_p + w_f\}$,
- $p\_assert := \{i \mid i = 3 \cdot w_p\}$.

and $N = 3 \cdot (w_p + w_f)$.

$$\begin{aligned}\phi_{Run} &\overset{\text{Def. 4}}{=} \phi_t(p\_asm,\ p\_asserted, p\_streams, p\_assert)\\ &\overset{\text{Def. 2}}{=} \bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in p\_asserted} smt(\psi)(i)\\ &\quad \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0<k\leq n} \sigma_k = smt(e_k)(i) \Big)\\ &\quad \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i)\end{aligned}$$

Similar to the other cases, for each base case and step case, if the formula is valid then the assertions must be correct.

□

By proving all cases of temporal accesses in the Propositions 3 to 6, the Proposition 2 is proven – the verification approach is sound. Soundness refers to the ability of an analyzer to prove the absence of errors — if a Lola specification is accepted, it is guaranteed that the assertions are not violated. The converse does not hold, i.e., the presented verification approach is not complete. Completeness refers to the ability of an analyzer to prove the presence of errors — if a Lola specification is rejected, the counter-example given should be a valid stream execution that results in an assertion violation. The following Lola specification is rejected even though no assertion is violated:

```
input a: Int32                                              1
assume <a1> a ≤ 10                                          2
output sum :=                                               3
      if sum[-1, 0] ≤ 10 then 0 else sum[-1, 0] + a         4
assert <a1> sum ≤ 100                                       5
```

Here, since the `if`-condition in Line 3 evaluates to *true* at the beginning of the stream execution, `sum` is a constant stream with value zero. Hence, the assertion in Line 4 is never violated. The verification approach rejects this specification. The reason for this is that `sum` ≤ 100 is added as an *asserted* condition in ϕ_{Run}. Therefore, the SMT solver can assign a value between 91 and 100 to the earliest `sum` variable of the unfolding, resulting in an assertion violation of the next `sum` variable.

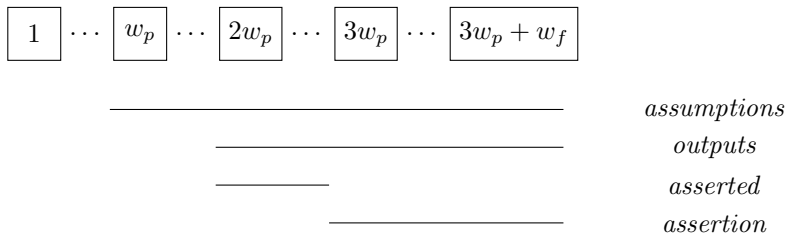


Fig. 10: The unfolding at the end of a stream execution is depicted. *Assumptions* and *outputs* are unfolded such that all accesses can be resolved. The inductive argument that is shown in the induction base is represented by *asserted*.

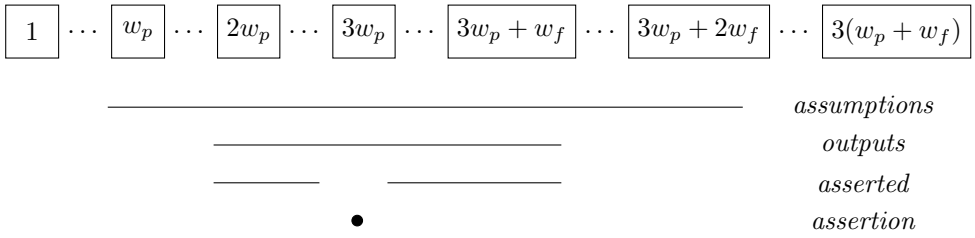


Fig. 11: The unfolding of a stream execution during run is depicted. The assertion is proven at position $3 \cdot w_p$. *Assumptions* and *outputs* are unfolded such that all accesses can be resolved. The inductive argument that is shown in the induction base is represented by *asserted*.

4 Application Experience in Avionics

In this section, we present details about the tool implementation and tool experiences on practical avionic specifications.

Tool Implementation and Usage

The tool is based on the open source RTLola framework[4]. Specifically, it uses the Lola frontend to parse a given specification into an intermediate representation. Based on this representation, the SMT formulas are created and evaluated with the Rust z3 crate[5]. At its current phase of the crate's development, a combined solver is implemented that internally uses either a non-incremental or an incremental solver. There is no information on the implemented tactics available, but all our requests could be solved within seconds. For functions that are not natively supported by the Rust Z3 solver, the output is arbitrarily chosen by the solver with respect to the range of the function. The tool expects a Lola specification augmented by *assumptions* and *assertions*. The verification is done automatically and produces a counter-example stream execution, if any exists. The counter-example can then be used by the user to debug its specifications. Two different kinds of users are targeted. First, users that write the entire augmented specification. Such a user could be a system engineer who is developing a safety monitor and wants to ensure that it contains critical properties. Second, users that augment an existing specification. Here, one reason could be that an existing monitor shall be composed with other critical components and certain behavioral properties are expected. Also, similar to software testing, the task of writing a specification and their respective assumptions and assertions could be separated between two users to ensure the independence of both.

[4] https://rtlola.org/. The extension is not open source yet, but will be integrated into [23].

[5] https://docs.rs/z3/0.9.0/z3/

Practical Results

To gain practical tool experience, previously written specifications based on interviews with engineers of the German Aerospace Center [25] were extended by assumptions and assertions. The previous specifications were tested using log-files and simulations – the authors considered them correct. We report several specification errors in Table 2 that were detected by the presented verification extension. In fact, the detected errors would have resulted in undetected failures. After the errors in the previous specifications were fixed, all assertions were proven correct. Note that the errors could have been found by manual reviews. However, such reviews are tedious and error-prone, especially when temporal behaviors are involved. The detected errors in Table 2 can be grouped into three classes: *Classical Bugs*, *Operator Errors*, and *Wrong Interpretations*. Classical bugs are errors that occur when implementing an algorithm. Operator errors are Lola specific errors, e.g., temporal accesses. Last, wrong interpretations refer to gaps between the specification and the user's design intend, e.g., violated assertions due to incomplete specifications. Next, we give one representative example for each group. We reduced the specification to the representative fragment.

Example 1 (Classical Bug)
The Lola specification in Listing 3 monitors the fuel level. A monitor shall notify the operator when one of the three different fuel levels are reached: half (Line 9), warning (Line 10), and danger (Line 11). The fuel level is computed as a percentage in Lines 7 to 8. It uses the fuel level at the beginning of the flight (Line 6) as a reference for its computation. Given the documentation of the fuel sensor, it is known that `fuel` values are within $\mathbb{R}^+$ and decreasing. This is formalized in Line 4 as an assumption. As an invariant, we asserted that the starting fuel is greater or equal to `fuel` (Line 16). Further, in Lines 17 to 19, we stated that once a level is reached it should remain at this level. During our experiment, the assertion led to a counter-example that pointed to the previously used and erroneous fuel level computation:

```
output fuel_level := (start_fuel - fuel) / start_fuel
```

In short, the output computed the consumed fuel and not the remaining fuel. The computation could be easily fixed by converting consumed fuel into remaining

```
// Inputs                                                          1
input fuel: Float64                                                2
// Assumptions                                                     3
assume<a5> fuel > 0.0 and fuel < fuel[-1, fuel + 0.1]              4
// Outputs                                                         5
output start_fuel := start_fuel[-1, fuel]                          6
output fuel_level :=                                               7
          1.0 - (start_fuel - fuel) / start_fuel                   8
output fuel_half := fuel_level < 0.50                              9
output fuel_warning := fuel_level < 0.25                           10
output fuel_danger := fuel_level < 0.10                            11
trigger_once fuel_half "INFO: Fuel is below 50%"                   12
trigger_once fuel_warning "WARNING: Fuel is below 25%"             13
trigger_once fuel_danger "DANGER: Fuel is below 10%"               14
// Assertions                                                      15
assert<a5> start_fuel >= fuel                                      16
         and (fuel_half[-1, false] -> fuel_half)                   17
         and (fuel_warning[-1, false] -> fuel_warning)             18
         and (fuel_danger[-1, false] -> fuel_danger)               19
```

Listing 3: The fixed version of the Lola ctrl_output specification that monitors the fuel level. Three level of engagement are depicted: half, warning, and danger.

fuel, see Line 8. Therefore, Listing 3 satisfies its assertion. Note, that offset accesses were used to assert the temporal behavior of the fuel level output stream. Further, `trigger_once` is an abbreviation which states that only the first raising edge is reported to the user.

Example 2 (Operator Error)
An important monitoring property is to detect frozen values as these indicate a deteriorated sensor. Such a specification is depicted in Listing 4. Here, as an input, the acceleration in $x-$direction is given. The frozen value check is computed from Line 6 to Line 10. It compares previous values using Lola's offset operator. To check this computation, we added the sanity check that asserts that no frozen value shall be detected (Line 13) when small changes in the input are present (Line 4). In the previous version, the frozen values were computed using the abbreviated offset operator:

```
output frozen_ax := ax[-5..0, 0.0, =]
```

This resulted in a counter-example that pointed to wrong default values. Although the abbreviated version is easier to read and reduces the size of the specification, it is unfortunately not suitable for this kind of property. The tool detected the unlikely situation that the first value of `ax` is 0.0 which would have resulted in evaluating `frozen_ax` to true. Although unlikely, this should be avoided as contingencies activated in such situations depend on correct results and otherwise could harm people on the ground. By unfolding the operator and adding a different default value to one of the past accesses, the error was resolved (Line 6). Listing 4 shows the fixed version which satisfies its assertion.

Specification	Appx.	#o	#a	#g	Detected errors
gps_vel_output	A.1	14	6	6	–
gps_pos_output	A.2	19	3	10	–
imu_output	A.3	18	6	6	Wrong default value Division by zero
nav_output	A.4	25	3	5	Missing abs()
tagging	A.5	6	2	2	–
ctrl_output	A.6	25	7	8	Wrong threshold comparisons
mm_output_1	A.7	4	1	2	–
mm_output_2	A.8	17	6	9	Missing if condition Wrong default value
contingency_output	A.9	4	8	1	Observation: both contingencies could be true in case of voting, i.e., both at 50%
health_output	A.10	1	5	1	–

Table 2: Detected errors by the verification extension, where #o, #a, and #g represent the number of outputs, assumptions, and assertions given in the specification, respectively.

```
// Inputs
input ax: Float32
// Assumptions
assume <a1> ax != ax[-1, ax + ε]
// Outputs
output frozen_ax := ax[-5, 0.1] = ax[-4, 0.0]
                and ax[-4, 0.0] = ax[-3, 0.0]
                and ax[-3, 0.0] = ax[-2, 0.0]
                and ax[-2, 0.0] = ax[-1, 0.0]
                and ax[-1, 0.0] = ax
trigger frozen_ax "WARNING: x-acceleration is frozen!"
// Assertions
assert <a1> !frozen_ax
```

Listing 4: The Lola imu_output specification that monitors frozen acceleration values.

Example 3 (Wrong Interpretation)
In Listing 5, two visual sensor readings are received (Lines 2-5). Both, readings argue over the same observations where `avgDist` represents the average distance to the measured obstacle, `actual` is the number of measurements, and `static` is the number of unchanged measurements. A simple rating function is introduced (Lines 7-12) that estimates the corresponding rating – the higher the better. Using these ratings, the trust in each of the sensors is computed probabilistically (Lines 11-13). When considering the integration of such a monitor as an ASTM F3269-21 switch condition that decides which sensor value should be forwarded, the specification should be revised. This is the case because the assertion in Line 17 produces a counter-example which indicates that both trust triggers (Lines 14-15) can be activated at the same time. A common solution for this problem is to introduce a priority between the sensors.

```
// Inputs
input avgDist_laser, actual_laser,
      static_laser: Float64
input avgDist_optical, actual_optical,
      static_optical: Float64
// Outputs
output rating_laser := 0.2 * static_laser
      + 0.4 * actual_laser + 0.4 * avgDist_laser
output rating_optical := 0.2 * static_optical
      + 0.4 * actual_optical + 0.4 * avgDist_optical
output trust_laser :=
      rating_laser / ( rating_laser + rating_optical)
output trust_optical := 1.0 - trust_laser
trigger trust_laser >= 0.5
trigger trust_optical >= 0.5
// Assertions
assert <a1> trust_laser != trust_optical
```

Listing 5: The Lola contingency_output specification that uses an heuristic to decide which sensor is more trustworthy.

The examples show how the presented Lola verification extension can be used to find errors in specifications. We also noticed that the annotations can serve as documentation. System assumptions are often implicitly known during development and are finally documented in natural language in separate files. Having these assumptions explicitly stated within the monitor specification potentially reduces future mistakes when reusing the specification, e.g., when composing with other monitor specifications. Listing 6 depicts such an example specification. Here, the monitor interfaces are clearly defined by the domain of input a (Line 5) and output o (Line 13). Also, *reset* is assumed

to be valid at least once per second (Line 5). Further, no deeper understanding of the internal computations (Lines 7-10) is required in order to safely compose this specification with others.

```
1  // Inputs with frequency 5Hz
2  input a: Float64
3  input reset: Bool
4  // Assumptions
5  assume <a1> 0.0 ≤ a ≤ 1.0 and reset[-4..0, false, ∨]
6  // Outputs
7  output o_1 := ...
8  ...
9  output o_n := ...
10 output o := o_1 + ... + o_n
11 trigger o ≥ 0.5 "Warning: Output o exceeds threshold!"
12 // Assertions
13 assert <a1> 0.0 ≤ o ≤ 1.0
```

Listing 6: Lola specification annotations describe interface properties.

5 Monitoring Assumptions

The presented verification approach offers an analysis of the specification that can guarantee the desired behavior of the monitor outputs. Yet, these guarantees are often based on assumptions that can be violated at runtime. Further, a violated assumption does not directly result in a violated assertions. As an example consider Listing 4 that states an assertion which is violated when six consecutive `ax` values are the same. Hence, the sequence $\langle 0.1;\ 0.1;\ 0.1;\ 0.1;\ 0.4;\ 0.5\rangle$ satisfies the assertion. Yet, it violates the assumption at the first positions.

In this section, we will give a translation of an annotated specification into a specification that checks assumptions at runtime and efficiently activates and deactivates assertion checks. Further, we will present experimental results showing that the verification extension not only provides static guarantees, but that translating it into a corresponding specification can lead to better runtime performance compared to a monitor that simply checks all assertions during runtime.

5.1 Translation into Lola 2.0

We translate an annotated Lola specification into a Lola 2.0 specification [10]. Lola 2.0 extends Lola by streams that can be spawned, filtered, and closed at runtime. For the translation, we only need to handle assumptions and assertions since trigger, input, and output streams are directly supported by Lola 2.0.

We replace each assumption by an output stream. A stream `output` o `: Bool spawn if` e_s `filter` e_f `close` e_c `:= e(`t_1`, ..., `t_m`, `s_1`, ..., `s_n`)` is an output stream that is created when its spawn condition e_s is true. It then starts producing values by evaluating its computation e if its filter condition e_f is *true* until its close condition e_c is satisfied. Since a violated assumption can influence previous and future assertions due to temporal offset accesses, the computation e is a counter that represents how many assertions are impacted by the assumption. If the counter is positive, then an assumption was violated that influences an assertion computation. To compute the impact of a violated assumption, we take the maximum between the longest chain of offset accesses from assertion to assumption plus one. This is achieved by an analysis of the dependency graph [8]. To start the counter, we use the negated assumption as spawn condition. Further, we close the stream when the impact of a violated assumption is over, i.e., when the counter is zero. As filter condition, we use *true* to decrease the counter in each step.

We also represent assertion checks by output streams. For each assertion, we use an output stream that is only extended if one of its corresponding assumptions is violated, i.e., its counter value is positive. We also add a trigger to report assertion violations.

As an example consider the annotated Lola specification

```
1 input vel : Float32 // Velocity
2 assume <a> -20.0 ≤ vel ≤ 20.0
3 output vel_max := max(abs(vel), vel_max[-1, 0.0])
4 trigger vel_max > 20.0 "Velocity threshold exceeded!"
5 assert <a> abs(vel[-2..0, 0.0, +]) / 3.0 ≤ 20.0
```

that checks the maximal velocity value (Line 3) and the average velocity over a discrete window of three (Line 5). The assumption (Line 2) and the assertion (Line 5) are transformed to

```
1  output assumption
2      spawn if !(-20.0 ≤ vel ≤ 20.0)
3      filter true close assumption = 0
4  := if -20.0 ≤ vel ≤ 20.0
5       then assumption[-1,0] - 1 else 1
6  trigger assumption > 0 "Assumption violated!"
7  output assertion
8      spawn if true filter assumption > 0 close false
9  := abs(vel[-2..0, 0.0, +]) / 3.0 ≤ 20.0
10 trigger !assertion "Assertion violated!"
```

The output *vel_max* and trigger remain unchanged. Note that the trigger could have been replaced by an assertion. Yet, this would not

reduce as much overhead as for the window check which the following experiments will show.

5.2 Experiments

For our experiments, we compare the performance of the presented translation to a naive translation that checks assumptions and assertions independently in each execution step. As annotated specification, we use

```
input a_i: Float64
assume <a> a_i ≤ 2.0
assert <a> a_i[-w..0, 0.0, +] ≤ w · 2.0
```

that we scale in the number of annotation pairs using the variable i and the computational load of the assertion by the window variable w. For instance, $i = 10$ and $w = 5$ produces ten inputs with the corresponding annotations where each assertion takes the sum over the last five input values including the current one. The naive translation v_n with omitted triggers is

```
input a_i: Float64
output assumption_i := a_i ≤ 2.0
output assertion_i := a_i[-w..0, 0.0, +] ≤ w · 2.0
```

The presented translation v_t also with omitted triggers is

```
input a_i: Float64
output assumption_i
    spawn if a_i > 2.0 filter true close assumption_i = 0
:= if a_i ≤ 2.0 then assumption_i[-1,0] - 1 else w
output assertion_i
    spawn if true filter assumption_i > 0 close false
:= a_i[-w..0, 0.0, +] ≤ w · 2.0
```

For the experiments, the considered values of i were 5, 10, and 15 and the values of w were 0, 5, and 10. The experiments were conducted on three different kinds of log-files: no assumption is violated, all assumptions are violated, half of the assumptions are violated. Each log-file contains 10.000.000 events that were sufficient to report the average time in nanoseconds required by the monitor to evaluate one input event. Each experiment was carried out three times and the average was taken. For the experiments, an eight-core machine with an 2.5GHz Intel i7 processor with 32GB RAM was used.

The results of the experiments are depicted in Figure 12. As can be seen in Table 12a, which considers log-files with no violation of assumptions, version v_t significantly improves runtime by up to 64.06%. It can also be seen that version v_t improves the required time per event by 8.17% already in the case of simple assertions. Further, the required time for v_t remained constant while increasing the window size which shows that no unnecessary assertion checks were computed; in contrast to v_n, where the required time correlates with the size of the window. Next, Table 12b considers log-files where all assumptions are violated. The results show that this time v_t correlates with the size of the window similar to v_n since all the assertions need to be checked due to violated assumptions. The experiments show that v_t incurs an overhead of up to −33.86%. However Table 12c shows that already in the case where half of the inputs violate the assumptions and a more complex assertion is used ($w = 5$), the Lola 2.0 specification version v_t pays off and outperforms v_n by up to 15.91%. The results are also graphically depicted in Figure 12d.

Overall, the experiments show that translating an annotated specification into a Lola 2.0 specification can be used to report assertion violations due to violated assumptions efficiently at runtime. Since assumptions are generally expected to be satisfied in the nominal case, the translation also improves the monitor's runtime without losing its guarantees. Especially complex assertions based on simple assumptions benefit from the translation. If the assertions are simple, the benefits from the translation are negligible.

Remark: We also considered the alternative Lola assumption encoding

```
output assumption_i
:= if a_i ≤ 2.0 then if assumption_i[-1,0] = 0 then 0
     else assumption_i[-1,0] - 1 else w
```

that no longer uses Lola 2.0 features. Our result showed a runtime improvement of up to 59.17% in the case of no violations and a runtime deterioration of only up to −8.54%. Still, we decided on the Lola 2.0 assumption encoding to gain the best performance, since assumptions should not be violated in the nominal case. Yet, these results indicate that the parameterization of the assumptions has the largest share in the reported deterioration.

Window w	Number of inputs i								
	5			10			15		
	$\mathbf{v_n}$ $[\mu s]$	$\mathbf{v_t}$ $[\mu s]$	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ $[\mu s]$	$\mathbf{v_t}$ $[\mu s]$	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ $[\mu s]$	$\mathbf{v_t}$ $[\mu s]$	$\mathbf{\Delta}$ [%]
0	1203.67	1105.34	8.17	2404.67	2096.67	12.81	3674.67	3236.34	11.93
5	1951.67	1118.00	42.72	4146.34	2127.67	48.69	6318.00	3273.34	48.19
10	2859.67	1124.34	60.68	5985.00	2151.00	64.06	8977.00	3228.00	64.04

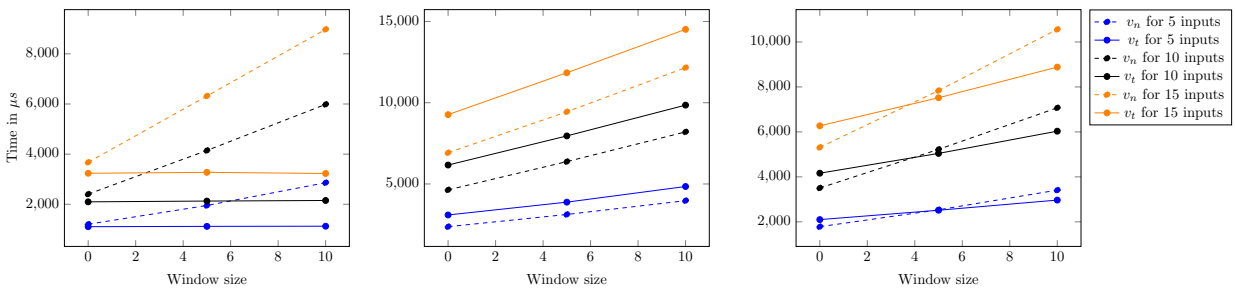
(a) None of the 10.000.000 events in the log-file violates the assumption.

Window w	Number of inputs i								
	5			10			15		
	$\mathbf{v_n}$ $[\mu s]$	$\mathbf{v_t}$ $[\mu s]$	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ $[\mu s]$	$\mathbf{v_t}$ $[\mu s]$	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ $[\mu s]$	$\mathbf{v_t}$ $[\mu s]$	$\mathbf{\Delta}$ [%]
0	2371.34	3092.67	−30.42	4637.67	6165.34	−32.94	6922.34	9266.34	−33.86
5	3124.00	3881.34	−24.24	6376.00	7956.34	−24.79	9440.34	11842.34	−25.44
10	3972.34	4844.00	−21.94	8203.67	9852.00	−20.09	12157.00	14513.67	−19.39

(b) All of the 10.000.000 events in the log-file violate the assumption.

Window w	Number of inputs i								
	5			10			15		
	$\mathbf{v_n}$ $[\mu s]$	$\mathbf{v_t}$ $[\mu s]$	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ $[\mu s]$	$\mathbf{v_t}$ $[\mu s]$	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ $[\mu s]$	$\mathbf{v_t}$ $[\mu s]$	$\mathbf{\Delta}$ [%]
0	1783.67	2099.00	−17.68	3505.00	4163.00	−18.77	5311.00	6270.34	−18.06
5	2535.00	2515.00	0.79	5222.34	5043.00	3.43	7839.00	7518.34	4.09
10	3409.34	2967.00	12.97	7071.00	6032.34	14.69	10560.67	8880.67	15.91

(c) Half of the 10.000.000 events in the log-file violate the assumption.



(d) Graphical representation of Figure 12a on the left, 12b in the middle, and 12c on the right.

Fig. 12: The results of the log-file analyses using the translations of an annotated specification is given. Entries in the table represent the time required by the monitor for one input event. The specification version $\mathbf{v_n}$ represents a specification that checks assumptions and assertion for each event in the log-file whereas the specification version $\mathbf{v_t}$ checks an assertions only if its corresponding assumption is violated by the use of output streams that use spawn, filter, and close conditions. The symbol $\mathbf{\Delta}$ represents the runtime effect of dynamic assertion checks, i.e., positive values indicate improvement and negative values indicate deterioration.

6 Conclusion

As both the relevance and the complexity of cyber-physical systems continue to grow, runtime monitoring is an essential ingredient of safety-critical systems. When monitors are derived from specifications it is crucial that the specifications are correct. In this paper, we have presented a sound verification approach for the stream-based monitoring language Lola. With this approach, the developer can formally prove guarantees on the streams computed by the monitor, and hence ensure that the monitor does not cause dangerous situations. The verification extension is motivated by upcoming aviation regulations and standards as well as by practical feedback of engineers.

The extension has been applied to previously written Lola specifications that were obtained based on interviews with aviation experts. In this process, we discovered and fixed several serious specification errors.

Further, since assumption can fail during runtime, they must be monitored and only when they are violated, their respective assertions need to be monitored as well. In this paper, we have efficiently monitored verified guarantees at runtime. Our experiments have shown that our Lola 2.0 encoding can significantly improve the monitors performance while maintaining a low overhead in case of few assumption violations. Yet, this improvement is highly dependent on the given specification. In general, simple assumptions and complex assertions benefit from this approach.

In the future, we plan to develop automatic invariant generation for Lola specifications. Another interesting direction for future work is to support the effort of [1] by exploiting the results of the analysis for the optimization of the specification and the resulting monitoring code. Finally, we plan to extend the verification approach to RTLola, the real-time extension of Lola.

Acknowledgement

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), by the European Research Council (ERC) Grant OSARES (No. 683300), and 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).

References

[1] Baumeister, J., Finkbeiner, B., Kruse, M., Schwenger, M.: Automatic optimizations for stream-based monitoring languages. In: Deshmukh, J., Ničković, D. (eds.) Runtime Verification. pp. 451–461. Springer International Publishing, Cham (2020)

[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.) Computer Aided Verification. pp. 28–39. Springer International Publishing, Cham (2020)

[3] Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: Fpga stream-monitoring of real-time properties. ACM Trans. Embed. Comput. Syst. **18**(5s) (oct 2019). https://doi.org/10.1145/3358220, https://doi.org/10.1145/3358220

[4] Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach, LNCS 4334, vol. 4334. Springer-Verlag (2007). https://doi.org/10.1007/978-3-540-69061-0

[5] Berry, G.: The Foundations of Esterel, p. 425–454. MIT Press, Cambridge, MA, USA (2000)

[6] Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with why3. International Journal on Software Tools for Technology Transfer **17**, 709–727 (2015)

[7] Cluzeau, J.M., Henriquel, X., van Dijk, L., Gronskiy, A.: Concepts of design assurance for neural networks (CoDANN). Tech. rep., EASA European Union Aviation Safety Agency (Mar 2020)

[8] D'Angelo, B., Sankaranarayanan, S., Sanchez, 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 (2005). https://doi.org/10.1109/TIME.2005.26

[9] Dauer, J.C., Finkbeiner, B., Schirmer, S.: Monitoring with verified guarantees. In: Feng, L., Fisman, D. (eds.) Runtime Verification. pp. 62–80. Springer International Publishing, Cham (2021)

[10] Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification. pp. 152–168. Springer International Publishing, Cham (2016)

[11] Finkbeiner, B., Oswald, S., Passing, N., Schwenger, M.: Verified rust monitors for lola specifications. In: Deshmukh, J., Ničković, D. (eds.) Runtime Verification. pp. 431–450. Springer International Publishing, Cham (2020)

[12] Floyd, R.W.: Assigning Meanings to Programs, pp. 65–81. Springer Netherlands, Dordrecht (1993), https://doi.org/10.1007/978-94-011-1793-7_4

[13] Gautier, T., Le Guernic, P., Besnard, L.: Signal: A declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture. pp. 257–277. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)

[14] Hagen, G., Tinelli, C.: Scaling up the formal verification of lustre programs with smt-based techniques. In: 2008 Formal Methods in Computer-Aided Design. pp. 1–9 (2008). https://doi.org/10.1109/FMCAD.2008.ECP.19

[15] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proceedings of the IEEE **79**(9), 1305–1320 (1991). https://doi.org/10.1109/5.97300

[16] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259, https://doi.org/10.1145/363235.363259

[17] Jagadeesan, L.J., Puchol, C., Von Olnhausen, J.E.: Safety property verification of Esterel programs and applications to telecommunications software. In: Wolper, P. (ed.) Computer Aided Verification. pp. 127–140. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)

[18] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[19] Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)

[20] Nagarajan, P., Kannan, S.K., Torens, C., Vukas, M.E., Wilber, G.F.: ASTM F3269 - An Industry Standard on Run Time Assurance for Aircraft Systems. https://doi.org/10.2514/6.2021-0525, https://arc.aiaa.org/doi/abs/10.2514/6.2021-0525

[21] Nenzi, L., Bortolussi, L., Ciancia, V., Loreti, M., Massink, M.: Qualitative and quantitative monitoring of spatio-temporal properties. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification. pp. 21–37. Springer International Publishing, Cham (2015)

[22] Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A hard real-time runtime monitor. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification. pp. 345–359. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[23] Reactive Systems Group, C.: RTLola. https://github.com/reactive-systems/

RTLola-Frontend, https://github.com/reactive-systems/RTLola-Interpreter (2023)

[24] Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 357–372. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

[25] Schirmer, S.: Runtime Monitoring with Lola. Master's thesis, Saarland University (Dec 2016)

[26] Schirmer, S., Torens, C., Adolf, F.: Formal monitoring of risk-based geofences. In: 2018 AIAA Information Systems-AIAA Infotech @ Aerospace (2018). https://doi.org/10.2514/6.2018-1986, https://arc.aiaa.org/doi/abs/10.2514/6.2018-1986

[27] Seto, D., Krogh, B., Sha, L., Chutinan, A.: The simplex architecture for safe online control system upgrades. In: Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207). vol. 6, pp. 3504–3508 vol.6 (1998). https://doi.org/10.1109/ACC.1998.703255

[28] Song, Y., Chin, W.N.: A synchronous effects logic for temporal verification of pure esterel. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 417–440. Springer International Publishing, Cham (2021)

Appendix A Lola Specifications – Experience Report

A.1 *Specification : gps_vel_output*

```
input sol_age: Float32
input hor_spd: Float32
input trk_gnd: Float32
input vert_spd: Float32
input time_s: UInt64
input time_us: UInt64
// Assumptions
assume <a1> time - time.offset(by: -1).defaults(to: time - 0.1) > 0.0
  and time - time.offset(by: -1).defaults(to: time - 0.1) <= 0.1
  and trace_pos >= 0

assume <a2> time - time.offset(by: -1).defaults(to: time - 0.1) > 0.0
  and time - time.offset(by: -1).defaults(to: time - 0.1) <= 0.1
  and trace_pos >= 0
// Frequency computations
output time := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: -1).defaults(to: -1.0) = -1.0 then time else start_time.offset(by:
    -1).defaults(to: -1.0)
output flight_time := time - start_time
output trace_pos @ sol_age or hor_spd or trk_gnd or vert_spd or time_s or time_us := trace_pos.offset(by:
    -1).defaults(to: -1) + 1
output frequency :=
    1.0 / ( time - time.offset(by: -1).defaults(to: time - 0.0001) )
output freq_sum :=
    freq_sum.offset(by: -1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
output freq_max := if frequency > freq_max.offset(by: -1).defaults(to: frequency) then frequency else
    freq_max.offset(by: -1).defaults(to: frequency)
output freq_min := if frequency < freq_min.offset(by: -1).defaults(to: frequency) then frequency else
    freq_min.offset(by: -1).defaults(to: frequency)
// Speed computations
output hor_spd_max := if hor_spd > hor_spd_max.offset(by: -1).defaults(to: 0.0) then hor_spd else
    hor_spd_max.offset(by: -1).defaults(to: 0.0)
output vert_spd_max := if vert_spd > vert_spd_max.offset(by: -1).defaults(to: 0.0) then vert_spd else
    vert_spd_max.offset(by: -1).defaults(to: 0.0)
// Solution age and track over ground (motion direction wrt. north)
trigger sol_age <= 0.5 "Sol age should remain zero!"
output trk_gnd_in_bound := if trk_gnd >= 0.0 and trk_gnd <= 360.0 then trk_gnd_in_bound.offset(by:
    -1).defaults(to: true) else false
output trk_gnd_max := if trk_gnd > trk_gnd_min.offset(by: -1).defaults(to: 0.0) then trk_gnd else
    trk_gnd_min.offset(by: -1).defaults(to: 0.0)
output trk_gnd_min := if trk_gnd < trk_gnd_max.offset(by: -1).defaults(to: 0.0) then trk_gnd else
    trk_gnd_max.offset(by: -1).defaults(to: 0.0)
// Assertions
assert <a1> time.offset(by: -1).defaults(to: -1.0) < time
  and start_time == start_time.offset(by: -1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: -1).defaults(to: 0.0)
assert <a2> frequency >= 10.0
  and freq_sum >= freq_sum.offset(by: -1).defaults(to: 0.0) + 10.0
assert <a3> trk_gnd_in_bound.offset(by: -1).defaults(to: true)
  or !trk_gnd_in_bound
```

A.2 Specification : gps_pos_output

```
import math
input lat: Float32
input lon: Float32
input hgt: Float32
input nObjs: UInt64
input nGPSL1: UInt64
input time_s: UInt64
input time_us: UInt64
// Assumptions
assume <a1> time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and trace_pos >= 0
// Frequency computations
output time: Float32 := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) == −1.0 then time else start_time.offset(by:
    −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @ lat or lon or hgt or nObjs or nGPSL1 or time_s or time_us := trace_pos.offset(by:
    −1).defaults(to: −1) + 1
output frequency := 1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) )
output freq_sum := freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
output freq_max := if frequency > freq_max.offset(by: −1).defaults(to: 0.0) then frequency else
    freq_max.offset(by: −1).defaults(to: 0.0)
output freq_min := if frequency < freq_min.offset(by: −1).defaults(to: 0.0) then frequency else
    freq_min.offset(by: −1).defaults(to: 0.0)
// Statistics
output lat_max := if lat > lat_max.offset(by: −1).defaults(to: lat) then lat else lat_max.offset(by:
    −1).defaults(to: lat)
output lat_min := if lat < lat_min.offset(by: −1).defaults(to: lat) then lat else lat_min.offset(by:
    −1).defaults(to: lat)
output lon_max := if lon > lon_max.offset(by: −1).defaults(to: lon) then lon else lon_max.offset(by:
    −1).defaults(to: lon)
output lon_min := if lon < lon_min.offset(by: −1).defaults(to: lon) then lon else lon_min.offset(by:
    −1).defaults(to: lon)
output lat_in_bound := max( abs(lat_max), abs(lat_min) ) <= 90.0
output lon_in_bound := max( abs(lon_max), abs(lon_min) ) <= 180.0
trigger !lat_in_bound "Irregular latitude value!"
trigger !lon_in_bound "Irregular longitude value!"
output begin := false
output start_height := if begin.offset(by: −1).defaults(to: true) then hgt else start_height.offset(by:
    −1).defaults(to: 0.0)
output hgt_inc_max := max( hgt_inc_max.offset(by: −1).defaults(to: 0.0), hgt − start_height )
output hgt_dec_max := min( hgt_dec_max.offset(by: −1).defaults(to: 0.0) , hgt − start_height )
trigger hgt_inc_max > 100.0 "Never increase height by more than 100m!"
trigger hgt_dec_max < −100.0 "Never decrease height by more than 100m"
// Assertions
assert <a1> time.offset(by: −1).defaults(to: −1.0) < time
  and start_time == start_time.offset(by: −1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: −1).defaults(to: 0.0)
assert <a2> hgt_inc_max >= 0.0 and hgt_dec_max <= 0.0
  and hgt_inc_max >= hgt_inc_max.offset(by: −1).defaults(to: 0.0)
  and hgt_dec_max <= hgt_inc_max.offset(by: −1).defaults(to: 0.0)
```

```
  and start_height = start_height.offset(by: −1).defaults(to: start_height)
  and (lat_in_bound.offset(by: −1).defaults(to: true) or !lat_in_bound)
  and (lon_in_bound.offset(by: −1).defaults(to: true) or !lon_in_bound)
```

A.3 *Specification : imu_output*

```
import math
input ax: Float32
input ay: Float32
input az: Float32
input time_s: UInt64
input time_us: UInt64
input counter: Int64
// Assumptions
assume <a1> time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and trace_pos >= 0
assume <a2> ax != ax.offset(by: −1).defaults(to: ax + 0.1)
  and ay != ay.offset(by: −1).defaults(to: ay + 0.1)
  and az != az.offset(by: −1).defaults(to: az + 0.1)
// Frequency computations
output time := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) == −1.0 then time else start_time.offset(by:
    −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @ ax or ay or az or time_s or time_us or counter := trace_pos.offset(by: −1).defaults(to:
    −1) + 1
output frequency := 1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) )
output freq_sum := freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
// Statistics
output deviation := abs( frequency − 100.0)
output exceeds_worst := deviation > worst_dev.offset(by: −1).defaults(to: 0.0)
output worst_dev_pos := if exceeds_worst then trace_pos else worst_dev_pos.offset(by: −1).defaults(to: 0)
output worst_dev := if exceeds_worst then deviation else worst_dev.offset(by: −1).defaults(to: 0.0)
output ax_max := max(abs(ax),ax_max.offset(by:−1).defaults(to:0.0))
output ay_max := max(abs(ay),ay_max.offset(by:−1).defaults(to:0.0))
output az_max := max(abs(az),az_max.offset(by:−1).defaults(to:0.0))
trigger ax > 15.0 or ay > 15.0 or az > 15.0
output frozen_ax := ax.offset(by:−1).defaults(to:0.0) = ax
  and ax.offset(by:−2).defaults(to:0.0)=ax.offset(by:−1).defaults(to:0.0)
  and ax.offset(by:−3).defaults(to:0.0)=ax.offset(by:−2).defaults(to:0.0)
  and ax.offset(by:−4).defaults(to:0.0)=ax.offset(by:−3).defaults(to:0.0)
  and ax.offset(by:−5).defaults(to:0.1)=ax.offset(by:−4).defaults(to:0.0)
output frozen_ay := ay.offset(by:−1).defaults(to:0.0) = ay
  and ay.offset(by:−2).defaults(to:0.0)=ay.offset(by:−1).defaults(to:0.0)
  and ay.offset(by:−3).defaults(to:0.0)=ay.offset(by:−2).defaults(to:0.0)
  and ay.offset(by:−4).defaults(to:0.0)=ay.offset(by:−3).defaults(to:0.0)
  and ay.offset(by:−5).defaults(to:0.1)=ay.offset(by:−4).defaults(to:0.0)
output frozen_az := az.offset(by:−1).defaults(to:0.0) = az
  and az.offset(by:−2).defaults(to:0.0)=az.offset(by:−1).defaults(to:0.0)
  and az.offset(by:−3).defaults(to:0.0)=az.offset(by:−2).defaults(to:0.0)
  and az.offset(by:−4).defaults(to:0.0)=az.offset(by:−3).defaults(to:0.0)
  and az.offset(by:−5).defaults(to:0.1)=az.offset(by:−4).defaults(to:0.0)
trigger frozen_ax or frozen_ay or frozen_az
```

```
output check_counter := if trace_pos = 0 then false else (counter != (counter.offset(by: −1).defaults(to: −1) + 1) % 100)
trigger check_counter "A counter value was ignored."
// Assertions
assert <a1> time.offset(by: −1).defaults(to: −1.0) < time
  and start_time == start_time.offset(by: −1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: −1).defaults(to: 0.0)
assert <a2> !frozen_ax and !frozen_ay and !frozen_az
```

A.4 *Specification : nav_output*

```
import math
input lat: Float32
input lon: Float32
input ug: Float32
input vg: Float32
input wg: Float32
input time_s: UInt64
input time_us: UInt64
// Assertion
assume <a1> trace_pos >= 0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
// Frequency Computation
output time := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) == −1.0 then time else start_time.offset(by: −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @lat or lon or ug or vg or wg or time_s or time_us := trace_pos.offset(by: −1).defaults(to: −1) + 1
output frequency := 1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) )
output freq_sum := freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
output freq_max := if frequency > freq_max.offset(by: −1).defaults(to: frequency) then frequency else freq_max.offset(by: −1).defaults(to: frequency)
output freq_min := if frequency < freq_min.offset(by: −1).defaults(to: frequency) then frequency else freq_min.offset(by: −1).defaults(to: frequency)
// Statistics
output velocity := sqrt( ug*ug + vg*vg + wg*wg)
output lon1_rad := lon.offset(by: −1).defaults(to: 0.0) * 3.1415926535 / 180.0
output lon2_rad := lon * 3.1415926535 / 180.0
output lat1_rad := lat.offset(by: −1).defaults(to: 0.0) * 3.1415926535 / 180.0
output lat2_rad := lat * 3.1415926535 / 180.0
output dlon := lon2_rad − lon1_rad
output dlat := lat2_rad − lat1_rad
output a := (sin(dlat/2.0))*(sin(dlat/2.0)) + cos(lat1_rad) * cos(lat2_rad) * (sin(dlon/2.0))*(sin(dlon/2.0))
output x_atan2 := sqrt(a)
output y_atan2 := sqrt(1.0−a)
output c := 2.0 * if x_atan2 > 0.0 then arctan(y_atan2/x_atan2)
  else if x_atan2 < 0.0 and y_atan2 >= 0.0
    then arctan(y_atan2/x_atan2) + 3.1415926535
  else if x_atan2 < 0.0 and y_atan2 < 0.0
    then arctan(y_atan2/x_atan2) − 3.1415926535
  else if x_atan2 = 0.0 and y_atan2 > 0.0 then 3.1415926535 / 2.0
  else if x_atan2 = 0.0 and y_atan2 < 0.0 then −3.1415926535 / 2.0
```

```
  else 0.0
output gps_distance := 6373000.0 * c
output passed_time := time - time.offset(by: -1).defaults(to: 0.0)
output distance_max := velocity * passed_time
output dif_distance := abs( gps_distance - distance_max )
output detected_jump :=if trace_pos=0 then false else dif_distance>1
trigger detected_jump "Jump!"
// Assertions
assert <a1> time.offset(by: -1).defaults(to: -1.0) < time
  and start_time == start_time.offset(by: -1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: -1).defaults(to: 0.0)
assert <a2> (!detected_jump or gps_distance > distance_max)
  or (!detected_jump or distance_max > gps_distance)
```

A.5 *Specification : tagging*

```
import math
input time_s: UInt64
input time_us: UInt64
input vel: Float64
// Assumptions
assume<a1> (time_s = time_s.offset(by: -1).defaults(to: 0)
  and time_us > time_us.offset(by: -1).defaults(to: 0))
  and ( time_s > time_s.offset(by: -1).defaults(to: 0)
      or time_us > time_us.offset(by: -1).defaults(to: 0))
// Exemplary State Statistics
output time := cast(time_s) + cast(time_us) / 1000000.0
output correct_vel := abs( vel ) < 0.3
output cur_state := if correct_vel then
  if cur_state.offset(by: -1).defaults(to: 0) = 0 then 1 else 2 else 0
output start_interval := cur_state = 2
output interval_start := if start_interval then interval_start.offset(by: -1).defaults(to: 0.0) else time
trigger start_interval "Interval started!"
output end_interval := cur_state.offset(by: -1).defaults(to: 0) > 0 and !correct_vel and time_since_start >
    5.0
trigger end_interval "Interval ended!"
output time_since_start := time - interval_start.offset(by: -1).defaults(to: 0.0)
// Assertions
assert <a1> !(start_interval and end_interval)
  and time_since_start > 0.0
```

A.6 *Specification : ctrl_output*

```
import math
input time_s: UInt64
input time_us: UInt64
input vel_x: Float64
input vel_y: Float64
input vel_z: Float64
input fuel: Float64
input power: Float64
input vel_r_x: Float64
input vel_r_y: Float64
input vel_r_z: Float64
```

```
// Assumptions
assume <a1> trace_pos >= 0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
assume<a2> power > 0.0
  and power <= power.offset(by: −1).defaults(to: power)
  and fuel > 0.0 and fuel < fuel.offset(by: −1).defaults(to: fuel + 0.1)
  and (time_s = time_s.offset(by: −1).defaults(to: 0)
  and time_us > time_us.offset(by: −1).defaults(to: 0))
  and (time_s > time_s.offset(by: −1).defaults(to: 0)
       or time_us > time_us.offset(by: −1).defaults(to: 0))
// Frequency computations
output time := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) == −1.0 then time else start_time.offset(by:
    −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @ time_s or time_us or vel_x or vel_y or vel_z or fuel or power or vel_r_x or vel_r_y or
    vel_r_z := trace_pos.offset(by:−1).defaults(to:−1) + 1
output frequency := 1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) ) // major improvement
output freq_sum := freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
output freq_max := if frequency > freq_max.offset(by: −1).defaults(to: frequency) then frequency else
    freq_max.offset(by: −1).defaults(to: frequency)
output freq_min := if frequency < freq_min.offset(by: −1).defaults(to: frequency) then frequency else
    freq_min.offset(by: −1).defaults(to: frequency)
// Exemplary phase detection
output velocity := sqrt( vel_x∗vel_x + vel_y∗vel_y + vel_z∗vel_z )
output velocity_max := if reset_max.offset(by: −1).defaults(to: false) then velocity else max( velocity,
    velocity_max.offset(by: −1).defaults(to: 0.0) )
output velocity_min := if reset_max.offset(by: −1).defaults(to: false) then velocity else min( velocity,
    velocity_min.offset(by: −1).defaults(to: 0.0) )
output dif_max := abs(velocity_max − velocity_min)
output reset_max := dif_max > 1.0
output reset_time := if reset_max or trace_pos = 0 then time else reset_time.offset(by: −1).defaults(to: 0.0)
output unchanged := if reset_max.offset(by: −1).defaults(to: false) then 0 else unchanged.offset(by:
    −1).defaults(to: 0) + 1
trigger unchanged = 150 "Phase detected!"
// Statistics
output vel_dev := abs(vel_r_x−vel_x) + abs(vel_r_y−vel_y) + abs(vel_r_z−vel_z)
output dev_sum := vel_dev + dev_sum.offset(by: −1).defaults(to: 0.0)
output vel_av := dev_sum / cast((trace_pos+1)∗3)
output worst_dev_pos := if worst_dev.offset(by: −1).defaults(to: vel_dev − 1.0) < vel_dev then trace_pos
    else worst_dev_pos.offset(by: −1).defaults(to: 0)
output worst_dev := if worst_dev.offset(by: −1).defaults(to: vel_dev − 1.0) < vel_dev then vel_dev else
    worst_dev.offset(by: −1).defaults(to: 0.0)
output start_fuel := start_fuel.offset(by: −1).defaults(to: fuel)
output fuel_level := 1 − ( start_fuel − fuel ) / start_fuel
output fuel_half := fuel_level < 0.50
output fuel_warning := fuel_level < 0.25
output fuel_danger := fuel_level < 0.10
output start_power := start_power.offset(by: −1).defaults(to: power)
output power_p_consumed := ( (start_power − power) / (start_power) )
trigger_once fuel_half "INFO: Fuel level is half reduced"
trigger_once fuel_warning "WARNING: Fuel level is below 25%"
trigger_once fuel_danger "DANGER: Fuel level is below 10%"
```

```
trigger_once power_p_consumed > 0.50 "Power below half capacity"
trigger_once power_p_consumed > 0.75 "Power below quarter capacity"
trigger_once power_p_consumed > 0.90 "Urgent: Recharge Power!"
// Assertions
assert <a1> time.offset(by: −1).defaults(to: −1.0) < time
  and start_time == start_time.offset(by: −1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: −1).defaults(to: 0.0)
assert<a2> reset_time >= 0.0
  and start_fuel >= fuel and start_power >= power
  and (!fuel_half.offset(by: −1).defaults(to: false) or fuel_half)
  and (!fuel_warning.offset(by: −1).defaults(to: false)or fuel_warning)
  and (!fuel_danger.offset(by: −1).defaults(to: false) or fuel_danger)
  and power_p_consumed >= power_p_consumed.offset(by: −1).defaults(to: power_p_consumed)
```

A.7 *Specification* : *mm_output_1*

```
import math
input stateID_SC: UInt64
// Assumptions
assume<a1> trace_pos >= 0
// Exemplary state transition analysis
output trace_pos @ stateID_SC := trace_pos.offset(by: −1).defaults(to: −1) + 1
output change_state := if trace_pos = 0 then false
  else stateID_SC != stateID_SC.offset(by: −1).defaults(to: 0)
output transitions := if stateID_SC.offset(by:−1).defaults(to: 0) = 0 then stateID_SC == 1
  else if stateID_SC.offset(by: −1).defaults(to: 0) == 1 then stateID_SC == 1 or stateID_SC == 2
  else if stateID_SC.offset(by: −1).defaults(to: 0) == 2 then stateID_SC == 1 or stateID_SC == 3
  else if stateID_SC.offset(by: −1).defaults(to: 0) == 3 then stateID_SC == 3
  else false
output invalid_transitions := change_state and !transitions
trigger invalid_transitions "Invalid state transition"
// Assertions
assert <a1> invalid_transitions or
!( stateID_SC.offset(by: −1).defaults(to: 0) != 0 and stateID_SC = 0 )
assert <a2> (stateID_SC == 1 or stateID_SC == 2 or stateID_SC == 3 )
  or !( stateID_SC.offset(by: −2).defaults(to: 0) = 1
   and transitions.offset(by: −1).defaults(to: false) and transitions )
```

A.8 *Specification* : *mm_output_2*

```
import math
input time_s: UInt64
input time_us: UInt64
input stateID_SC: Int64
input OnGround: UInt64
// Assumptions
assume <a1> trace_pos >= 0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
assume <a2> (time_s = time_s.offset(by: −1).defaults(to: 0)
  and time_us > time_us.offset(by: −1).defaults(to: 0))
  and (time_s > time_s.offset(by: −1).defaults(to: 0)
       or time_us > time_us.offset(by: −1).defaults(to: 0))
// Frequency computations
```

```
output time := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) == −1.0 then time else start_time.offset(by:
    −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @ time_s or time_us or stateID_SC or OnGround := trace_pos.offset(by: −1).defaults(to:
    −1) + 1
output frequency := 1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) )
output freq_sum := freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
// Phase Statistics
output change_state := if trace_pos = 0 then false
      else stateID_SC != stateID_SC.offset(by: −1).defaults(to: 0)
trigger change_state
output entrance_time := if change_state then time
        else entrance_time.offset(by: −1).defaults(to: time)
output hover_end := change_state and stateID_SC.offset(by: −1).defaults(to: −1) = 4
output hover_cur_time := if hover_end then
time − entrance_time.offset(by: −1).defaults(to: 0.0)else 0.0
output hover_sum_time := hover_sum_time.offset(by: −1).defaults(to: 0.0) + hover_cur_time
output hover_num_times := hover_num_times.offset(by: −1).defaults(to: 0) + if hover_end then 1 else 0
output hover_max_time := max ( hover_max_time.offset(by: −1).defaults(to: 0.0), hover_cur_time )
output hover_avg_time := if hover_num_times != 0 then hover_sum_time / cast(hover_num_times) else 0.0
output landing_info := if change_state and stateID_SC = 5 then 0.0 else time − entrance_time.offset(by:
    −1).defaults(to: time)
output landing_error := stateID_SC = 5 and OnGround != 1 and landing_info > 20.0
// Assertions
assert <a1> time.offset(by: −1).defaults(to: −1.0) < time
  and start_time == start_time.offset(by: −1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: −1).defaults(to: 0.0)
assert <a2> time >= entrance_time and start_time <= entrance_time
  and hover_cur_time >= 0.0 and hover_max_time <= flight_time
assert <a3> !(landing_error and hover_end)
  and (!landing_error or landing_info > 0.0)
```

A.9 *Specification : contingency_output*

```
input avgDist_laser: Float64
input actual_laser: Float64
input static_laser: Float64
input avgDist_optical:Float64
input actual_optical: Float64
input static_optical: Float64
// Assumptions
assume <a1> avgDist_laser >= 0.0 and actual_laser >= 0.0
  and static_laser >= 0.0 and avgDist_optical >= 0.0
  and actual_optical >= 0.0 and static_optical >= 0.0
  and (avgDist_laser + actual_laser + static_laser > 0.0)
  and (avgDist_optical + actual_optical + static_optical > 0.0)
// Trust computations
output rating_laser := 0.2 ∗ static_laser + 0.4 ∗ actual_laser
     + 0.4 ∗ avgDist_laser
output rating_optical := 0.2 ∗ static_optical + 0.4 ∗ actual_optical + 0.4 ∗ avgDist_optical
output trust_laser := rating_laser / ( rating_laser + rating_optical)
output trust_optical := 1.0 − trust_laser
trigger trust_laser >= 0.5 "Trust in laser"
```

```
trigger trust_optical > 0.5 "Trust in optical sensor"
// Assertions
assert <a1> trust_laser ≠ trust_optical
```

A.10 *Specification* : *health_output*

```
import math
// average distance to the measured ostacle (range of sight) using laser
input avgDist_laser: Float64
// average distance to the measured ostacle (range of sight) using camera
input avgDist_optical: Float64
input vel: Float64
// Assumption
assume <a1> avgDist_laser <= 100.0 and avgDist_laser >= 0.0 // both in m
  and avgDist_optical <= 50.0 and avgDist_optical >= 0.0 // both in m
  and abs(vel) < 5.5 // in m/s
// Line of sight
output avgDst_dif := min( avgDist_laser, avgDist_optical ) − abs(vel)
trigger avgDst_dif < 5.0 "WARNING: Dynamic Velocity Limit reached"
trigger avgDst_dif < 2.0 "ERROR: Abort mission."
// Assertions
assert <a1> avgDst_dif < 54.5 and avgDst_dif > −5.5
```