Verified Rust Monitors for Lola Specifications*

Bernd Finkbeiner[0000−0002−4280−8441], Stefan Oswald,
Noemi Passing[0000−0001−7781−043X], and
Maximilian Schwenger[0000−0002−2091−7575]

CISPA Helmholtz Center for Information Security
66123 Saarbrücken, Germany
{finkbeiner,noemi.passing,maximilian.schwenger}@cispa.saarland
s.oswald@stud.uni-saarland.de

Abstract. The safety of cyber-physical systems rests on the correctness of their monitoring mechanisms. This is problematic if the specification of the monitor is implemented manually or interpreted by unreliable software. We present a *verifying compiler* that translates specifications given in the stream-based monitoring language Lola to implementations in Rust. The generated code contains verification annotations that enable the Viper toolkit to automatically prove functional correctness, absence of memory faults, and guaranteed termination. The compiler parallelizes the evaluation of different streams in the monitor based on a dependency analysis of the specification. We present encouraging experimental results obtained with monitor specifications found in the literature. For every specification, our approach was able to either produce a correctness proof or to uncover errors in the specification.

1 Introduction

Cyber-physical systems are inherently safety-critical, because failures immediately impact the physical environment. A crucial aspect of the development of such systems is therefore the integration of reliable monitoring mechanisms. A *monitor* is a special system component that typically has broad access to the sensor readings and the resulting control decisions. The monitor assesses the system's health by checking its behavior against a specification. If a violation is detected, the monitor raises an alarm and initiates mitigation protocols such as an emergency landing or a graceful shutdown.

An obvious concern with this approach is that the safety of the system rests on the correctness of the monitor. *Quis custodiet ipsos custodes?* For simple specifications, this is not a serious problem. An LTL [25] specification, for example, can be translated into a finite-state automaton that is proven to correspond to the semantics of the specification. Implementing such an automaton correctly

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

as a computer program is not difficult. For more expressive specification languages, establishing the correctness of the monitor is much more challenging. Especially problematic is the use of interpreters, which read the specification as input and then rely on complicated and error-prone software to interpret the specification dynamically at runtime [3, 9, 11–13]. Recently, however, much effort has gone into the development of compilers. Compared to a full-scale interpreter, the code produced by a compiler for a specific specification is fairly simple and well-structured. Some compilers even include special mechanisms that increase the confidence in the monitor. For example, the RTLola compiler [4] generates VHDL code that is annotated with tracing information that relates each line of code back to the specific part of the specification it implements. The Copilot compiler [23] produces a test suite for the generated C code. The framework even includes a bounded model checker, which can check the correctness of the output for input sequences up to a fixed length. However, none of these approaches actually proves the functional correctness of the monitor.

In this paper, we present a *verifying compiler* that translates specifications given in the stream-based monitoring language Lola [10] to implementations in Rust[1]. The generated code is fully annotated with formal function contracts, loop invariants, and inline assertions, so that functional correctness and guaranteed termination can be automatically verified by the Viper [19] toolkit, without any restriction on the length of the input trace. Since the memory requirements of a Lola specification can be computed statically, this yields a formal guarantee that on any platform that satisfies these requirements, the monitor will never crash and will always compute the correct output.

A major practical concern for any compiler is the performance of the generated code. Our Lola-to-Rust compiler produces highly efficient monitor implementations because it parallelizes the code for the evaluation of the specifications. Since Lola is a stream-based specification language, it exhibits a highly modular and memory-local structure, i.e., the computation of a stream writes only in its own local memory, although it may read from the local memory of several other processes. The compiler statically analyzes the dependencies between the streams, resulting in a partial evaluation order. To prove correctness, it is shown that streams that are incomparable with respect to the evaluation order can indeed be evaluated in parallel.

We have used our compiler to build monitors from specifications of varying sizes found in the literature. In our experience, the compiler itself scales very well. The verification in Viper, however, is expensive. It appears that the running times of the underlying SMT solver Z3 [18] vary greatly, even for different runs on the same monitor and specification. Nevertheless, we have been successful in all our benchmarks in the sense that the compiler either generated a verified monitor or uncovered an error in the specification. This is a major step forward towards the *verified monitoring* of real-life safety-critical systems.

[1] `https://www.rust-lang.org/`

2 Introduction to Lola

The source language of our verifying compiler is the stream-based monitoring language Lola [10]. A Lola monitor is a reactive component that translates, in an online fashion, input streams into output streams. In each time step, the monitor receives new values for the input streams and produces new values for the output streams in accordance with the specification. In principle, the monitoring can continue forever; if the monitor is terminated, it wraps up the remaining computations, produces a final output, and shuts down. Lola specifications are declarative in the sense that the semantics leaves a lot of implementation freedom: the semantics defines how specific values are combined arithmetically and logically, but the precise evaluation order and the memory management are determined by the implementation.

A Lola specification defines a set of streams. Each stream is an ordered sequence of typed values that is extended throughout the monitor execution. There are three kinds of streams:

Input Streams constitute the interface between the monitor and an external data source, i.e., the system under scrutiny.

Output Streams compute new values based on input streams, other output streams, and constant values. The computed values contain relevant information regarding the performance and health status of the system.

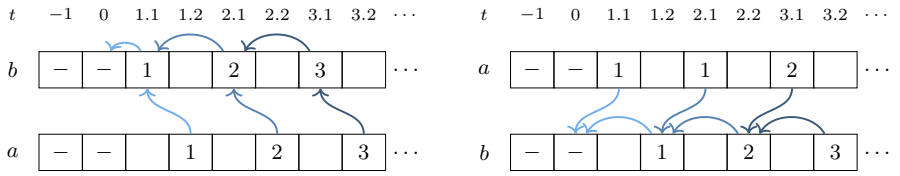
Triggers constitute the interface between the monitor and the user. Trigger values are binary and indicate the violation of a property. In this case, the monitor alerts the user.

Syntactically, a Lola specification is given as a sequence of stream declarations. Input stream declarations are of the form $i_j : T_j$, where i_j is an input stream and T_j is its type. Output stream and trigger declarations are of the form $s_j : T_j = e_j(i_1, \ldots, i_m, s_1, \ldots, s_n)$, where $i_1, \ldots, i_m$ are input streams, $s_1, \ldots, s_n$ are output streams, and the e_j are stream expressions. A stream expression consists of constant values, streams, arithmetic and logic operators $f(e_1, \ldots, e_k)$, if-then-else expressions $\texttt{ite}(b, e_1, e_2)$, and stream accesses $e[k, c]$, where e is a stream, k is the *offset*, and c is the constant *default value*. Stream accesses are either *synchronous*, i.e., a stream accesses the latest value of a stream, or *asynchronous*, i.e., a stream accesses a past or future value of another stream.

The example specification shown in Listing 1.1 monitors the altitude of a drone, detects whether the drone flies below a given minimum altitude or above a given maximum altitude for too long, and raises an alarm if needed. The input stream `altitude` contains sensor information of the drone. The output stream `tooLow` checks whether the altitude is lower than the given minimum altitude of `200` in the last, current, and next step, denoted by `altitude[-1,0]`, `altitude`, and `altitude[1,0]`, respectively. If this is the case, a trigger is raised. Analogously, `tooHigh` checks whether the altitude is above the given maximum altitude in the last, current, and next step, and a trigger is raised in this case. The evaluations of `tooHigh` and `tooLow` try to access the second to last value of `altitude` as well as the last and the next one. If `altitude`

```
input altitude: Int32
output tooLow: Bool :=
  altitude[-1,0] < 200 ∧ altitude < 200 ∧ altitude[1,0] < 200
output tooHigh: Bool :=
  altitude[-1,0] > 600 ∧ altitude > 600 ∧ altitude[1,0] > 600
trigger tooLow "Flying below minimum altitude."
trigger tooHigh "Flying above maximum altitude."
```

Listing 1.1: A Lola specification monitoring the altitude of a drone. The output stream `tooLow` (`tooHigh`) checks whether the drone flies below (above) a given minimum (maximum) altitude in the last, current, and next step. If this is the case, an alarm is raised.



(a) The result of evaluating the output streams respecting the evaluation order.

(b) The result of evaluating the output streams in order of their declaration.

Fig. 1: Two different evaluations of the output streams a and b, where a accesses b synchronously and b accesses its previous value. Both accesses default to 0 and both a and b increase the obtained value by 1.

does not have at least two values, the accesses with offset -1 fail and the default value, in this case 0, is used. If `altitude` ceases to produce values, the accesses with offset 1 fail. Hence, in contrast to negative offsets, the default value for accesses with positive offset is used at the end of the execution.

The semantics of Lola is defined in terms of *evaluation models*. Intuitively, an evaluation model consists of evaluations of each output stream of the specification. The evaluation is a natural translation of the stream expressions. The full formal definition is given in [10].

Definition 1 (Evaluation Model [10]). *Let φ be a Lola specification over input streams $i_1, \ldots, i_\ell$ and output streams $s_1, \ldots, s_n$. The tuple $\langle \sigma_1, \ldots, \sigma_n \rangle$ of streams of length $N + 1$ is called an* evaluation model *if for each equation $s_j = e_j(i_1, \ldots, i_\ell, s_1, \ldots, s_n)$ in φ, $\langle \sigma_1, \ldots, \sigma_n \rangle$ satisfies $\sigma_j(k) = v(e_j)(k)$ for $0 \leq k \leq N$, where $v(e_j)(k)$ evaluates the stream expression e_j at position k.*

Synchronous accesses harbor a pitfall for the monitor realization as illustrated in Figure 1. Consider the corresponding Lola specification:

```
output a: Int32 := b[ 0, 0] + 1
output b: Int32 := b[-1, 0] + 1
```

Here, a accesses b synchronously, while b accesses its previous value. The evaluation of a tries to access the current value of b and increases the result by one, which yields the next stream value of a. In contrast, the evaluation of b tries to access the last value of b and increases the result by one to determine the next stream value of b. Figure 1a depicts the resulting output. If the monitor evaluates the streams in order of their declaration, however, the resulting output, shown in Figure 1b, differs from the expected one. The reason is that the *current* value of b changes depending on whether or not b has already been extended when accessing the value. This problem is solved by respecting the evaluation order, a partial order on the output streams. It is induced by the dependency graph of a Lola specification.

Definition 2 (Dependency Graph [10]). *The* dependency graph $D_\varphi = (V, E)$ *of a Lola specification φ is a weighted directed multigraph. Each vertex represents a stream and each edge an access operation. Thus, $s \in V$ iff s is a stream or trigger in φ and $(s_1, n, s_2) \in E$ for $s_1, s_2 \in V$, $n \in \mathbb{N}$ iff the stream expression of s_1 contains an access to s_2 with offset n.*

Based on the dependency graph, d'Angelo et al. define the *shift* of a stream [10]. Intuitively, the shift of s indicates how many steps the evaluation of its expression needs to be delayed. For instance, suppose the delay is $n > 0$. Then the value of s for time t can be computed at time $t + n$.

Definition 3 (Shift [10]). *For a Lola specification φ, the* shift $\Delta(s)$ *of a stream s is the greatest weight of a path through the dependency graph of φ originating in s: $\Delta(s) = \max(0, \max\{w + \Delta(s') | (s, w, s') \in E\})$.*

The shift allows us to define an order in which streams need to be evaluated. For this, we define the set of synchronized edges E^* where the weight of a synchronized edge $(s, n, s') \in E^*$ indicates when s can access s' successfully with an offset of n. Let $E^* = \{(s, \Delta(s) - w - \Delta(s'), s') \mid (s, w, s') \in E\}$.

Definition 4 (Evaluation Order). *The* evaluation order $\leq_{eo}$ *is a partial order on the output streams of a Lola specification φ. Let $D_\varphi = (V, E)$ be the dependency graph of φ. The evaluation order is the transitive closure of a relation $\prec$ with $s \prec s'$ iff $(s', 0, s) \in E^*$.*

Clearly, we obtain $b \leq_{eo} a$ for the above Lola specification, yielding the expected result depicted in Figure 1a. For the Lola specification from Listing 1.1, however, the output streams `tooLow` and `tooHigh` are incomparable according to the evaluation order. A total evaluation order on the output streams, denoted $\leq_{eo}^+$, is obtained by relating incomparable streams arbitrarily.

Remark 1 (On Asynchronous Accesses and Off-by-one Errors). It is fairly easy to make off-by-one errors in asynchronous stream accesses. When two streams within one layer access each other asynchronously, one of the offsets needs to be decreased by 1, depending on which stream is evaluated first. This cannot be avoided for any $\leq_{eo}^+$. To simplify the presentation, we will ignore this issue in the remainder of the presentation, the correct adjustment of the indices is, however, implemented in the compiler.

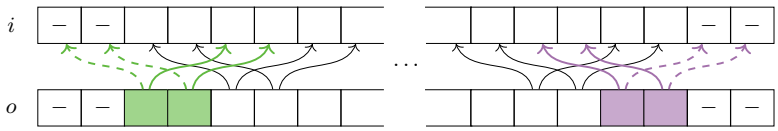


Fig. 2: Illustration of stream accesses in different phases of the execution. An output stream o accesses an input stream i with offsets -2 and $+2$. In the prefix (postfix) of the execution, the past (future) accesses need to be substituted by their default values.

Specifications where the dependency graph has no positive cycles are called *efficiently monitorable*: such specifications can be monitored with constant memory, and an output value can always be produced after a constant delay [10]. All example specifications considered in this paper are efficiently monitorable.

3 From Lola to Rust

The compilation proceeds in two steps. First, the Lola specification is analyzed to determine inter-stream dependencies, the overall memory requirement, and the different phases of the monitoring process. Second, the compiler produces Rust code that implements the specification.

3.1 Specification Analysis

Execution Pre- and Postfix. Refer back to the Lola specification in Listing 1.1. Another beneficial property of the synchronous input model is that, starting from $t = 2$, both stream accesses with offset -1 to `altitude` will always succeed since the offset refers to the last evaluation of `altitude` which did already happen at $t \geq 1$. For a more general analysis, suppose an output stream s accesses another stream s' with an offset of n. If n is non-positive, then accesses may fail until $t = \Delta(s) - n - \Delta(s')$, i.e., they will not fail from $\Delta(s) - n - \Delta(s') + 1$ on. If n is strictly positive, however, the evaluation of s needs to be delayed by $\Delta(s) - n$, i.e., until s' received the respective value. By generally delaying the execution of s, all accesses to s' continue to succeed until s' ceases to produce new values. As soon as this is the case, the monitor needs to evaluate s for $\Delta(s) - n$ more times to compensate for the delay. For instance, the evaluations of `tooLow` and `tooHigh` both have to be delayed by one step.

This behavior induces the structure of the monitor execution: it starts with a prefix where past accesses always fail, loops in the regular execution where all accesses always succeed, and ends in a postfix where future accesses always fail.

Figure 2 illustrates stream accesses in the different phases. It shows an output stream o that accesses an input stream i with an offset of -2 and 2. In the first two iterations of the monitor execution, i.e., in the prefix, the accesses to the past values will fail, requiring the monitor to use the default values instead.

Afterwards, all accesses succeed until the input stream ends. In the last two evaluations, i.e., in the postfix, the future accesses fail and need to be replaced by the default values.

While the shift only concerns time, it can also be used to compute the memory requirement of a stream, i.e., the number of values of a single stream that can be relevant at the same time. If a stream s of type T has a memory requirement $\mu(s) = i$, the monitor needs to reserve $i \cdot \text{size}(T)$ bytes of memory for s.

Definition 5 (Memory Requirement). *The* memory requirement *of a dependency $(s', w, s) \in E$ is determined by the shifts of the streams as well as the weight w of the dependency, i.e., the offset of the stream access: $\Delta(s) - \Delta(s') - w$. The memory requirement of a stream is thus the maximum requirement of any outgoing dependency: $\mu(s) = \max\{\Delta(s) - \Delta(s') - w \mid (s', w, s) \in E\}$.*

Hence, the compilation determines three key values for each specification.

Definition 6 (Memory Consumption, Prefix- and Postfix Length). *Let μ^*_φ, $\eta^{\leftarrow}_\varphi$, and $\eta^{\rightarrow}_\varphi$ be the* memory consumption, prefix length *and* postfix length *of φ, respectively, defined as follows:*

$$\mu^*_\varphi = \sum_{s \in \varphi} \{\mu(s) \cdot \text{size}(T_s)\}$$
$$\eta^{\leftarrow}_\varphi = \max_{s \in \varphi} \{\Delta(s) + \mu(s)\}$$
$$\eta^{\rightarrow}_\varphi = \max_{s \in \varphi} \{\Delta(s)\}$$

Furthermore, the evaluation order $\leq_{eo}$ of the output streams of a Lola specification induces the so-called *evaluation layers*.

Definition 7 (Evaluation Layer). *Let φ be a Lola specification and let $\leq_{eo}$ be the evaluation order induced by its dependency graph. If $Layer(s) = k$ for an output stream s, then there is a strictly decreasing sequence of k streams with respect to $\leq_{eo}$ starting in s.*

Intuitively, an evaluation layer consists of all streams that are incomparable according to the evaluation order. For the Lola specification from Listing 1.1, for instance, the output streams `tooLow` and `tooHigh` are incomparable according to the evaluation order. Thus, they are contained in the same evaluation layer. Evaluation Layers are also used to identify independent streams and thus to enable their concurrent evaluation as described in Sect. 5.

3.2 Code Generation

The monitor code starts with a *prelude* which declares data structures and helper functions. It also contains the `main` function starting with the static allocation of the working memory. The remainder of the `main` function is the operative monitoring code consisting of three components: the *execution prefix*, the *monitor loop*, and the *execution postfix*. The general structure is illustrated in Listing 1.2, details follow in the remainder of this section.

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

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

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

Listing 1.2: Structure of the generated Rust code.

Prelude. The prelude declares several functions required throughout the monitor execution and declares as well as allocates the working memory. The functions consist of two I/O functions and evaluation functions.

The `get_input() -> Option<(`$T_{s_1}, \ldots, T_{s_\ell}$`)>` function, where $T_{s_1}, \ldots, T_{s_\ell}$ are the types of all input streams, models the receipt of input data. It produces either `None` if the execution of the system under scrutiny terminated, or `Some(v)`, where v is an ℓ-tuple containing the latest input values. Conversely, the function `emit(&(`$T_{s_{\ell+1}}, \ldots, T_{s_k}$`))` conveys a $(k-\ell)$-tuple of output values to the system.

For each stream, there are evaluation functions in several variants depending on whether they will be called in the prefix, the loop, or the postfix. The implementations differ only in the logic accessing other streams. The Lola semantics dictates that the evaluation needs to check whether the accessed value exists and to substitute it with the respective default value if needed. However, an analysis of the dependency graph reveals statically which accesses will fail. Thus, providing several implementations makes the need for such a check during runtime redundant.

The working memory is a struct aptly named `Memory`. It consists of a static array for each stream in the specification and reads as follows:

```
struct Memory { s_1: [T_{s_1}, μ(s_1)], ... , s_k: [T_{s_n}, μ(s_n)] }
```

Here, $s_1, \ldots, s_k$ are all input and output streams with types $T_1, \ldots, T_k$. The monitor allocates `Memory` once in its main function, keeps it on the stack, and grants read access to functions evaluating stream expressions.

Execution Prefix. The prefix consists of $\eta_\varphi^{\leftarrow}$ conditional blocks, each processing an input event of the system under scrutiny. If the system terminates before

the prefix concludes, the function returns true, indicating an early termination, which prompts the `main` function to initiate the postfix. Otherwise, the input is added to the working memory and, evaluation layer by evaluation layer, each output stream is evaluated in a dedicated function as can be seen in the following code snippet. For this, assume that the specification has λ^* evaluation layers, i.e., $\lambda^* = \max\{x \mid \exists s_1, \ldots, s_x \colon s_1 \leq_{eo} \cdots \leq_{eo} s_x\}$ Moreover, $\lambda_i = |\{s \mid \mathit{Layer}(s) = i\}|$ denotes the number of streams within evaluation layer $i \leq \lambda^*$. Lastly, let $s_{i,j} \leq^+_{eo} s_{i,j+1}$ with $\mathit{Layer}(s_{i,j}) = \mathit{Layer}(s_{i,j+1}) = i$.

```
let val_s_{1,1} = eval_pre_1_s_{1,1}(&Memory);
...
let val_s_{1,λ1} = eval_pre_1_s_{1,λ1}(&Memory);
memory.write_layer_1(val_s_{1,1}, ..., val_s_{1,λ1})
...
let val_s_{λ*,1} = eval_pre_s_{λ*,1}(&Memory);
...
let val_s_{λ*,λλ*} = eval_pre_s_{λ*,λλ*}(&Memory);
Memory.write_layer_λ*(val_s_{λ*,1}, ..., val_s_{λ*,λλ*});
if val_s_{t1} == true { emit(m_{t1}) }
```

Note that, as indicated in the prelude, each conditional block calls a different set of evaluation functions. This allows for a fine-grained treatment of stream accesses, improving the overall performance at the cost of greater code size. Also, the call passes a single argument to the evaluation function: an immutable reference for `Memory`. As a result, the Rust type system guarantees that the evaluation does not mutate its state. The function returns a value that is committed to `Memory` after fully evaluating the current layer. The bodies of these functions are straight-forward translations of stream expressions: each arithmetic and logical expression has a counterpart in Rust. Stream lookups access the only argument passed to the function, i.e., a read-only reference to the working memory.

The `write_layer_`i functions commit computed stream values to `Memory`. After $\mu(s)$ iterations, the memory evicts the oldest data point for stream s, thus constituting a ring buffer.

Monitor Loop The main difference between the monitor loop and the prefix is, as the name indicates, that the former consists of a loop. The loop terminates as soon as the system ceases to produce new inputs. At this point, the monitor transitions to the execution postfix.

Within the loop, the monitor proceeds just as in the prefix except that the evaluation functions are agnostic to the current iteration number. In the evaluation, all stream accesses are guaranteed to succeed rendering the evaluation free of conditionals except when the stream expression itself contains one.

Execution Postfix The structure of the execution postfix closely resembles the prefix except for two differences: The postfix does not check for the presence of new input values and calls a different set of evaluation functions, specifically tailored for the postfix iteration.

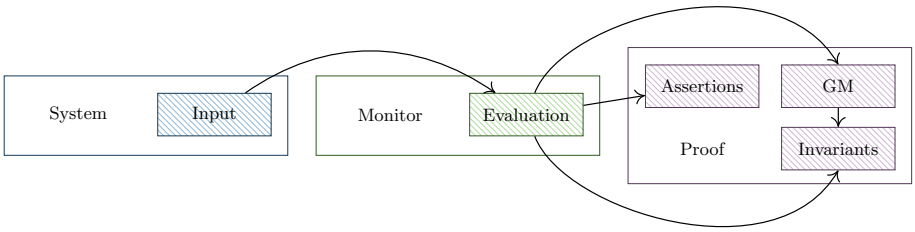


Fig. 3: Information flow between the monitor and the ghost memory.

Code Characteristics The generated code exhibits two advantageous characteristics. First, the trade-off between an increase in code size by quasi-duplicating the evaluation functions leads to an excellent performance in terms of running time. The functions require few arguments, avoid conditional statements as much as possible, and utilize memory locality. This is further emphasized by the lack of dynamic memory allocation and utilization of native datatypes. Second, the clear code structure, especially with respect to memory accesses, drastically simplifies reasoning about the correctness of the code.

4 Verification

Our goal is to prove that the verdicts produced by the monitor correspond to the formal semantics. The main challenge is that the the evaluation model of the Lola semantics refers to unbounded data sequences, disregarding any memory concerns. The implementation, however, manages the monitoring process with only a finite amount of memory. As a result, the Lola semantics may refer to data values long after they have been discarded in the implementation. Hence, the relation between the memory content and the evaluation model, and thus the correctness of the computation, is no longer apparent.

We solve this problem with the classic proof technique of introducing so-called *ghost memory*. The compilation introduces another data structure named `Ghost Memory` (GM) which is a wrapper for Rust vectors, i.e., dynamically growing sequences of data. Whenever the monitor receives or computes any data, it commits it to the GM. The GM's size thus obviously exceeds any bound, voiding the memory guarantees. However, the ghost memory's sole purpose is to aid the verification and not the monitor; information flows from the program into the GM and the proof, but remains strictly separated from the monitor execution. This allows for removing the GM after successfully verifying the correctness of the monitor without altering its behavior. Figure 3 illustrates the flow of information between the monitor and GM. Clearly, the monitor remains unaffected when removing any proof artifacts.

The correctness proof has two major obligations: proving compliance between values in the GM and the working memory, and proving the correctness of the trigger evaluations with respect to the ghost memory. These obligations are encoded as verification annotations, such that the Viper framework verifies them automatically. The compilation generates additional annotations to guide the verification process. Viper annotations fall into the following categories:

Function Contracts Annotations in front of a function `f` consist of preconditions and guarantees. Viper imposes constraints on the function caller and the function body itself. Each call to `f` is replaced by an assertion of the preconditions of `f`, prompting Viper to prove their validity, and an assumption of the guarantees. In a separate step, Viper assumes the preconditions and verifies that the guarantees hold after executing the function body. Note that the Rust type system already ensures that references passed to the function are accessible and cannot be modified or freed unless they are explicitly declared mutable.

Loop Invariants Viper analyzes while-loops similarly to functions in three steps. First, the code leading to the loop needs to satisfy the invariants. Second, Viper assumes both the loop invariant and the loop condition to hold and verifies that the invariant again holds after the execution of the body. Lastly, Viper assumes the invariant and the negation of the loop condition to hold for the code after the loop.

Inline Assertions Both loop invariants and function contracts impose implicit assertions on the code. Viper allows for supplementing them with explicit inline assertions using the Rust `assert!` macro. Usually, the macro checks an expression during runtime. Viper, however, eliminates the need for this dynamic check as it verifies the correctness statically and transforms it into an assumption for the remainder of the verification. Thus, the assertions serve a similar function as the ghost memory: they are a proof construct and do not influence the monitor per se (cf. Figure 3).

Annotation Generation. The compilation inserts annotations at several key locations. First, as an example for function annotations, consider a function that retrieves a value of the stream s from the working memory. The function takes the relative index of the retrieved value as single argument, i.e., an index of 1 accesses the second to newest value. The annotation requires that the index must not exceed the memory reserved for s. Syntactically, this results in the following annotation in front of the function head: `#[requires="index < μ(s)"]`. Moreover, the function needs to guarantee that the return value corresponds to the respective value stored in `Memory`. This is expressed by the annotation `#[ensures="index == i ==> result == self.s[i]"]` for each $i \leq \mu(s)$. The remaining function annotations follow a similar pattern, i.e., they require valid arguments, and ensure correct outputs as well as the absence of undesired changes. Note that the ghost memory is essentially a wrapper for Rust vectors as they represent a growing list of values. Thus, functions concerning the ghost

memory carry the standard annotation ensuring correctness of the vector as presented in the Viper examples.[2]

Second, the loop has several entry checks that are expressed as inline assertions. These ensure that the iteration count is $\eta_\varphi^{\leftarrow}$ and that the length of the ghost memory for a stream s is $\eta_\varphi^{\leftarrow} - \Delta(s)$. This is necessary because the loop invariant asserts equivalence between an excerpt of the ghost memory and the working memory. While the existence of all accessed values in the working memory is guaranteed due to the static allocation, the GM grows dynamically. Hence, the compilation adds the entry checks.

In terms of memory equivalence, it remains to be shown that all values in the working memory correspond to the respective entry in the ghost memory. Formally, let m be the working memory and let g be the ghost memory where index 0 marks the latest value. Furthermore, let η be the current iteration count. Then, the invariant checks:

$$\forall s\colon \forall i\colon (0 \leq i < \mu(s) \implies m_s[i] = g_s[i]). \tag{1}$$

At loop entry, $\mu(s) = \eta_\varphi^{\leftarrow} - \Delta(s) = \eta - \Delta(s)$ is the number of iterations in which a value for s was computed. In each further iteration of the loop, the invariant checks that the former $\mu(s) - 1$ entries remained the same and that the new values in the ghost memory g and the working memory m are equal. The first of these checks is not strictly necessary for the proof because it immediately follows from the function contracts of the helper functions. However, after completing one loop iteration, Viper deletes prior knowledge about all variables that were mutated in the loop. Further reasoning about these variables is thus solely based on the loop invariants.

To express Equation (1) in Viper, the compilation needs to statically resolve the universal quantification over the streams. Thus, for each stream s, the compilation generates the annotation `#[invariant="forall i: usize :: (0 <= i && i < μ(s)) ==> mem.get_s(i) == gm.get_s(iter - 1)"]`, where `iter` is a variable denoting the current iteration, `mem` is the working memory, and `gm` is the ghost memory. Viper is able to handle the remaining universal quantification over i. However, the compilation reduces the verification effort further by unrolling it. This is possible since the memory requirement $\mu(s)$ of a stream s is determined statically.

Lastly, the compilation introduces inline assertions after the evaluation of stream expressions, i.e., in the prefix, postfix, and loop body. These annotations show that computed values are correct when assuming that the values retrieved from the working memory are correct as well. This argument is well-founded because the compilation substitutes failing stream accesses by their respective default values. Thus, any value retrieved from `Memory` was computed in an earlier iteration or layer and therefore proven correct by Viper.

[2] See e.g. the verified solution for the Knapsack Problem: `https://github.com/viperproject/prusti-dev/blob/master/prusti/tests/verify/pass/rosetta/Knapsack_Problem.rs`.

It only remains to be shown that the stream expression is properly evaluated. Expressions consist of arithmetic or logical functions, constants, and stream accesses. The former two can be trivially represented in Viper. Since the memory is assumed to be correct and failing accesses are substituted by constants when possible, accesses also translate naturally into Viper.

Conclusion. The validity of the assertions after the evaluation logic shows that newly computed values are correct if the values in the working memory m and the ghost memory g coincide. This fact is guaranteed by the loop invariant. Furthermore, the inductive argument of the loop invariants allows us to conclude that, if m were to never discard values, $m_s[i] = g_s[i]$ for all streams s and $i \leq \eta$. Thus, m is a real subsequence of g, which is a perfect reflection of the evaluation model. As a result, any trigger violation detected by the monitor realization corresponds to a violation in the evaluation model for the same sequence of input values; The realization is verifiably correct.

5 Concurrent Evaluation

Evaluating independent streams concurrently can significantly improve the performance of the monitor. In the following, we devise an analysis of Lola specifications that enables safe parallelization. We observe two characteristics of Lola: the computation of a stream expression can only *read* the memory of other streams, and inter-stream dependencies are determined statically. The evaluation layers are a manifestation of the second observation. They group streams which are incomparable according to the evaluation order. Combined with the first observation, we can conclude that all streams within one layer may be computed in parallel. Thus, the compilation spawns a new thread for each stream within the layer with read access to the global memory. We add annotations to the code that enable Viper to verify that the parallel execution remains correct.

The compilation capitalizes on Rust's concurrency capabilities by evaluating different output streams in parallel. A major advantage of Rust is that its ownership model enforces a strict separation of mutable and immutable data. Any data point has exactly one owner who can transfer ownership for good or let other functions borrow the data. Borrowing data is again either mutable or immutable. If a function mutably borrows data, no other function, including the owner, can read or write this data. Similarly, if a function immutably borrows data, other functions and the owner can only read it. A consequence of this fine-grained access management with static enforcement is that enabling concurrency becomes rather easy when compared to languages like C.

Enabling the concurrent evaluation requires slight changes in the code generation. First, evaluation functions are annotated with `#[pure]`. This indicates that a function mutates nothing but its local stack portion. For the evaluation logic, the compiler still proceeds layer by layer, opening a *scope* for each of them. In the scope, it generates code following the total evaluation order $\leq_{eo}^{+}$. However, rather than calling the respective evaluation functions directly, the parallelized

version spawns a thread for each stream and starts the evaluation inside it. Assume $s_1, \ldots, s_n$ constitute a single layer of a specification. The evaluation then looks as follows:

```
let (v_1, ..., v_n) = crossbeam::scope(|scope| {
  let handle_s1 = scope.spawn(move |_| {
      eval_s1(&memory)
  });
  ...
  let handle_sn = scope.spawn(move |_| {
      eval_sn(&memory)
  });
  (handle_s1.join().unwrap(), ..., handle_sn.join().unwrap())
}).unwrap()
```

Note that the code snippet uses the Rust crate crossbeam, a standard concurrency library. A similar result can be achieved without external code by moving the global memory to the heap and using the standard Rust thread logic.[3]

The correctness of this approach is an immediate consequence of the correctness of the evaluation order and memory locality of streams. In particular, the independence of streams within the same evaluation layer and the pureness of the functions are crucial. The latter ensures that the function does not mutate anything outside of its local stack. The former ensures that using pure evaluation functions within the same layer is indeed possible. Thus, the order of execution cannot change the outcome of the function, enabling the concurrent evaluation.

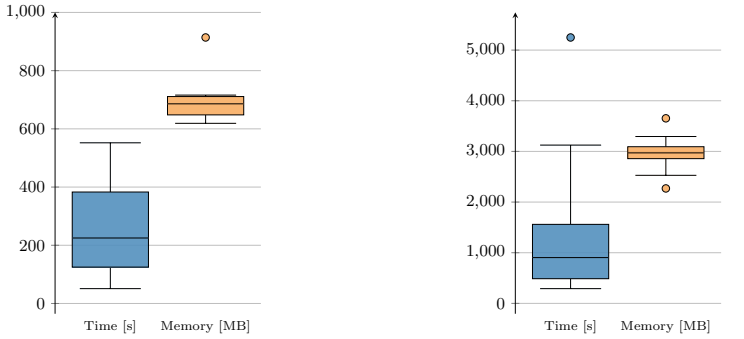
Note that spawning a thread for each stream evaluation is a double-edged sword. While it can drastically reduce the monitor's latency, each spawn induces a constant overhead. Thus, reducing the number of spawns while increasing the parallel computation time maximizes the gain. Consequently, the monitor benefits stronger from the parallel evaluation when its dependency graph is wide, enabling several cores to compute in parallel. Similarly, specifications with large stream expressions benefit from the multi-threading because the share of parallel computations increases. This lowers the relative impact of the constant thread-spawning overhead.

6 Experimental Evaluation

The implementation of the compiler is based on the RTLola[4] framework written in Rust. The code verification uses the Rust-frontend of the Viper framework called Prusti [2]. Prusti translates a Rust program into the Viper intermediate verification language, followed by a translation into an SMT model, which

[3] On a technical note: Rust's type system requires the programmer to guarantee that the global memory will not be dropped until all threads terminate. Thus, the memory needs to be wrapped into an *Atomically Reference Counted (Arc)* pointer. This has two disadvantages: all accesses to memory require generally slower heap access and the evaluation suffers from the overhead accompanying atomic reference counting.

[4] `http://www.rtlola.org/`



(a) Altitude Monitor (b) Network Traffic Monitor

Fig. 4: Results of 20 runs in terms of running time (blue, in seconds) and memory consumption (orange, in MB) for the verification of the annotated Rust code of the specification, where the altitude of a drone is monitored (cf. Listing 1.1), and the network traffic monitor specification.

is checked by the Z3 [18] SMT solver. Thus, our toolchain enables completely automatic proof checking.

The experiments were conducted on a machine with a 3.1GHz Dual-Core Intel i5 processor with 16GB of RAM. The artifacts for the evaluation are available on github.[5] In all experiments, the compilation itself has a negligible running time of under ten milliseconds and memory consumption of less than 4MB, mainly due to the RTLola frontend. As expected, the verification of the annotated rust code using Prusti and the Viper toolkit takes significant time and memory. While the translation into the SMT model is deterministic and can be parallelized, the verification with Z3 exhibits generally high and unpredictable running time.

We discuss the results of compiling and verifying three Lola specification of varying size. The process works flawlessly on two of them while the third one occasionally runs into timeouts and inconclusive verification results.

First, we consider the specification from Listing 1.1, where the altitude of a drone is monitored. The results in terms of both running time and memory consumption for 20 runs are depicted in Figure 4a. Note that the y-axis displays both the running time in seconds (left plot) and the memory consumption in megabytes (right plot). The plot shows that the running time never exceeds 600s with a median of 225s. The memory consumption is significantly more stable ranging between 648 and 711MB with one outlier (914MB).

While the first specification was short and illustrative, the second one is more practically relevant. The specification monitors the network traffic of a server based on the source and destination IP of requests, TCP flags, and the length of the payload [4]. The specification counts the number of incoming connections and computes the workload, i.e., the number of bytes received over push requests.

[5] `https://github.com/reactive-systems/Lola2RustArtifact`

```
input src, dst, length: Int32
input fin: Bool, push: Bool, syn: Bool
constant server: Int32 := ...

output count : Int32 := if count[-1,0] > 201 then 0 else count[-1,0] + 1
output receiver : Int32 := if dst=server then receiver[-2,0] + 2 else
    if count > 200 then 0 else receiver[-1,0]
trigger receiver > 50 "Many incoming connections."

output received : Int32 := if dst=server ∧ push then 0 else length
output workload : Int32 := if count > 200 then workload[-1,0] + 1 else 0
trigger workload > 25 "Workload too high."
output opened : Int32 := opened[-1,0] + int(dst=server ∧ syn)
output closed : Int32 := closed[-1,0] + int(dst=server ∧ fin)
trigger opened - closed < 0 "Closed more connections than have been opened."
```

Listing 1.3: Lola specification for monitoring network traffic

If any of these numbers exceeds a threshold, the specification raises an alarm. Moreover, it keeps track of the number of open connections. A trigger indicates when the the server attempts to close a connection even though none is open. The full specification can be found in Listing 1.3. Figure 4b depicts the results both in terms of running time and memory consumption for 20 runs. Again, the y-axis represents both running time in seconds and memory consumption in megabytes. The increase in resource consumption clearly reflects the increase in complexity and size of the input specification. While the longest run took nearly 90min, most of the runs took less than 25min with a median of roughly 15min. Like before, the memory consumption is relatively stable ranging around 3GB.

Lastly, we considered a Lola specifications that shows the limitations of our approach. It detects different flight phases of a drone and raises an alarm if actual velocity and a reference velocity provided by the flight controller deviate strongly. The specification is based on a Lola specification for flight phase detection shown in Listing 1.4.

After a successful compilation, the verification was able to reveal potential arithmetic errors in the original specification [1]. The errors arose from division in which the denominator was an input stream access. The resulting value is not necessarily non-zero, so Viper reported that the respective annotation cannot be verified. Hence, our approach is able to detect flaws in specifications stemming from implicit assumptions on the system. These assumptions may not hold during runtime, causing the monitor to fail.

Thus, we modified the flight phase detection specification to work without division. Yet, only four of our runs terminated successfully. The running time varies between 6 and 16min and the memory consumption between 1.38GB and 1.66GB. The successful runs show that our approach is able to verify monitor realizations of large and arithmetically challenging Lola specifications. However, two runs did not terminate within three hours. The reason lies within the underlying SMT solver: an unfavorable path choice in the solving procedure can result in extended running times. Additionally, for four runs, the verification reported that some assertions might not hold or crashed internally. While restarting the

```
input time_s, time_micros, velo_x, velo_y, velo_r_x, velo_r_y: Int32

output time := time_s + time_micros / 1000000
output count := count[-1,0] + 1
output frequency := 1 / (time - time[-1,0])
output freq_sum := frequency + freq_sum[-1,0]
output freq_avg := freq_sum / count
output velo : Int32 := vel_x*vel_x + vel_y*vel_y
output velo_max : Int32 := if res_max[-1,false] then velo
    else max(velo_max[-1,0], velo)
output velo_min : Int32 := if res_max[-1,false] then velo
    else min(velo_min[-1,0], velo)
output res_max: Bool := (velo_max - velo_min) > 1
output unchanged: Int32 := if res_max[-1,false] then 0 else unchanged[-1,0] + 1
output velo_dev : Int32 := velo_r_x - velo_x + velo_r_y - velo_y
output worst_dev: Int32 := if unchanged > 15 then velo_dev else max(velo_dev,
     worst_dev[-1,-10])

trigger freq_avg < 10 "Low input frequency."
trigger velo_dev > 10 "Deviation between velocities too high."
trigger worst_dev > 20 "Worst velocity deviation too high."
```

Listing 1.4: Lola specification for flight phase detection

verification procedure can lead to finding a successful run, the incident shows the reliance of our approach on external tools. Hence, the applicability increases with advances in research on automated proof checking of annotated code. This constitutes another reason for the continued development of valuable tools like Prusti and the Viper framework.

6.1 Performance of Generated Monitors

As expected, the compiled monitors exhibit superior running time when compared against the RTLola [11] interpreter. The comparison is based on randomly generated input data for the Altimeter[6] and Network Traffic Monitor. For the first specification, the interpreter required 438ns per event on average out of 10 runs, whereas the compiled version took 6.2ns. The second, more involved specification shows similar results: 1.535µs for the interpreter and 63.4ns for the compiled version.

7 Related Work

The development of a verifying compiler was identified by Tony Hoare as a grand challenge for computing research [16]. Milestone results have been the concept of proof-carrying code (PCC) [20] and the technique of checking the result of each compilation instead of verifying the compiler's source code [21]. PCC architectures [7] and certifying compilers [8] exist for general purpose languages like Java. A variation of the PCC, abstraction-carrying code [5,15] was developed

[6] The specification was adapted to be compliant with RTLola: rather than accessing the input with a future offset, the specification used a negative offset of -2.

for constraint logic programs, where a fixpoint of an abstract interpretation serves as certificate for invariants. This enables automatic proof generation.

In this paper, we present a verifying compiler for the stream-based monitoring language Lola. Compared to general programming languages, the compilation of monitoring languages is still a young research topic. Some work has focused on compiling specifications immediately into executable code. Rmor [14], for instance, generates constant memory C code.

Similarly, a Copilot [23] specification can be compiled into a constant memory and constant time C realization. The Copilot toolchain [24] enables the verification of the monitor using the CBMC model checker [6]. As opposed to our approach, their verification is limited to the absence of various arithmetic errors, lacking functional correctness. While CBMC can verify arbitrary inline assertions, Copilot does not generate them. Note that, in contrast to Lola, Copilot can express real-time properties.

RTLola [12, 27], on the other hand, is a real-time, asynchronous extension of Lola, for which a compilation into the hardware description language VHDL exists [4]. The VHDL code contains traceability annotations [**?**] and can then be realized on an FPGA. Similarly, Pellizzoni et al. [22] and Schumann et al. [17, 26] realize their runtime monitors on FPGAs, yet without verification or traceability.

Rather than using a dedicated specification language, there are several logics for which verified compilers exist. Differential dynamic logic [**?**], for example, was specifically designed to capture the complex hybrid dynamics of cyber-physical systems. The ModelPlex [**?**] framework translates such a specification into several verified components monitoring both the environment with respect to the assumed model and the controller decisions. Lastly, there is work on verifying monitors for metric first-order temporal [**?**] and dynamic logic [**?**].

8 Conclusion

We have presented a compilation of Lola specifications into Rust code. Using Rust as the compilation target has the advantage that the executables are highly performant and can be used directly on many embedded platforms. The generated code contains annotations that enable the verification of the code using the Viper framework. With the guiding assertions in the code, as well as function contracts and loop invariants, Viper can verify monitors even for large specifications.

Our results are promising and encourage further research in this direction, such as compiling more expressive dialects of Lola such as RTLola [12, 27]. RTLola extends Lola with real-time aspects and can handle asynchronous inputs. The added functionality is highly relevant in the design of monitors for cyber-physical systems [3, 11]. While generating verifiable RTLola monitors in Rust will require additional effort, such an extension would further improve the practical applicability of our approach.

References

1. Adolf, F., Faymonville, P., Finkbeiner, B., Schirmer, S., Torens, C.: Stream Runtime Monitoring on UAS. In: RV 2017. LNCS, vol. 10548, pp. 33–49. Springer (2017). https://doi.org/10.1007/978-3-319-67531-2_3
2. Albert, E., Puebla, G., Hermenegildo, M.V.: Abstraction-Carrying Code. In: 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2004). pp. 380–397. No. 3452 in LNAI, Springer-Verlag (March 2005)
3. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust Types for Modular Specification and Verification. Proc. ACM Program. Lang. **3**(OOPSLA), 147:1–147:30 (2019). https://doi.org/10.1145/3360573
4. Basin, D.A., Dardinier, T., Heimes, L., Krstic, S., Raszyk, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12166, pp. 432–453. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_25
5. Baumeister: Tracing Correctness: A Practical Approach to Traceable Runtime Monitoring. Master thesis, Saarland University (2020)
6. Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., Torens, C.: RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft. In: CAV 2020. LNCS, vol. 12225, pp. 28–39. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_3
7. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: FPGA Stream-Monitoring of Real-time Properties. ACM Trans. Embedded Comput. Syst. **18**(5s), 88:1–88:24 (2019). https://doi.org/10.1145/3358220
8. Besson, F., Jensen, T.P., Pichardie, D.: Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. Theor. Comput. Sci. **364**(3), 273–291 (2006). https://doi.org/10.1016/j.tcs.2006.08.012
9. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
10. Colby, C., Lee, P., Necula, G.C.: A Proof-Carrying Code Architecture for Java. In: CAV 2000. LNCS, vol. 1855, pp. 557–560. Springer (2000). https://doi.org/10.1007/10722167_44
11. Colby, C., Lee, P., Necula, G.C., Blau, F., Plesko, M., Cline, K.: A Certifying Compiler for Java. In: PLDI 2000. pp. 95–107. ACM (2000). https://doi.org/10.1145/349299.349315
12. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: Temporal Stream-Based Specification Language. In: SBMF 2018. LNCS, vol. 11254, pp. 144–162. Springer (2018). https://doi.org/10.1007/978-3-030-03044-5_10
13. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: Runtime Monitoring of Synchronous Systems. In: TIME 2005. pp. 166–174. IEEE Computer Society Press (June 2005)
14. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: StreamLAB: Stream-based Monitoring of Cyber-Physical Systems. In: CAV 2019. LNCS, vol. 11561, pp. 421–431. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_24
15. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time Stream-based Monitoring. CoRR **abs/1711.03829** (2017), `http://arxiv.org/abs/1711.03829`

16. Gorostiaga, F., Sánchez, C.: Striver: Stream Runtime Verification for Real-Time Event-Streams. In: RV 2018. LNCS, vol. 11237, pp. 282–298. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_16
17. Havelund, K.: Runtime Verification of C Programs. In: FATES 2008. LNCS, vol. 5047, pp. 7–22. Springer (2008). https://doi.org/10.1007/978-3-540-68524-1_3
18. Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. J. ACM **50**(1), 63–69 (2003). https://doi.org/10.1145/602382.602403
19. Mitsch, S., Platzer, A.: Modelplex: verified runtime validation of verified cyber-physical system models. Formal Methods Syst. Des. **49**(1-2), 33–74 (2016). https://doi.org/10.1007/s10703-016-0241-z
20. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. Formal Methods Syst. Des. **51**(1), 31–61 (2017). https://doi.org/10.1007/s10703-017-0275-x
21. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
22. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2
23. Necula, G.C.: Proof-Carrying Code. In: POPL 1997. pp. 106–119. ACM Press (1997). https://doi.org/10.1145/263699.263712
24. Necula, G.C., Lee, P.: The Design and Implementation of a Certifying Compiler. In: PLDI 1998. pp. 333–344. ACM (1998). https://doi.org/10.1145/277650.277752
25. Pellizzoni, R., Meredith, P.O., Caccamo, M., Rosu, G.: Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems. In: RTSS 2008. pp. 481–491. IEEE Computer Society (2008). https://doi.org/10.1109/RTSS.2008.43
26. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A Hard Real-Time Runtime Monitor. In: RV 2010. LNCS, vol. 6418, pp. 345–359. Springer (2010). https://doi.org/10.1007/978-3-642-16612-9_26
27. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Copilot: Monitoring Embedded Systems. ISSE **9**(4), 235–255 (2013). https://doi.org/10.1007/s11334-013-0223-x
28. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reasoning **41**(2), 143–189 (2008). https://doi.org/10.1007/s10817-008-9103-8
29. Pnueli, A.: The Temporal Logic of Programs. In: Annual Symposium on Foundations of Computer Science, 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
30. Schneider, J., Basin, D.A., Krstic, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 310–328. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_18
31. Schumann, J., Moosbrugger, P., Rozier, K.Y.: R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. In: RV 2015. LNCS, vol. 9333, pp. 233–249. Springer (2015). https://doi.org/10.1007/978-3-319-23820-3_15
32. Schwenger, M.: Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS. Master thesis, Saarland University (2019)