SAARLAND UNIVERSITY
Faculty of Mathematics and Computer Science
Department of Computer Science

Bachelor's Thesis

Verifiable Runtime Monitor Generation for Lola Specifications

by
Stefan Oswald

submitted
February 18, 2020

Supervisor:
Prof. Bernd Finkbeiner, Ph.D.

Advisor:
Noemi Passing, B. Sc.
Maximilian Schwenger, B. Sc.

Reviewer:
Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr. Sebastian Hack

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,______________________ ______________________________

(Datum/Date) (Unterschrift/Signature)

Abstract

Runtime monitoring nowadays is commonly used to verify program behavior whenever static verification fails. This can happen either due to missing information about input data or because of insufficient analysis precision. A monitor supervises a program with respect to a specification and reports program behavior violating the specification. So far, there are no formal guarantees that the monitor reflects the semantics of the underlying specification correctly. It is important to verify the monitor's behavior to ensure the soundness and completeness of all specification violation reports.

In this thesis, we introduce a translation algorithm to transform a Lola specification into a Viper program. Lola is a specification language for formalizing the correct behavior of a synchronous system. The language Viper provides a static verification toolchain and is used to prove the functional correctness of programs. The generated program models the runtime monitor and is annotated with verification conditions. With this, we prove the correctness of the monitor and examine the behavior of a Lola monitor in a concurrent context. We determine all computations within the specification that can be processed in parallel and show the interleaving invariance and correctness of the monitors computations.

Acknowledgements

I would like to thank Prof. Bernd Finkbeiner for supervising me during this thesis and including me in the enjoyable Reactive Systems group. Thanks to Kallistos Weis for proofreading, Jessica Schmidt, Florian Kohn and all people in the lab for mental and motivational support. I want to thank my advisors Noemi and Maximilian for the excellent support, feedback, and crêpes. Thanks to my family, friends and especially my brother Jan for the assistance I needed during this work. Last but not least, thanks to Prof. Sebastian Hack for reviewing this thesis.

Contents

1 Introduction **1**

2 Background **3**
2.1 Lola . . . 3
2.1.1 Formal Specification . . . 4
2.1.2 Dependency Graph . . . 5
2.2 Viper . . . 7

3 Translation **11**
3.1 Algorithm Overview . . . 11
3.2 Dependency Graph Analysis . . . 11
3.3 Pre-/Postfix Construction . . . 14
3.4 Loop Generation . . . 15

4 Verification **19**
4.1 Trigger Evaluation . . . 19
4.1.1 Ghost Memory . . . 19
4.1.2 Invariant Generation . . . 20
4.2 Concurrency . . . 21

5 Evaluation **27**
5.1 Division and Rationals . . . 27
5.2 Expression Inlining Depth . . . 28
5.3 Runtime and Memory . . . 29

6 Related Work **37**

7 Conclusion **39**
7.1 Future Work . . . 39

Bibliography **43**

1. Introduction

Correctness of a program can be shown by exhaustive execution for all program paths and input values. This is infeasible for all non-trivial systems. The best way to guarantee the correctness of a system is to formally prove that the behavior fulfills the system specification. Due to bad scalability and incomplete information about the input data, this is not always achievable. When formal correctness cannot be proven, testing is used to obtain confidence in the program's behavior. Testing is only applicable during production and is not able to completely cover big systems. Consider big autonomous systems that are used in safety-critical areas like self-driving cars and medical tools. These systems, typically well tested, take place in modern daily life and often fulfill their duty unnoticed. Historical incidents emphasize the devastation of unproven software. Software failures in the Therac-25[1] radiation therapy machine killed multiple patients due to radiation poisoning. Another example of critical software failure was a lethal car crash in 2018 by the Tesla autopilot[2], where a bug in the software failed to recognize a white truck in front of the bright sky. Accidents like this increase the demand for formally proven software. To ensure proper program behavior after deployment, an additional method is required.
Runtime monitoring can be used when other forms of verification cannot prove the correctness of a system. It checks that the current system trace satisfies a formal specification during the execution. A runtime monitor observes the system and checks if the obtained data violates the specification. Reported errors can be fed back to the system to possibly correct false behavior. While testing compares the result of a system for a concrete input with an expected result, a runtime monitor observes the system at every point of the execution. The monitor verifies the current behavior of the system, especially in the context of unknown scenarios, not considered by testing and not encountered previously.

When the behavior of safety-critical systems is supervised through runtime monitoring, the correctness of the system depends on the correctness of the monitor

[1]https://www.bugsnag.com/blog/bug-day-race-condition-therac-25
[2]https://www.wired.com/story/tesla-autopilot-self-driving-crash-california/

supervising the system. For most specification languages, besides Copilot [23], there are no formal guarantees that the monitor's semantics correctly reflects the specification. Without formal correctness of the monitor, the problem of software flaws is not solved but rather moved to the monitor.

In this thesis, we introduce a translation algorithm that provides these formal guarantees and proves the correctness of a monitor. We describe the specification language Lola [8] and its stream-based computation model to specify correct system behavior. A Lola specification allows the formulation of correct behavior through numerical expressions. We use this specification to generate a Viper [22] program that models the monitor supervising the system. Viper is an intermediate language connected to a set of tools using abstract interpretation and an SMT-solver to prove the correctness of the input program with respect to given verification conditions. We extend the model of the monitor with verification statements to prove that the monitor semantics is equivalent to the formal description. To ensure correct behavior in a concurrent context, we encapsulate the evaluation functions for stream computations and introduce an evaluation order and a layer classification. With stream layers and the permission model of Viper, we prove the absence of data races and non-deterministic behavior for all stream computations.
A formally proven monitor can ensure the correctness of the system and safe lives, as can be seen in the Therac-25 incident .

2. Background

2.1 Lola

Lola is a specification language that provides a simple but expressive grammar to formulate a correctness description for synchronous systems. A runtime monitor uses this specification and validates that the supervised system fulfills the specification or reports violations to the user. Lola models runtime monitoring through stream computations. Intuitively, a *stream* can be described as a continuously growing array of values. Each time a new input value is provided by the system, the value is appended to the end of the corresponding stream. Every time this happens can be seen as a computation iteration and in each iteration, new values for dependent streams are computed and appended. The newest value of a stream is given by an expression which may depend on other stream values. Formally, streams can be classified into two kinds of streams, input streams and output streams. *Input streams* capture the observed behavior of the system, *output streams* derive values from input and other output streams. Stream values can be accessed by another stream with a given discrete offset. Figure 2.1 shows an example with a single input stream and two output streams and their dependencies for a single value computation. A system is called *synchronous* when all specified streams are extended at the same time. Similarly, a system is *asynchronous* if the streams have independent frequencies.

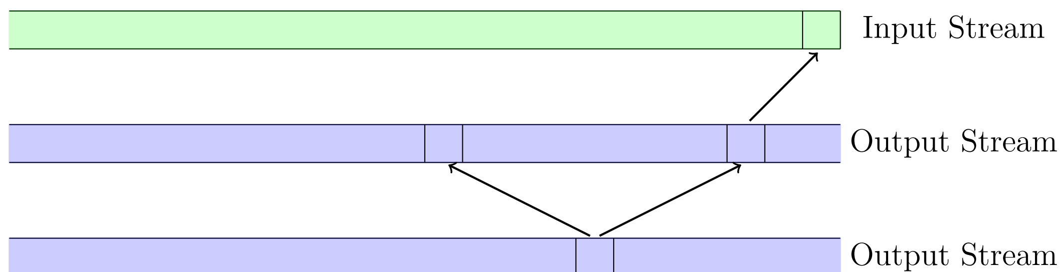


Figure 2.1: Visualization of two output streams, blue, and an input stream, green. The arrows show positional dependencies i.e. which values are used.

2.1.1 Formal Specification

A Lola specification is defined by a set of equations over typed *stream variables* of the following form:

$$
\begin{array}{lcl}
\textit{input } i_1 : T_1 & & \\
\quad\vdots & & \\
\textit{input } i_m : T_m & & \\
\textit{output } o_1 : T_{m+1} & := & e_1(i_1, \dots, i_m, o_1, \dots, o_n) \\
\quad\vdots & & \qquad\qquad\vdots \\
\textit{output } o_n : T_{m+n} & := & e_n(i_1, \dots, i_m, o_1, \dots, o_n) \\
\textit{trigger } \psi_1 & & \\
\quad\vdots & & \\
\textit{trigger } \psi_l & &
\end{array}
$$

where $i_1, \dots, i_m$ are called *independent stream variables* or *input streams*, $o_1, \dots, o_n$ are called *dependent stream variables* or *output streams* and $e_1, \dots, e_n$ are *stream expressions* over $i_1, \dots, i_m$ and $o_1, \dots, o_n$. $T_1, \dots, T_{m+n}$ are the corresponding types for each stream. Within a Lola specification, the user can declare some output streams as *triggers*. A trigger is an output stream of boolean type which reports an error to the user if its trigger expression ψ_i evaluates to true. Triggers are defined by the *trigger* keyword and a *trigger expression* ψ.

A stream expression is defined recursively as follows:

- If c is a constant of type T, then c is an *atomic stream expression* of type T.
- If s is a stream variable of type T, then s is an *atomic stream expression* of type T.
- Let $f : T_1 \times T_2 \times \ldots \times T_k \mapsto T$ be a k-ary operator. If for $1 \leq i \leq k$, e_i is a stream expression of type T_i, then $f(e_1, \dots, e_k)$ is a stream expression of type T.
- If e_1 and e_2 are stream expressions over the same type T and b is a expression of type *bool*, $ite(b, e_1, e_2)$ is a expression of type T matching an *if-then-else* expression
- If e is a stream expression of type T, c is a constant of type T, and i is an integer, then $e[i, c]$ is a stream expression of type T. The expression refers to the value of the stream e accessed with an offset of i positions of the current position or to the given constant c if the value does not exist.

Figure 2.2 shows an example specification. In this specification, the monitor gets two different inputs from the monitored system; the current data flow amount

```
input flow: Int
input signal: Bool
output sum: Int := flow[1,0] + flow + flow[-1,0]
output expects: Bool := sum < 5 => signal[2,false]
trigger !expects "flow below threshold without signal"
```

Figure 2.2: A Lola example specification. The first output stream sums up three input values. If this value is to low and no signal is given, the specification would be violated and an error is reported.

and a binary signal. The monitor computes the `sum` by adding the current, the next and the previous value of the input stream. Then, it checks if this value falls below a given threshold of 5 and expects an input signal in two steps. Otherwise, the trigger reports an error.

The semantics of a Lola specification is defined by its *evaluation model*. Given a specification ϕ with input variables $i_1, \ldots, i_n$ and output variables $o_1, \ldots, o_m$, each with corresponding type T_i for $0 \leq i \leq n+m$. Let $\tau_1, \ldots, \tau_m$ be streams of length $N+1$. The tuple $S = \langle \sigma_1, \ldots \sigma_n \rangle$ is called an evaluation model if each stream equation $o_i := e_i(i_1, \ldots, i_m, o_1, \ldots, o_n)$ and S fulfill the associated equation:

$$\sigma_i(j) = val(e_i)(j) \quad \text{for } 0 \leq j \leq N$$

The function *val* is defined recursively:

$$\begin{array}{lcc}
val(c)(j) & = & c \\
val(i_i)(j) & = & \tau_i(j) \\
val(o_i)(j) & = & \sigma_i(j) \\
val(f(e_1, \ldots, e_k))(j) & = & f(val(e_1)(j), \ldots, val(e_k)(j)) \\
val(ite(b, e_1, e_2)) & = & \mathit{if}\ val(b)(j)\ \mathit{then}\ val(e_1)(j)\ \mathit{else}\ val(e_2)(j) \\
val(e[k,c](j)) & = & \begin{cases} val(e)(j+k) & 0 \leq j+k \leq N \\ c & \mathit{otherwise} \end{cases}
\end{array}$$

$\tau_i(j)$ is an access to the input stream τ_i at position j, equally $\sigma_i(j)$ returns the value of σ_i at position j. Intuitively, $val(e_i)(j)$ evaluates the mathematical stream expression e_i, accesses streams at position j and returns a value with the type T_i.

2.1.2 Dependency Graph

A Lola specification induces a *dependency graph* (DG) through all stream accesses. A dependency graph is a weighted multi-graph $G = \langle V, E \rangle$ with ver-

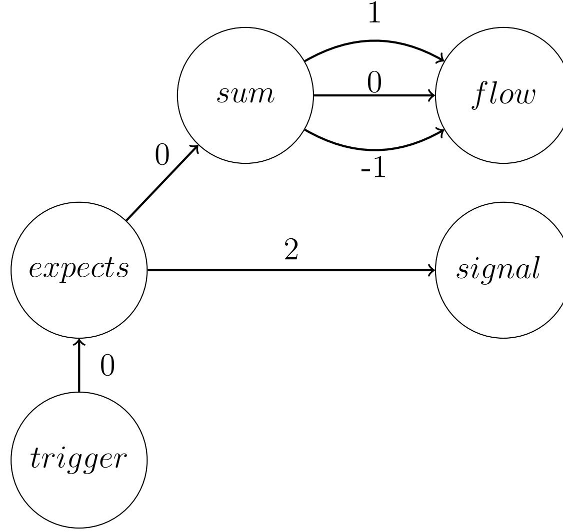


Figure 2.3: Dependency Graph for the flow example

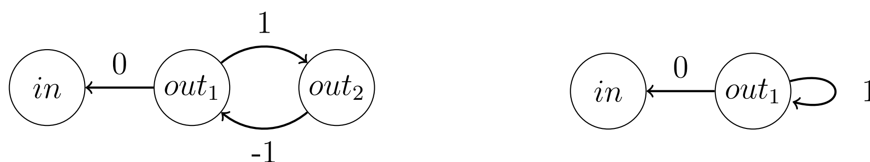
tices V containing all streams and E containing an edge $e = (s_i, s_j, w)$ of weight w from s_i to s_j iff the stream expression of s_i contains a stream access to s_j with offset w. The DG records that s_i depends on a particular position of s_j. Note that vertices representing input streams have no outgoing edges and trigger output streams have no incoming edges. Figure 2.3 shows the dependency graph for the specification of Figure 2.2. The DG is used to infer the maximum memory requirement needed for each stream and the earliest moment a stream value can be evaluated.

To classify specifications, we introduce the definition of *efficiently-monitorable* and *well-formed* specifications.

Definition 2.1.1. A *path* p in the DG is a sequence of vertices $p := v_1, \dots, v_{k+1}$ for $k \geq 1$ and edges $e_1, \dots, e_k$ such that $e_i : (v_i, v_{i+1}, w_i)$. A path is called a *cycle* if $v_1 = v_{k+1}$. The *weight* of a path is defined as the sum over all edge weights.

Definition 2.1.2. If the dependency graph contains no cycle with weight 0, the specification is called *well-formed*.

Definition 2.1.3. If the DG contains no cycle with positive weight, the specification is called *efficiently monitorable* as the total memory consumption is constant.



(a) DG of a not well-formed specification

(b) DG of a not efficiently monitorable specification

Figure 2.4: Dependency Graphs for specifications violating Definition 2.1.2(left) and 2.1.3(right)

Figure 2.4a shows the dependency graph of this not well-formed specification:

```
input in: Int
output out1: Int := in + out2[1,1]
output out2: Int := out1[-1,-1] - 1
```

As the graph contains the cycle of $out_1 \xrightarrow{1} out_2 \xrightarrow{\text{-1}} out_1$ with a weight of 0, there is no unique evaluation model. Intuitively, this specification cannot be evaluated because to compute the value of out_1 at iteration i we have to know the value of out_2 at iteration $i+1$, but out_2 at iteration $i+1$ depends on out_1 at iteration $(i+1)-1=i$. This results in a circular dependency and we cannot compute a value for either of the two streams.
Next, consider the DG in Figure 2.4b and the following specification which contains a cycle of positive weight:

```
input in: Bool
output out1: Bool := in || out1[1,false]
```

Specifications that are not efficiently monitorable can have a unique evaluation model but require unbounded memory to evaluate. To evaluate the stream at position i, we first have to consider position $i+1$, which depends on position $i+2$ etc. until the default value is used. All stream entries can be evaluated when the stream ends. When the last position can be resolved, the result can be back-propagated to all other entries.

2.2 Viper

Viper is an imperative programming language and a suite of tools that uses a separation logic-based intermediate language. It is used to verify programs modeled in Viper. The Viper toolchain utilizes abstract interpretation to enhance the program's state information. This state information is analyzed with

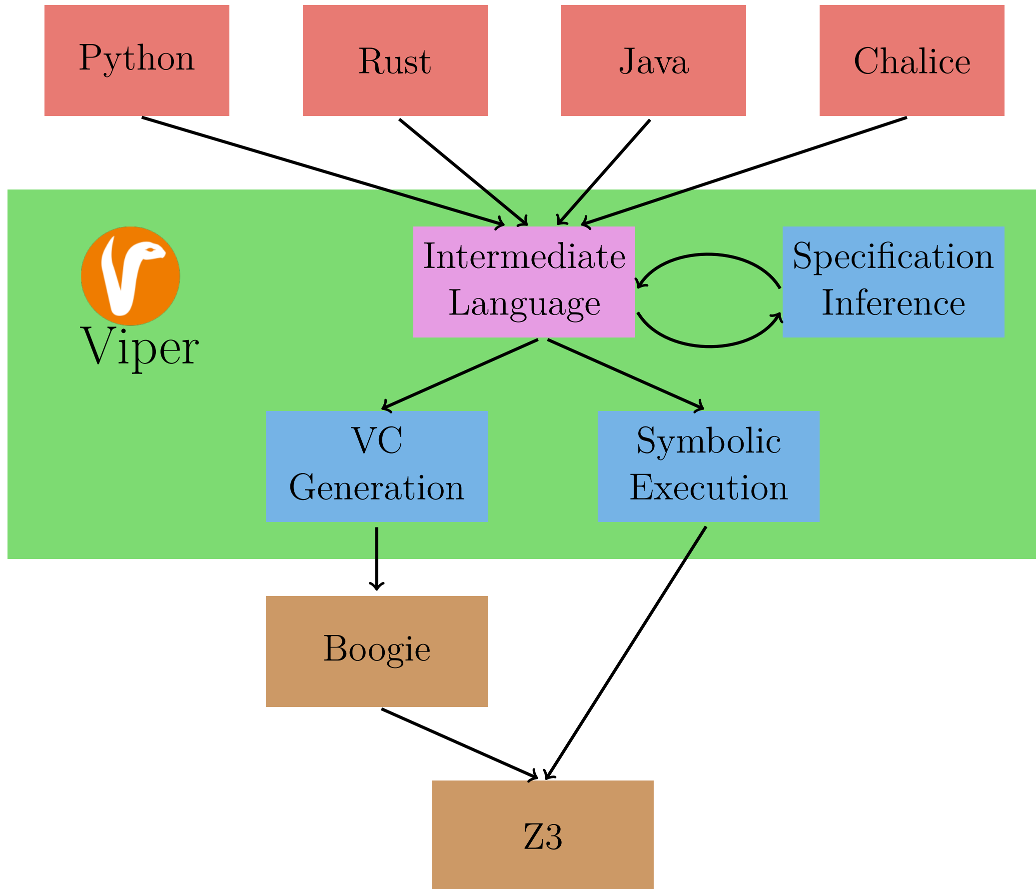


Figure 2.5: Overview of the Viper[1]toolchain and the used components

symbolic execution and interpreted by an SMT Solver. An alternative workflow uses verification condition generation via an encoding into the Boogie language, which is then passed to an SMT Solver. The general toolchain can be seen in Figure 2.5.

A Viper program consists of a sequence of fields, methods, functions, and global declarations. A *field* models a class field, storing data of an object. As Viper has no class construct, every object contains all declared fields. *Methods* and *functions* describe the program semantics and are like traditional C-style functions. The difference is the state information. Viper functions and methods can both read object memory, but only a method can change the data. The body of a method consists of a list of statements and expressions, a function can only use expressions to define its semantics. Global declarations consist of *predicates*,

[1]Image source: https://www.pm.inf.ethz.ch/research/viper.html

```
method sum(n: Int) returns (res: Int)
    requires n >= 0
    ensures  res == n * (n + 1) / 2
{
    res := 0
    var i: Int := 0
    while(i <= n)
        invariant i <= (n + 1)
        invariant res == (i - 1) * i / 2
    {
        res := res + i
        i := i + 1
    }
}
```

Figure 2.6: Viper example program. The method computes the sum of the first n natural numbers and ensures this result in the postcondition.

used to abstract over recursive assertions, and *domains*, to create custom types, and *macros*.

Program verification is performed independently for each method. On the entrance of a method, the callee asserts its precondition and ensures only its postcondition for the result and the heap state. The caller loses ownership and all knowledge of passed objects. Loops within the program are considered to be a black box for all statements after the loop. Each variable assigned within a loop is considered unknown within the program state and only the loop invariant is assumed to be valid. The loop body must ensure that the invariant holds before and after every iteration.

Figure 2.6 shows a Viper program for iterative summation over the first `n` natural numbers. The postcondition, provided with the keyword `ensures`, states what we want to prove—the equivalence between the result of the computation and the closed formula. The precondition, here `requires n >= 0`, defines the only assumptions about the input arguments. The most important part is given by the loop statement and its invariants. The first invariant bounds the value of the iteration variable `i` and ensures the termination of the loop. The second invariant states the relationship between the current value of `res` and the current iteration. With this, the backend can verify the method as the negated loop condition and the invariant implies the postcondition.

To read or write a field, the function or method must hold the permission to do so. Permissions are represented by a rational number r with $0 \leq r \leq 1$. Any non-zero permission allows for reading the heap location, and a value of 1 exclusively permits modifications. If the permission value r equals zero, all access to the heap is denied. Declaring a new object can be achieved by creating a variable of *reference* type and calling the *new* function. Every field, access is required to, has to be passed by its name as a function argument, directly granting permissions to all given fields. Consider the following statement:

```
    var R: Ref := new(a,b)
```

This line creates a heap object with fields `a` and `b` and stores its reference in the variable `R`. A function checking if both field values are equal has to ensure it has the permission to read from memory and can be defined like this:

```
function check(obj: Ref): Bool
    requires acc(obj.a, 1/1) && acc(obj.b, 1/1)
{
    obj.a == obj.b
}
```

An important build-in type of Viper is the sequence type, given by the `Seq` keyword. A sequence models an array of a single type and allows access through indexing if the index is within the range of the sequence. Viper also provides direct operators to append two sequences, take a subsequence or get the number of elements. These functionalities are shown here:

```
Seq(0,1,2)[1..] ++ Seq(3)  == Seq(1,2,3)
Seq(1,2)[0] + |Seq(3,4)| == 3
```

Here we first take a subsequence of all but the first element(`[1..]`) append a one element sequence containing a 3 (`++`), resulting in a new sequence `[1,2,3]`. In the second example, we take the first element (`[1]`) and add the length of the next sequence (`| |`).

3. Translation

In this section, we introduce an algorithm that generates a Viper program for a given Lola specification. We define functions for a streams computation delay and memory requirement and present the algorithm with an example.

3.1 Algorithm Overview

Algorithm 1 shows the translation algorithm and its main parts. The translation can be split into three parts. The *prefix*, *postfix* and the main *loop*. Every evaluation of a stream access with an offset in Lola can be seen as a case distinction. The monitor has to check whether the offset leads to a position before the beginning of the monitoring or after the stream ends. To simplify stream access expressions in the later verification, we unroll the loop iterating over all input positions. In the prefix, we explicitly handle all iterations in which a default value has to be used due to a negative offset. The needed delay for the evaluation of future reference will also be applied in this part. The postfix handles default values for future offsets and the remaining delayed computations after the input ends. The loop is responsible for all other iterations. Due to the unrolling, no case distinction is needed. Several loop invariants are required for Viper to ensure valid sequence accesses and termination. The only precondition i.e. input assumption, the algorithm has to make is that all input streams have an equal length greater than the minimal prefix length. To compute the prefix length and other important information, we have to analyze the dependency graph.

3.2 Dependency Graph Analysis

To evaluate a system trace for a specification with future dependencies, we have to infer when a stream value can be computed and all values needed for the evaluation are present. Consider the stream:

```
output check: Bool := sum[2,0] < 10
```

To compute the value of `check` at least three values of `sum` must already exist, i.e. this value cannot be computed before the third iteration. If the expression of `sum` also has future dependencies this delay could increase further. We define

```
Input   : LolaSpec
Output : Viper-Program
begin
    shifts, memory, p ⟵ Graph_Analysis(LolaSpec);
    Header(); // initializes meta variables
    for i ← 0 to p do // Prefix
        for stream ∈ LolaSpec do
            if shifts[stream] ≤ i then
                Iteration(i,stream)
            end
            if memory[stream] > 0 then
                Memory_Update(memory[stream],stream)
            end
        end
    end
    Generate_Invariant();
    Loop_Computation();
    for i ← 0 to max(shifts) do // Postfix
        for stream ∈ LolaSpec do
            if shifts[stream] ≤ i then
                Iteration(i,stream)
            end
            if memory[stream] > 0 then
                Memory_Update(memory[stream],stream)
            end
        end
    end
end
```

Algorithm 1: Lola translation algorithm

```
input in: Int
output b: Int := in[-3,0]
output f: Int := in[3,0]
output o: Int := f[-4,0]
```

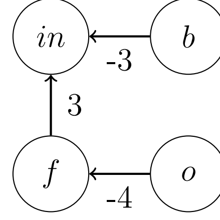


Figure 3.1: A small specification and its DG to show the effect of a streams shift to its memory requirement.

this computation delay as the *shift* of a stream, designated by the function $\Delta : Stream \mapsto \mathbb{N}$. Δ maps a positive integer to each stream s which corresponds to the weight of the longest path in the dependency graph starting in s. If the weight of this path is negative, the shift is set to zero. With this, we can also derive the memory needed for each stream.

Consider the specification in Figure 3.1. As `b` accesses the input stream with a weight of -3 we need to store the last 3 input values of this stream. The stream `f` only has a future access to the input stream and does not influence the memory requirement. The stream `o` uses the value of `f` with an offset of -4 but we only need to store a single value of `f`. Because $\Delta(o) = 0$ and $\Delta(f) = 3$ the newest value of `f` is already three iterations behind `o` and could be accessed by `o` with an offset of `-3`. The expression for `o` tries to get the previous value of `f` due to the shift, so only this value has to be stored. The memory needed for a stream s can be computed by the function $memory : Stream \mapsto \mathbb{N}$

$$memory(s) = \max(0, \max_{(s_i,s_j,w)\in E} \begin{Bmatrix} -w + \Delta(s_i) & \text{if } s_j = s \\ 0 & otherwise \end{Bmatrix} - \Delta(s))$$

Intuitively, we check the oldest value ever needed for a stream by iterating over all incoming edges in the DG and subtract the number of iterations that are not computed until the stream ends due to the streams shift.

To compute Δ we have to solve the all-pairs longest path problem in the DG, with any algorithm of choice. Note that the actual path is not of interest, only the length of the path influences the results. With the two functions *memory* and Δ, we obtain the minimal prefix length p. The prefix length for a specification ϕ is defined as follows:

$$p(\phi) = \max\{\Delta(s) + memory(s) \mid s \in Streams\ in\ \phi\}$$

	flow	signal	sum	expect	trigger
Δ	0	0	1	2	2
memory	2	0	1	0	0

Table 3.1: The values of *memory* and Δ for the flow example from Figure 2.2

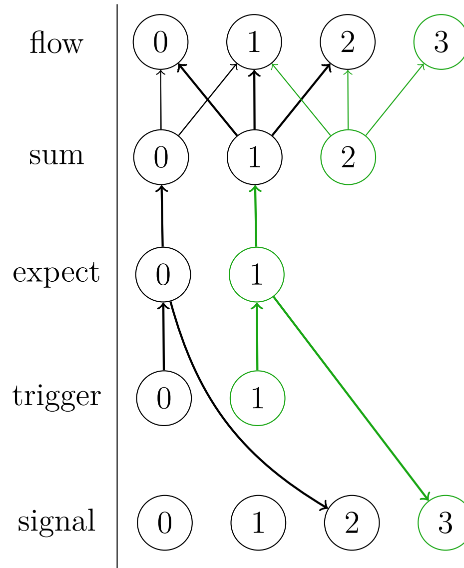


Figure 3.2: Visualization of stream accesses in the third iteration (black) and new additions of iteration four (green) of the flow example specification. Green nodes show values computed in the fourth iteration. Black nodes with ingoing green edges need to be stored in previous iterations.

For an explicit example, we compute all needed values for the flow example in Table 3.1. The shift for all streams except the trigger can be read directly from the edges, as the result is equal to the longest outgoing edge. $\Delta(trigger)$ is equal to the shift of `expect` due to the zero edge.

3.3 Pre-/Postfix Construction

The prefix applies the shift to each stream and creates a code block for each iteration needed. Consider again the example of Figure 2.2. In the very first iteration we get the first set of input data, but cannot compute any values for the outputs streams as both streams `sum` and `expects` have a shift greater than 0. In the second iteration, we can evaluate the first value of `sum` as the sum of the first and second entry of the flow stream and the default value for the -1 access. This stream evaluation then could be emitted as the following

Viper code:

$$sum := flow[1] + flow[0] + 0 \tag{3.1}$$

The zero addition at the end results from the default value of the -1 access i.e. $flow[-1]$. To ensure proper modeling of the Lola evaluation model, we only allow access to the sequence in the current iteration and store the value for the expression $flow[0]$ in the memory sequence:

$$sum := flow[1] + mem.flow[0] + 0 \tag{3.2}$$

Each value later needed is stored in sequences addressed by a single reference object. Figure 3.2 shows the temporal dependencies between the streams and what values are present or can be computed in the third iteration. The output variable `expect` can be computed in the third iteration resulting in the first value for this stream. Streams with a delay higher than the current iteration will not be evaluated

The postfix reverses the computation delay and handles all accesses leading to values after the stream ends. After the loop ends, the monitor received all N values of all input streams and all remaining computations can be solved. If a stream s was shifted by $\Delta(s) = k$ then the last k entries still have to be resolved. For a better intuition, we omit memory accesses and provide the last computation of the *sum* stream:

$$sum := 0 + flow[N-1] + flow[N-2] \tag{3.3}$$

3.4 Loop Generation

The loop handles all remaining iterations not computed in the pre- or postfix. The loop starts at iteration $p + 1$ and models all other $N - p - 1$ iterations. In the loop body all streams are evaluated and the memory is updates like in the prefix. Additionally the loop variable is incremented. The important part lies within the invariants we need to provide to Viper. The information reflected in the invariants are:

- loop variable i is bounded: $p \leq i \leq N$
- permission for heap access is preserved
- length of each memory sequence is unchanged

These points can best be seen in the next example. Consider the following example specification with its complete code in Figure 3.3:

Example 3.4.1.

```
input a: Int
input b: Int
output out: Int := b[1,1] + a[-1,-1]
```

To provide a complete intuition, we analyse the output for this specification. First, we see that $\Delta(out) = 1$, $\Delta(a) = 0$ and $\Delta(b) = 0$ as well as $memory(out) = 0$, $memory(a) = 2$, $memory(b) = 0$ and $p = 2$. We now know a memory sequence for `a` of size two is needed but no memory for any other streams. The value of p shows that the prefix handles the first three iterations.

Inspect the code model in Figure 3.3 line-by-line. The first line defines a field of sequence type for every stream with $memory > 0$, here only the first input stream `a`. Line 2 declares the monitor method. For each independent stream, a sequence is given as an argument. The preconditions in lines 3 and 4 assert equal length of all inputs and a minimal length of $p + 1$. The body starts with declaration initialization of some meta-variables, the memory reference, and all output variables, lines 6 to 11.

The first iteration has no output stream value to compute, only the current value of `a` is stored in the corresponding memory. The second iteration can evaluate the first value of `out`, using the default value for the stream access to `a` as it would access the input stream at position -1. In the next step, we see a full computation without the use of default values or missing evaluations for iteration two. This additional iteration is not needed for the translation, but it will be needed to verify the loop invariants we add in the next chapter. Line 23 declares the loop variable with the initial value $p + 1$. The loop handles the remaining iterations from $p+1$ to N. The first invariant has to ensure the loop terminates and sequence accesses with index i are valid by bounding the variable in line 25. The next two invariant expressions ensure access permission to the memory sequence is preserved and the sequence length is maintained. Without these two lines, the access to memory in line 29 cannot be validated.

As we change the value of a field, without the invariant all information considering this field would be lost, including access permission. The ordering of these invariants is important. All previous and already proven invariants are used as an assumption to prove the next invariant. The loop body performs a similar computation as the last prefix code, using parametrized access instead of a constant value and increasing `i`. After the loop body, we compute the remaining computation for the N-th value of `out` as in iteration zero we did not compute a value for this stream, due to its shift.

```
1  field a_mem: Seq[Int]
2  method monitor(a: Seq[Int], b: Seq[Int])
3      requires |a| == |b|
4      requires |a| >= 3
5  {
6      var N: Int := |a|
7      var mem: Ref
8      mem := new(a_mem)
9
10     mem.a_mem := Seq()
11     var out: Int
12
13     //Iteration 0
14
15     mem.a_mem := mem.a_mem ++ Seq(a[0])
16     //Iteration 1
17     out := b[1] + (-1)
18     mem.a_mem := mem.a_mem ++ Seq(a[1])
19     //Iteration 2
20     out := b[2] + (mem.a_mem[0])
21     mem.a_mem := mem.a_mem[1..] ++ Seq(a[2])
22
23     var i: Int := 3
24     while(i < N)
25         invariant 3 <= i && i <= N
26         invariant acc(mem.a_mem)
27         invariant |mem.a_mem| == 2
28     {
29         out := b[i] + (mem.a_mem[0])
30         mem.a_mem := mem.a_mem[1..] ++ Seq(a[i])
31
32         i := i + 1
33     }
34     //Iteration N+1
35     out := 1 + (mem.a_mem[0])
36     mem.a_mem := mem.a_mem[1..]
37 }
```

Figure 3.3: Complete translation output for the specification of Example 3.4.1

4. Verification

The translation in Chapter 3 returns a model for the monitor in Viper. In this chapter, we provide additional verification conditions and invariants to prove the soundness and completeness of all trigger expressions within the model of the monitor. Afterwards, we examine the monitor's behavior in a concurrent context and prove its correctness in Section 4.2.

4.1 Trigger Evaluation

Consider the running example from Figure 2.2 and its trigger expression. The output stream for the trigger so far is represented by the expression stated in the specification. The expression `trigger := !expect` determines the result of the trigger based on intermediate computations. To show the correctness of the computation, we assert the equivalence of this result to the input values leading to this value. For the first possible evaluation of this stream in iteration two, we inline all subexpressions step by step

$$\begin{aligned} trigger &= !expect \\ &= !(sum < 5 => signal[2]) \\ &= !((flow[1] + flow[0] + 0) < 5 => signal[2]) \end{aligned}$$

and then emit an assertion for this expression

```
assert trigger == !((flow[1] + flow[0] + 0)
         < 5 => signal[2])
```

If the DG of the specification contains a negative cycle, inlining cannot be done directly as the expression substitution can diverge or referenced values may no longer be available. To handle this problem, we make use of *ghost memory* in the proof.

4.1.1 Ghost Memory

Ghost memory is used to store additional information about the program state to improve the expressiveness in the proof. While the monitor has only constant

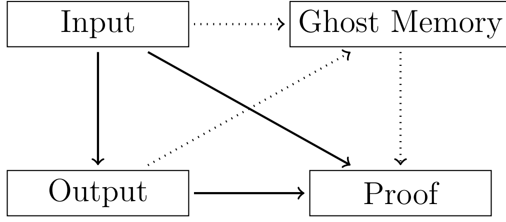


Figure 4.1: Visualization of ghost memory, additional memory stores intermediate results to be used in the proof but not in the computation

memory available, ghost memory uses potentially linear memory in the size of the trace. The computation of the model is not influenced in any way by this extension. Figure 4.1 shows an overview of the functionality. The input and output of the function, which are normally used within the proof, now also may be stored in the ghost memory alongside intermediate results. This will be used to model an abstract memory for the proof. For each trigger expression, we show the equivalence between the original computations and the evaluation on the abstract state and input values.

4.1.2 Invariant Generation

To guarantee access to all variables needed in the abstract memory, we need to store a value if a trigger expression or an inlined subexpression tries to access its memory. To determine these values, we analyze the DG to search for affected streams. The notion of *direct edges* simplifies this categorization.

Definition 4.1.1 (*Direct Edge*)**.** An edge $e = (s_i, s_j, w)$ in a dependency graph is called a *direct edge* iff $\Delta(s_i) - w = \Delta(s_j)$, and an *indirect edge* otherwise.

Evaluating an access over a direct edge results in the use of the newest stream value, accesses over indirect edges result in a memory access. For each trigger, we start at the corresponding node in the DG and mark all output streams s if a path $trigger = v_1, \ldots, v_{k+1} = s$ with edges $e_1, \ldots, e_k$ exists and $e_1, \ldots, e_{k-1}$ are direct edges and e_k is an indirect edge. For each marked stream and all triggers, we introduce a local sequence variable to store all occurring values. This information enables us to show the equality of the current trigger value and its computation on the abstract memory state.

To verify these assertions for an arbitrary iteration within the loop body, it is necessary to provide and prove additional loop invariants. First, we include an invariant for the memory of input variables and the actual input. If an input variable x has a memory requirement m_x greater than zero, we provide the invariant $x_{mem} = x[i - m_x \,..\, i]$, which proves that the memory entries are a subsequence of the complete input stream. Next, we make use of the ghost

memory information and provide a similar invariant for output streams. For each output variable y marked as needed for a trigger and that has memory requirements m_y greater than zero, we prove the equivalence between the older values of this stream and the ghost memory for this variable. The invariant $y_{ghost}\ [i - \Delta(y) - m_y\ ..\ i - \Delta(y)] = y_{mem}$ shows that the memory for an output stream is a subsequence of the ghost memory. To ensure all accesses to the ghost memory are within the sequence range, an invariant for each new sequence is added to state its length, $|y_{ghost}| = i - \Delta(y)$.

With these invariants, we prove the correctness for the trigger evaluations. For each trigger, it is necessary to provide a loop invariant and an assertion in the prefix and postfix for each computed value. The assertion for the second iteration of the running example is given by:

```
assert trigger_ghost[0] ==
    !(sum_ghost[0] < 5 => signal[2])
```

The added invariants for the running example from Figure 2.3 are:

```
    invariant mem.c_mem == flow[i-2 .. i]
    invariant |sum_ghost| == i - 1
    invariant |trigger_ghost| == i - 2
    invariant sum_ghost[i-2..] == mem.sum_mem
    invariant trigger_ghost[i - 3] ==
        !(!(sum_ghost[i - 3] < 5) || signal[i - 1])
```

4.2 Concurrency

The next property we want to verify is the correctness of the monitor in a concurrent context. Intuitively, the evaluation of different streams can be done in parallel if they do not use the result of each other. To analyze the monitor in this context, we define independent functions and the notion of a stream layer based on the input specification.

To separate the computation of each stream, we introduce a Viper function handling the expression computation within the loop body. The previous assignment of $expect := sum_{mem}[0] < 5 \Rightarrow signal[i]$, which corresponds to a loop computation of the boolean `expect` stream of Figure 2.2, is moved to a parametrized function. Every value accessed via memory can be passed through the memory reference object, in this case, the value of `sum`. The second stream value needed, here $signal[i]$, is first obtained in the current iteration and is not stored in memory at this point. All values of this kind are passed as an argument to the new function. This results in the new assignment of

$expect := compute_expect(mem, signal[i])$. The compute function is then appended to the main method. As the control-flow enters a new function scope when calling this function, we have to ensure the function holds all needed permissions to evaluate the expression. For every sequence in memory the stream tries to access, the precondition requires access to the sequence and the sequence length has to be stated. The emitted function to compute this stream is then:

```
function compute_expect(mem: Ref, signal: Bool) : Bool
    requires acc(mem.sum_mem,1/3)
    requires |mem.sum_mem| == 1
{
    mem.sum_mem[0] < 5 => signal
}
```

The chosen permission fraction of 1/3 in this example corresponds to one divided by the number of output streams. When the main method creates the memory object, it holds the complete permission of 1. Every function call transfers the demanded permission to the callee and regains it after the call. An equal split of this size can enable simultaneous read access for all streams. Note that using Viper functions over methods simplifies pre- and postcondition and argumentation about the memory state significantly. Modeling an equivalent computation as method would require the following code:

```
method compute_expect(mem: Ref, signal: Bool)
        returns (expect: Bool)
    requires acc(mem.sum_mem,1/3)
    requires |mem.sum_mem| == 1
    ensures acc(mem.sum_mem,1/3)
    ensures |mem.sum_mem| == 1
    ensures expect == mem.sum_mem[0] < 5 => signal
{
    expect := mem.sum_mem[0] < 5 => signal
}
```

A method has to define its return value with a specific variable and explicitly ensure that the access to all used memory fields is maintained. To get any information about the return value as a caller, we need to provide a respective postcondition. The last condition in the code example above reflects this property and asserts the exact computation. The evaluation of this expression needs assumptions about the sequence length, so the postcondition has to ensure the length, too. These postconditions could annotate the function, too, but are not needed as Viper can infer these properties automatically. The function model

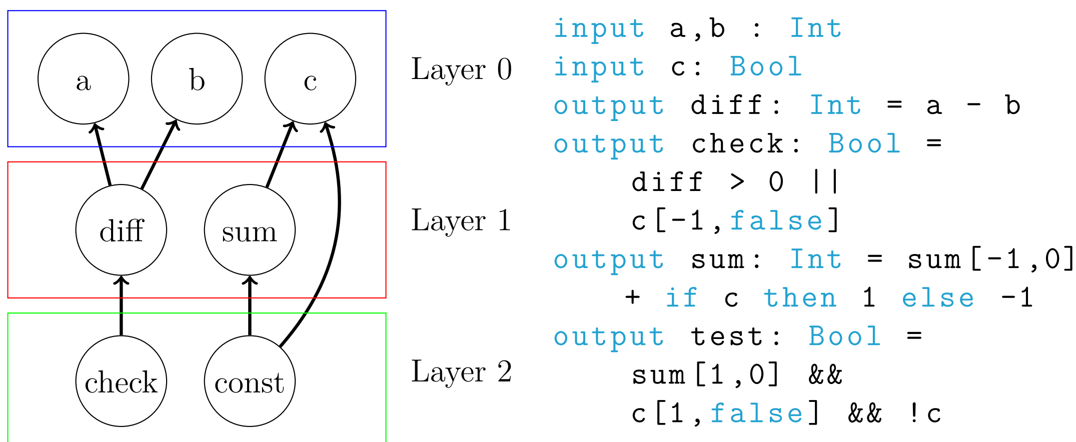


Figure 4.2: Layer visualization and the corresponding specification, the graph only contains the direct edges of the DG

directly ensures an unchanged memory state, due to the read-only access to the heap of functions, which is an important property we will use later in this chapter.

Next, we want to model the concurrent behavior of the monitor in the annotated monitor model. For this, we define *dependent computations* and the *layer* of a stream.

Definition 4.2.1 (*Layer*). The *layer* $L : Stream \mapsto \mathbb{N}$ of a stream s_i in the set of all streams S in the specification ϕ is the least $n \in \mathbb{N}$ such that $\forall s_j \in S : L(s_j) > L(s_i)$ if s_j has a direct edge to s_i and $L(s_j) < L(s_i)$ if s_i has a direct edge to s_j.

Figure 4.2 shows a visualization of the stream layers. Note that input streams are always in layer zero and all streams that have one or more direct edges are at least in layer one. The stream layers determine an *evaluation order* for all stream computations, representing a partial order over all streams, due to the less-than ordering for different layers and no ordering for streams in an equal layer. Before a new stream value can be computed, all streams in lower layers have to be evaluated. A similar definition for layers and evaluation order is defined in recent work on RTLola [4,24]. These publications do not directly consider future offsets in their definitions, which leads to differences in the definitions. For the streams of Figure 2.2 we obtain a layer of zero for both input streams, layer one for `sum` and `expect` and only the trigger lies in layer two as it directly depends on `expect`, which lies in layer one. Note that `expect` has an access with an offset of zero to `sum` but does not lie in a higher layer as this access is based on

an indirect edge. This difference can be seen in Figure 3.2, the node representing the first value for the `expect` stream has no edge to the newest value of `sum`. We can use the information of a stream layer to divide all streams into disjunct classes. All streams in the same layer do not need the current value of any other stream in this layer or a higher layer. We modify the monitor model concerning the stream layers by sorting all stream computations in increasing layer order. The layer ordering ensures that no variable is used before its computation within the model and the semantics are unchanged.

In the context of computations based on heap state information, we define dependency of functions and show independency of all streams in the same layer. The notation of $\langle \cdot , \cdot \rangle \Downarrow$ symbols the evaluation of a function on a given program state and returns the functions result and the program state after the evaluation. The ; operator symbolizes consecutive execution.

Definition 4.2.2 (*State-Dependency*)**.** A pair of functions or methods $f_1 : A_1 \mapsto B_1$,$f_2 : A_2 \mapsto B_2$ is called *state-dependent* on a state h iff for any valid inputs $x_1 \in A_1, x_2 \in A_2$ and some values in the target sets $y_1, y_1' \in B_1, y_2, y_2' \in B_2$ the two possible evaluations are $\langle f_1(x_1), h\rangle \Downarrow y_1, h_1; \langle f_2(x_2), h_1\rangle \Downarrow y_2, h_2$ and $\langle f_2(x_2), h\rangle \Downarrow y_2', h_2'; \langle f_1(x_1), h_2'\rangle \Downarrow y_1', h_1'$ and $y_1 \neq y_1' \vee y_2 \neq y_2'$ holds.

Theorem 1. A pair of Viper functions is state independent on any state.

Proof. A Viper function can only read heap information and not change them due to the language semantics, even with a permission value of 1. Given two Viper functions f_1, f_2, some inputs x_1, x_2 and a state h the two possible executions are $\langle f_1(x_1), h\rangle \Downarrow y_1, h; \langle f_2(x_2), h\rangle \Downarrow y_2, h$ and $\langle f_2(x_2), h\rangle \Downarrow y_2', h$; $\langle f_1(x_1), h \rangle \Downarrow y_1', h$ with only h as possible state after any execution. The unchanged state directly implies $y_1 = y_1' \wedge y_2 = y_2'$. As functions are deterministic, two execution on the same state and input return the same result. □

Definition 4.2.3 (*Dependency*)**.** A pair of functions or methods f_1 and f_2 is *dependent* if they are state dependent on an arbitrary state or one function uses the input of the other function, i.e. f_1 returns a value v, and the input of f_2 uses v directly or as subexpression.

This definition intuitively says two functions are dependent if the order of evaluation influences the results i.e. $f_1; f_2 \neq f_2; f_1$.

Theorem 2. Two stream evaluation functions within the same layer are independent.

Proof. The pair of functions for two streams s_i and s_j in the same layer cannot depend on the result of the other function as the definition of stream layer would otherwise ensure $L(s_i) > L(s_j)$ or $L(s_j) > L(s_i)$. Furthermore, we know that the evaluation functions are state-independent, shown in Theorem 1. □

As stream evaluations are expressed as independent functions within a layer, all computations within this layer are interleaving invariant. An arbitrary scheduler cannot change the function results or produce data-races within a layer. Extracting the stream evaluation into functions with partial permissions and arranging all computations in order of the stream layers models the parallel processing of the Lola semantics within a layer. With this and proving the independence of functions, we can conclude the correctness of the concurrent behavior of the monitor provided Viper can prove the correctness of the new model.

5. Evaluation

The translation algorithm is implemented in Rust and is available on GitLab[1]. The implementation uses the Lola parser from the StreamLAB [11] framework. The Viper IDE is available as a Visual-Studio Code extension and a standalone interpreter[2]. The translation implementation has to make several restrictions on the input specification due to the semantics of Viper. We encountered some specifications with a long verification time. To reduce the runtime to an acceptable level, we bound the expression inlining for trigger invariants, see Section 5.2.

5.1 Division and Rationals

Real numbers are needed to accurately describe real-world systems. In Viper, these numbers are approximated with the built-in `Rational` type. A `Rational` value is given by an expression of the form `x / y`. While `x` is a value of type `Int` or `Rational`, `y` has to be of type `Int`. Computations like $5/(3/7)$ cannot be expressed without a semantic analysis and constant folding, e.g. a valid representation is $5 * 7/3$. If the divisor is a variable of rational type, a representation is not possible. Each division expressed in Viper has to be verified separately. If the divisor cannot be proven to be unequal to zero, the verification fails. Therefore, for each evaluation of a division in the specification, an internal assertion has to be proven. Consider the following example expression with two divisions and assume neither variable is a constant:

```
out := a / ((b - c)/d)
```

The assignment could not be verified unless it is extended with two assumptions:

```
assume d != 0
assume ((b - c)/d) != 0
out := a / ((b - c)/d)
```

[1]https://gitlab.com/Nekronod/vitola
[2]https://www.pm.inf.ethz.ch/research/viper/downloads.html

5.2 Expression Inlining Depth

To show the equivalence of the trigger evaluations to the computation on the abstract memory state, we described an expression inlining procedure in Section 4.1.2. These inlined expressions are used in every iteration and the loop invariant. The verification statements for these expressions can be hard for Viper to prove as we do not reuse expressions and recompute all values if no memory access is required. Consider a specification with a computation of the following form:

```
a:= in1[i] + in1[i]
b:= a * a
c:= b * b
d:= c * c
e:= d * d
f:= e * e
trigger := in2[i] > f
```

All used values are either inputs or variables without the need of memory. Intuitively, this is easy to prove, but the computed invariant would expand into this form:

```
invariant trigger_ghost[i-1] == in2[i-1] >
(in1[i-1] + in1[i-1]) * (in1[i-1] + in1[i-1]) *
(in1[i-1] + in1[i-1]) * (in1[i-1] + in1[i-1]) *
(in1[i-1] + in1[i-1]) * (in1[i-1] + in1[i-1]) * ...
```

After inlining, long specifications with many intermediate values and only a few indirect edges in their DG, like the example above, can have an invariant exponentially big in the size of the expression . To be able to prove specifications like this, it is possible to restrict the number of recursive calls when inlining the trigger expression. With a restricted exploration depth, other variables are stored in the ghost memory and a smaller invariant is emitted. With a bound of two iterations, the invariant would change to:

```
invariant trigger_ghost[i-1] == in2[i-1] >
(c_ghost[i-1]) * (c_ghost[i-1]) *
(c_ghost[i-1]) * (c_ghost[i-1])
```

This would reduce the number of subexpressions from 64 to four and can decrease the verification time in larger examples significantly. This expression and runtime expansion shows the trade-off between verification runtime, the accuracy of the monitor model and the expressiveness of the verification conditions.

Table 5.1: Runtime statistic over 50 runs of verifying the drone specification. A timeout of 60 minutes was applied for long runs. Machine used: 16GB RAM, Intel Core i7-8565U CPU @ 1.80GHz, 4 Cores

inline level	max [s]	min [s]	avg [s]	number of timeouts
1	13	5	7	0
2	10	5	7	0
3	TO	5	284	2
4	TO	8	889	10
5	TO	TO	TO	50

5.3 Runtime and Memory

The presented translation approach produces a Viper file for a given Lola specification. The Viper toolchain takes a Viper program as input, translates it into an SMT model and calls the Z3 solver to obtain a verification result.
The Lola translation algorithm has negligible runtime. Generating a Viper file for a specification is completed in under five milliseconds. The translation result was evaluated and tested with a simplified version of a specification used to cross-validate GPS and IMU sensor data for drone flights. This specification was originally presented in a case study on drone monitoring [1]. We simplified the specification by changing the type of all variables from a real number to an integer, as we have to ensure all divisors are of integer type. At two places, the specification uses division by a variable. Therefore additional assumptions are need to prove the correctness of the concerned stream expressions. Note that this could be interpreted as an error within in the specification. In both cases the correctness of the division results from an internal invariant which is not explicitly stated. Furthermore, we removed mathematical function calls such as *sin* and *sqrt* due to incompatible types. The used specification can be seen in Figure 5.1. The direct translation of this specification could not be proven within four hours.With a bounded inlining level, the specification was provable for several small levels. While the SMT model generation of Viper is deterministic and can be multi-threaded, the solving by Z3 only works with a single thread and has an unpredictable runtime. As Z3 chooses random paths to explore in the search space, the solver's runtime can vary drastically. The translation for the drone specification uses only five MB memory and runs in under ten milliseconds.

Table 5.1 shows the evaluation results for different inlining depths. Low degrees of inlining could always be proven in under 20 seconds. For an inlining-bound

```
input lat: Int32, lon: Int32, ug: Int32, vg: Int32, wg: Int32,
   time_s: Int32 , time_micros: Int32

output time: Int32 := time_s + time_micros / 1000000
output count: Int32 := count[-1,0] + 1
output flight_time := (time - time[-1,0]]) * count
output frequency := 1 / (time - time[-1,0]])

output freq_sum := frequency + freq_sum[-1,0]
output freq_avg := freq_sum / count

output freq_max := if frequency > freq_max[-1,0] then frequency
   else  freq_max[-1,0]
output freq_min := if frequency < freq_min[-1,0] then frequency
   else  freq_min[-1,0]

output velocity := ug*ug + vg*vg + wg*wg
output R := 6373000
output approximate_pi := 3

output lon1_rad := lon[-1,0] * approximate_pi / 180
output lon2_rad := lon * approximate_pi / 180
output lat1_rad := lat[-1,0] * approximate_pi / 180
output lat2_rad := lat * approximate_pi / 180

output dlon := lon2_rad -lon1_rad
output dlat := lat2_rad -lat1_rad

output a := (dlat/2)*(dlat/2) * lat1_rad * lat2_rad * (dlon/2)*(
   dlon/2)

output c := 2 * ( a*a  + (a/2) * (a/2))
output gps_distance := R * c

output passed_time := time - time[-1,0]
output distance_max := velocity * passed_time
output dif_distance := gps_distance - distance_max
output delta_distance := 10
output detected_jump: Bool := dif_distance > delta_distance

trigger detected_jump "Jump"
```

Figure 5.1: Simplified specification for drone sensor cross validation.

of three, some cases exceeded the timeout of one hour. Most evaluations could be verified in several seconds. Inlining for four iterations shows similar behavior as the previous case but took about four times longer on average. We can see a drastic runtime increase when inlining for five iterations, as it could not be evaluated within several hours. With a bound of five, the expression of the stream `a` in Figure 5.1 is used inside the evaluation of stream `c` which leads to a big expansion of the formula and the corresponding invariant. While for lower inlining levels the model generation of Viper is completed in a few seconds, this part takes up to 30 minutes for an inlining level of five. The resulting invariants and their size for different inlining levels can be seen in Figure 5.2.

Figure 5.3a shows runtime and memory consumption for a test series of 100 runs, with an inlining-bound of three and a timeout of 15 minutes. This experiment has shown that executions with low runtime also have low memory usage. High memory consumptions imply a high runtime but not vise versa. Every Z3 execution that needed more then 500MB memory on this example resulted in a timeout. Long runs did, however, not always need a lot of memory. Note, that the runtime distribution builds a cluster of values below the one minute mark. 20 out of 100 runs showed a runtime above three minutes.

Specifications with only a few intermediate values and few direct edges within a single path in the DG are far easier to verify. This is due to the fact that the expression inlining for the invariant does not substitute dependencies through memory accesses. The average runtime for all other examples, beside Figure 5.1, is below five seconds. Besides all examples including division on real numbers, Viper was able to prove all specifications that did not exceed the timeout. Every proven specification could also be verified after applying the changes of Section 4.2. With this, we have shown the monitor's correctness in a concurrent context.

Additionally, an experiment series on a smaller specification with a highly connected DG has shown a relation between the specification size and the verification time. While the drone specification shows a tree like structure with edge weights close to zero, the specification used here has many edges with small negative weights. The corresponding DG has multiple edges with negative weights and cycles with negative weights. The base test case, seen in Figure 5.4 has an average verification time of nine seconds and needs 460 MB memory in the process. Reducing the weight of all edges in the DG by one increased the verification time average to ten seconds. Iterating this process up to five times increased the average runtime by about one second and the memory usage by 10 MB. Doubling the number of streams and providing a second trigger expression increased the runtime up to 50 seconds and increased the needed memory by

```
invariant trigger_Jump_ghost[i - 1] == ((((R_ghost[i - 1] *
   c_ghost[i - 1]) - (velocity_ghost[i - 1] * passed_time_ghost[
   i - 1])) > 10))
```

(a) Generated with a bound of three

```
invariant trigger_Jump_ghost[i - 1] == ((((6373000 * (2 * ((
   a_ghost[i - 1] * a_ghost[i - 1]) + ((a_ghost[i - 1] / 2) * (
   a_ghost[i - 1] / 2))))) - ((((ug[i - 1] * ug[i - 1]) + (vg[i
   - 1] * vg[i - 1])) + (wg[i - 1] * wg[i - 1])) * (time_ghost[i
    - 1] - time_ghost[i - 2]))) > 10))
```

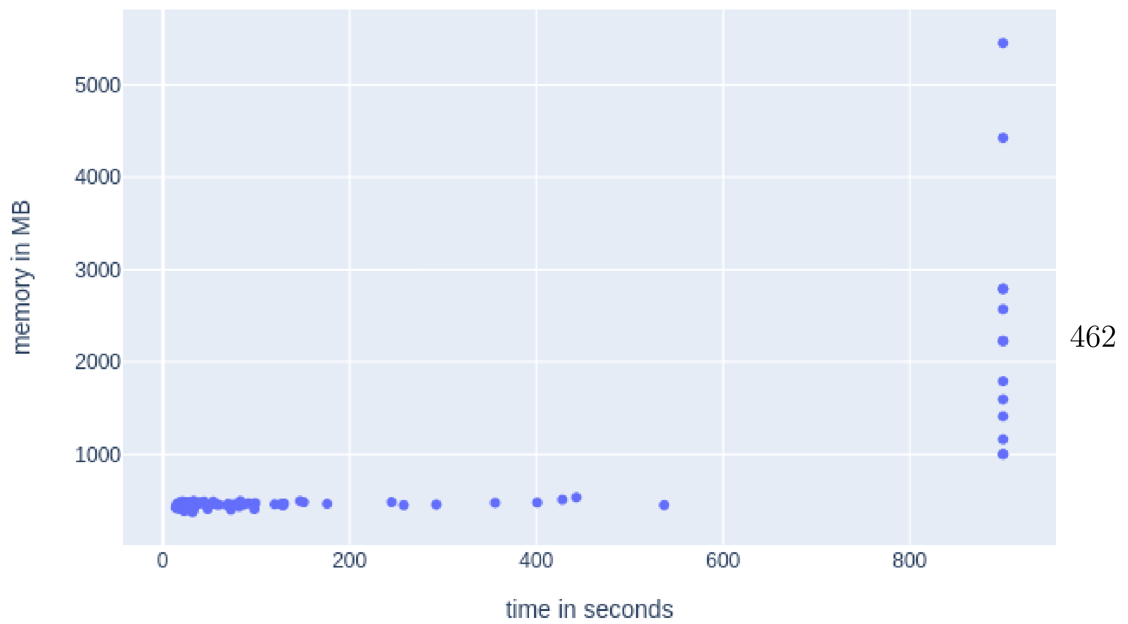
(b) Generated with a bound of four

```
invariant trigger_Jump_ghost[i - 1] == ((((6373000 * (2 *
   ((((((((dlat_ghost[i - 1] / 2) * (dlat_ghost[i - 1] / 2))
   * lat1_rad_ghost[i - 1]) * lat2_rad_ghost[i - 1]) * (
   dlon_ghost[i - 1] / 2)) * (dlon_ghost[i - 1] / 2)) *
   ((((((dlat_ghost[i - 1] / 2) * (dlat_ghost[i - 1] / 2)) *
   lat1_rad_ghost[i - 1]) * lat2_rad_ghost[i - 1]) * (
   dlon_ghost[i - 1] / 2)) * (dlon_ghost[i - 1] / 2))) +
   ((((((((dlat_ghost[i - 1] / 2) * (dlat_ghost[i - 1] / 2))
   * lat1_rad_ghost[i - 1]) * lat2_rad_ghost[i - 1]) * (
   dlon_ghost[i - 1] / 2)) * (dlon_ghost[i - 1] / 2)) / 2) *
   ((((((dlat_ghost[i - 1] / 2) * (dlat_ghost[i - 1] / 2)) *
    lat1_rad_ghost[i - 1]) * lat2_rad_ghost[i - 1]) * (
   dlon_ghost[i - 1] / 2)) * (dlon_ghost[i - 1] / 2)) / 2))))
   ) - ((((ug[i - 1] * ug[i - 1]) + (vg[i - 1] * vg[i - 1]))
   + (wg[i - 1] * wg[i - 1])) * ((time_s[i - 1] + (
   time_micros[i - 1] / 1000000)) - time_ghost[i - 2]))) >
   10))
```

(c) Generated with a bound of five

Figure 5.2: The invariant for the specification in Figure 5.1 with different inlining levels

100 MB, up to 560 MB. Adding a stream to the specification yields a higher increase of the runtime than changes in the DG structure or changes of edge weights. This experiment supports the conclusion that the specification is the most relevant aspect of the verification runtime.



(a) Visualization of runtime and memory usage of the drone specification. The data consists of 100 runs with an inline bound three and 15 minutes timeout.

(b) Statistical values of the used data set. Memory is reported in MB, time in seconds. 14 evaluations exceed the timeout of 15 minutes.

	time [s]	memory [MB]
max	TO	5455
min	14.22	375
avg	185.93	700
mean	42.32	460

Figure 5.3: Time and memory evaluation for the drone specification.

```
input a:Int32, b:Int32, c:Int32
output s:= a + b[1,1] + c[-1,2]]
output r1 := s[-3,0] + s[-2,0]
output d := r1[-2,0] - r1[-3,0] + m
output h :=  d[-1,3] - r1[-2,13]
output k := h[-2,2] +r1[-1,2]
output n := a + b +c
output m := h + d[-1,3] - n[-1,4]
output j := b * n[-1,1] - c
trigger j > m[-1,2] "tr2"
```

Figure 5.4: The base specification for the test series to evaluate the influence of the DG structure on the verification runtime. This specification is step-by-step expanded to different sizes.

6. Related Work

Similar to Lola [8], languages for synchronous programming like ESTEREL [5], LUSTRE [15] and SIGNAL [14] also define programs as stream equations. These programs can be compiled into an executable binary but lack future references. An alternative approach for runtime monitoring, EAGLE [3], is based on extended regular expressions. By adding complementation to regular expressions, the language allows for an intuitive way to formalize specifications. Each specification is described by a set of parameterized recursive definitions and a few primitive temporal operators. EAGLE allows the user to define rules that implement a monitor and is the monitoring tool that most resembles Lola. As EAGLE uses logical formulas similar to LTL, it lacks the power to express numerical queries. The common approach for runtime monitoring is based on temporal logic, but these approaches have to modify their logic to handle finite traces. Building on Lola 2.0, RTLola (Real-Time Lola) [12] is capable of dealing with variable-rate input streams and sliding windows, e.g. computing the average value of an input stream over the last 5 seconds.

Other static verification tools for different languages, besides Viper [22], rely on common intermediate representations like Boogie [20] or Why [13]. Boogie, for example, is the core of Chalice [21] and Corral [19], while e.g. Frama-C [18] is built on Why. These tools lack the ability to support permission-based logic, which allows simulating heap read-only accesses and reasoning about concurrency and aliases. Several verifiers for commonly used programming languages are build on top of Viper, like Nagini [10] for Python and Prusti [2] for Rust. The tool coreStar [6] builds the basis for the JStar [9] verification tool for Java and also builds on separation logic including permissions. Building on coreStar requires the user to provide proof and abstraction rules to customize the behavior of the symbolic execution. Viper supports a wide variety of languages without the need for detailed back-end knowledge.
Iris [16, 17], a framework similar to Viper, is built to encode and unify advanced higher-order concurrent separation logic and formalizes them in Coq. Iris additionally provides soundness of all proof rules and is not bound to a specific

surface logic. The logic used for Viper provides lower overhead for annotations than approaches using Coq formalization and complete automation after the logic application.

An approach similar to the one presented in this thesis is presented by Copilot [23], a Haskell-based specification language. A Copilot specification is compiled into two separate C sources. One compiled through a custom back-end and one by a copilot-SBV-C toolchain. A driver interleaves both sources step-wise and asserts their equality. CBMC [7], a model-checker for C sources, proves that both programs have equal outputs for all inputs within a stated iteration range. The given verification attempt only considers a constant number of iterations and cannot verify arbitrary behavior. Copilot specifications have to be causal, i.e. no future references are allowed. Lola, which respects the *descriptive* nature of runtime specifications, allows for future references. One significant advantage of Copilot is the ability to monitor real-time systems. This also can be achieved with RTLola, however, a translation to Viper is not immediately possible, as Viper is not real-time capable.

7. Conclusion

In this thesis, we presented a transpilation for the stream-based specification language Lola to the verification language Viper. The Viper program models the monitor which reflects the Lola specification. The model is annotated with verification conditions to use the verification system of Viper. We use Viper to prove the correctness of the monitor. The translation is implemented as a Rust program and uses the Lola parser of the StreamLAB framework. To evaluate a stream expression, the algorithm performs a graph analysis for the longest paths and checks for streams reachable with paths over direct edges on the dependency graph. Several loop invariants and assertions are added based on a syntactical analysis to show soundness and completeness of all trigger evaluations i.e. reports to the user. The notions of evaluation order and stream layer are introduced to prove the correct behavior of the monitor in a concurrent context. The evaluation shows features of the Rust implementation and bounds of the Viper proof toolchain. Logical complex specifications with highly connected dependency graphs and large edge weights are complex to model, but easy to proof, while computation heavy specifications can overwhelm the Viper backend.

7.1 Future Work

In future work, we aim to expand the input grammar to the parametrized template streams of Lola 2.0. Checking a network connection for too many requests by a single user cannot be expressed in the first version of Lola easily. Parametrized stream conditions can, for example, store the number of requests per IP-address in a single line in the specification. With the extended grammar it will be possible to prove the correctness monitors for more generic and abstract specifications and systems.
Another interesting aspect to explore in more detail is the RTLola language for real-time specifications and dependencies. Stream expressions aggregating over the last minute of input values are useful to describe real-time systems but are not easily representable inside the Viper architecture. Alternative verification techniques or toolchains, next to Viper, are required to formally prove correctness for RTLola runtime monitoring.

The final goal to prove the correctness of a Lola runtime monitor would be based on the verification of the Lola interpreter and compiler. The presented approach is the first step in this direction. As the Lola interpreter and compiler are written as a Rust program another approach based on Viper can be applied. The Rust verifier Prusti [2] is built on top of the Viper toolchain and can be used to annotate Rust programs with verification conditions and prove their correctness. Building on the language's strong type system, it can show the correctness of complete Rust programs and could be used to verify the whole Lola compiler.

Bibliography

[1] F.-M. Adolf, P. Faymonville, B. Finkbeiner, S. Schirmer, and C. Torens. Stream runtime monitoring on UAS. In *International Conference on Runtime Verification*, pages 33–49. Springer, 2017.

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging rust types for modular specification and verification. 2018.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 44–57. Springer, 2004.

[4] J. Baumeister, B. Finkbeiner, M. Schwenger, and H. Torfah. FPGA stream-monitoring of real-time properties. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–24, 2019.

[5] G. Berry. The foundations of Esterel. In *Proof, language, and interaction*, pages 425–454, 2000.

[6] M. Botincan, D. Distefano, M. Dodds, R. Grigore, D. Naudziuniene, and M. J. Parkinson. corestar: The core of jStar. *Boogie*, 2011:65–77, 2011.

[7] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[8] B. d'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE, 2005.

[9] D. Distefano and M. J. Parkinson J. jStar: Towards practical verification for Java. *ACM Sigplan Notices*, 43(10):213–226, 2008.

[10] M. Eilers and P. Müller. Nagini: a static verifier for Python. In *International Conference on Computer Aided Verification*, pages 596–603. Springer, 2018.

[11] P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, and H. Torfah. StreamLAB: stream-based monitoring of cyber-physical systems. In *International Conference on Computer Aided Verification*, pages 421–431. Springer, 2019.

[12] P. Faymonville, B. Finkbeiner, M. Schwenger, and H. Torfah. Real-time stream-based monitoring. *arXiv preprint arXiv:1711.03829*, 2017.

[13] J.-C. Filliâtre and A. Paskevich. Why3where programs meet provers. In *European Symposium on Programming*, pages 125–128. Springer, 2013.

[14] T. Gautier, P. Le Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Conference on Functional Programming Languages and Computer Architecture*, pages 257–277. Springer, 1987.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[16] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *ACM SIGPLAN Notices*, volume 50, pages 637–650. ACM, 2015.

[17] J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[18] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[19] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *International Conference on Computer Aided Verification*, pages 427–443. Springer, 2012.

[20] K. R. M. Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[21] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009.

[22] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference*

on Verification, Model Checking, and Abstract Interpretation, pages 41–62. Springer, 2016.

[23] L. Pike, N. Wegmann, S. Niller, and A. Goodloe. Copilot: monitoring embedded systems. *Innovations in Systems and Software Engineering*, 9(4):235–255, 2013.

[24] M. Schwenger. Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS. Master thesis, Saarland University, 2019.