Declarative Stream Runtime Verification (hLola)*

Martín Ceresa[1], Felipe Gorostiaga[1,2,3], and César Sánchez[2]

[1] CIFASIS, Argentina
[2] IMDEA Software Institute, Spain
[3] Universidad Politécnica de Madrid, Spain

Abstract. Stream Runtime Verification (SRV) is a formal dynamic analysis technique that generalizes runtime verification algorithms from temporal logics like LTL to stream monitoring, allowing the computation of richer verdicts than Booleans (quantitative values or even arbitrary data). The core of SRV algorithms is a clean separation between temporal dependencies and data computations. In spite of this theoretical separation previous engines include ad-hoc implementations of just a few data types, requiring complex changes in the tools to incorporate new data types.
In this paper we present a solution as a Haskell embedded domain specific language that is easily extensible to arbitrary data types. The solution is enabled by a technique, which we call *lift deep embedding*, that consists in borrowing general Haskell types and embedding them transparently into an eDSL. This allows for example the use of higher-order functions to implement static stream parametrization. We describe the Haskell implementation called hLola and illustrate simple extensions implemented using libraries, which would require long and error-prone additions in other ad-hoc SRV formalisms.

1 Introduction

In this paper we study the problem of implementing a truly generic Stream Runtime Verification (SRV) engine, and show a solution using an embedded domain specific language (eDSL) based on borrowing very general types from the host language into the SRV language and then applying a deep embedding.

Runtime Verification (RV) [20,26,2] is an area of formal methods for reactive systems that analyses dynamically one trace of the system at a time. Compared to static techniques like model checking [8] RV sacrifices completeness to obtain an applicable and formal extension of testing and debugging. Monitors are generated from formal specifications which then inspect a single trace of execution at a time. Early RV languages were based on logics like LTL [27] or past LTL adapted for finite paths [3,12,21]. Other approaches followed, based on regular

* This work was funded in part by Madrid Regional Government project "S2018/TCS-4339 (BLOQUES-CM)" and by Spanish National Project "BOSCO (PGC2018-102210-B-100)".

expressions [36], rule based languages [1], or rewriting [34]. These specification languages come from static verification, where decidability is key to obtain algorithmic solutions to decision problems like model checking. Therefore, the observations and verdicts are typically Boolean values.

Stream Runtime Verification [10,35] starts from the observation that most monitoring algorithms for logics from static verification can be generalized to richer observations and outcomes (verdicts), by generalizing the datatypes of the individual internal operations of the monitors. Languages for SRV, pioneered by Lola [10], describe monitors declaratively via equations that relate streams of input and streams of output, offering a clean separation between the time dependencies and the concrete operations. The temporal part is a sequence of operations on abstract data, mimicking the steps of the algorithms for temporal logics. Each individual operation can then be performed on a datatype implementation, obtaining monitors for arbitrary data. Offset expressions allow us to refer to stream values in different moments of time, including future instants (that is, SRV monitors need not be causal).

Most previous SRV developments [9,25,18,16] focus on efficiently implementing the temporal engine, promising that the clean separation between time and data allows incorporating off-the-shelf datatypes easily. However, in practice, adding a new datatype requires modifying the parser, the internal representation, and the runtime system that keeps track of offset expressions and partially evaluated expressions. Consequently, these tools only support a limited hardwired collection of datatypes. In this paper, we give a general solution to this problem via a Haskell eDSL, resulting in the language hLola[4], whose engine implements a generic SRV monitoring algorithm that works for general datatypes.

Typically, a DSL is designed as a complete language, first defining the types and terms of the language (this is, the underlying theory), which is then implemented—either as an eDSL or as a standalone DSL—, potentially mapping the types of the DSL into types of the host. However, our intention with hLola is to have a language where datatypes are not decided upfront but can be added on demand without requiring any re-implementation. For this reason, hLola borrows (almost) arbitrary types from the host system and then embeds all these borrowed types, so hLola is agnostic from the stream types (even types added in the future). Even though this technique has been somewhat part of the folklore of modern Haskell based eDSLs (e.g. [40]), this is a novel approach to build runtime verification engines. We called this technique a *lift deep embedding*, which consists of (1) lifting the types and values of the host language into the generic DSL using generic programming, and (2) deep embedding the resulting concrete DSL into the host language. This technique allows us to incorporate Haskell datatypes into hLola, and enables the use of many features from the host language in the DSL. For example, we use higher-order functions to describe transformations that produce stream declarations from stream declarations, obtaining static parameterization for free. In turn, libraries collect these transformers, which allows defining in a few lines new logics like LTL, MTL, etc.

[4] available open source at `http://github.com/imdea-software/hlola`

or quantitative semantics for these logics. Haskell type-classes allow to implement *simplifiers* which can compute the value of an expression without resolving all its sub-expressions first. If the unevaluated sub-expressions contain future offset references, the engine may anticipate verdicts ahead of time. Implementing many of these in previous SRV systems has required to re-invent and implement features manually (like macro expansions or ad-hoc parameterization). We use polymorphism both for genericity (to simplify the engine construction) and to enable the description of generic stream specifications, which, again, is not allowed by previous SRV engines. Finally, we also exploit features present in Haskell to offer IO for many stream datatypes for free.

Related work. SRV was pioneered by Lola [10] for monitoring synchronous systems only supporting Integers and Booleans. Copilot [31] is a Haskell implementation that offers a collection of building blocks to transform streams into other streams, but Copilot does not offer explicit time accesses (and in particular future accesses). Lola 2.0 [16] extends Lola with special constructs for runtime parameterization and real-time features. TeSSLa [9] and Striver [18] are two modern SRV approaches that target real-time event streams. All these languages still support only limited hard-wired datatypes.

RV and SRV overlap with other areas of research. Synchronous languages –like Esterel [5] or Lustre [19]– are based on data-flow. These languages force causality because their intention is to describe systems and not observations or monitors, while SRV removes the causality assumption allowing the reference to future values. In Functional Reactive Programming (FRP) [13] reactive behaviors are defined using the building blocks of functional programming. An FRP program describes a step of computation, a reaction when new input values (events) are available, thus providing implicitly the dependency of the output streams at the current point from input streams values. Again, the main difference is that FRP programs do not allow explicit time references and that the dependencies are causal (after all, FRP is a programming paradigm). In comparison, FRP allows immediately all the features of the programming language without needing the solution proposed in this paper. It would be interesting to study the opposite direction that we solve in this paper: how to equip FRP with explicit times and non-causal future references. Also, FRP does not typically target resource calculation, while this is a main concern in RV (and in SRV).

Contributions. In summary, the contributions of the paper are: (1) An implementation of SRV, called hLola, based on an eDSL that exploits advanced features of Haskell to build a generic engine. A main novelty of hLola as an SRV implementation is the use of a lift deep embedding to gain very general types without costly implementations. Section 3 describes the runtime system of hLola. (2) An implementation of many existing RV specification languages (including LTL, MT-LTL and MTL) in hLola, which illustrates the simplicity of extending the language. This is shown in Section 4. (3) A brief empirical evaluation, which suggests that the hLola engine executes using only the theoretically predicted resources, shown in Section 5.

2 Preliminaries

We briefly introduce SRV using Lola (see [35]) and then present the features of Haskell as a host language that we use to implement hLola.

2.1 Stream Runtime Verification: Lola

Intuitively speaking, Lola is a specification language and a monitoring algorithm for synchronous systems. Lola programs describe monitors by expressing, in a declarative manner, the relation between output streams and input streams. Streams are finite sequences of values, for example, a Boolean stream is a sequence of Boolean values. The main idea of SRV is to cleanly separate the temporal dependencies from the data computation.

For the data, monitors are described declaratively by providing one expression for each output stream. Expressions are terms from a multi-sorted first order theory, given by a first-order signature and a first-order structure. A theory is a finite collection of interpreted *sorts* and a finite collection of interpreted function symbols. Sorts are interpreted in the sense that each sort is associated with a *domain*, for example the domain of sort *Bool* is the set of values $\{true, false\}$. For the purpose of this paper we use sorts and types interchangeably, as we use Haskell types to implement Lola sorts. Function symbols are interpreted, meaning that f is both (1) a constructor to build terms; and (2) a total function (the interpretation) used to evaluate and obtain values of the domain of the return sort. For example, natural numbers uses two sorts (*Nat* and *Bool*), constant function symbols 0, 1, 2, ... of sort *Nat*, and *True* and *False* of type *Bool*, as well as functions $+$, $*$, $\cdots$ $Nat \times Nat \to Nat$ and predicates $<$, $\leqslant$, ..., that are symbols that return *Bool*. We assume that our theories include equality, and also that for every sort T there is a ternary function `if · then · else ·` that returns a value of sort T given a Boolean and two arguments of sort T. We use $e : T$ to represent that e has sort T.

Given a set Z of (typed) *stream variables*, *offset expressions* are $v[k, d]$ where v is a stream variable, $d : T$ is a constant and k is an integer number. For example, $x[-1, false]$ is an *Bool* offset expression and $y[+3, 5]$ is a *Nat* offset expression. The intended meaning of $v[k, d]$ is to represent, at time n, the value of the stream v at time $n+k$. The second argument d indicates the default value to be used beyond the time limits. When it is clear from the context, we use v to refer to the offset expression $v[0]$ (that does not need a default value). The set of *stream expressions* over a set of variables Z (denoted $Expr(Z)$) is the smallest set containing Z and all offset expressions of variables of type Z, that is closed under constructor symbols of the theory used. For example $(x[-1, false] \vee x)$ and $(y + y[+3, 5] * 7)$ are stream expressions.

A Lola *specification* consists of a set $\{s_1, s_2 \ldots\}$ of input stream variables and a set $\{t_1, t_2 \ldots\}$ of output stream variables, and one *defining expression* $t_i = exp_i$ per output variable over the set of input and output streams, including t_i itself.

Example 1. The following is a LOLA specification with input stream variable $s : Bool$ and output stream variable $once\_s : Bool$:

```
input  bool s
output bool once_s = once_s [-1,false] || s
```

This example corresponds to the LTL formula $\diamondsuit s$. The following specification counts how many times s was *True* in the past (`toint` is the function that returns 0 for *False* and 1 for *True*):

```
output int n_once_s = n_once_s [-1,0] + toint(s)
```

A valuation of a specification associates a stream of length N to each of its stream variables, all of which are of the same length. Given a stream σ_i for each input stream variable s_i and a stream τ_i for each output stream variable t_i in a specification, every expression e can be assigned a stream $[\![e]\!]$ of length N. For every $j = 0 \dots N-1$:

- $[\![c]\!](j) = c$ for constants;
- $[\![s_i]\!](j) = \sigma_i(j)$ and $[\![t_i]\!](j) = \tau_i(j)$ for stream variables;
- $[\![f(e_1, \dots, e_n)]\!](j) = f([\![e_1]\!](j), \dots, [\![e_n]\!](j))$; and
- $[\![v[k, d]]\!](j) = [\![v]\!](j + k)$ if $0 \leqslant j + k < N$, and $[\![v[k, d]]\!](j) = d$ otherwise.

We say that a valuation is an evaluation model, if $[\![t_i]\!] = [\![e_i]\!]$ for each output variable t_i, that is, if every output stream satisfies its defining equation. The dependency graph is the graph of offset dependencies between variables, and can be used to rule out cycles in specifications to guarantee that every specification has a unique output for every input.

One very important aspect of SRV is its ability to analyze specifications and automatically calculate the necessary resources. A monitor is *trace-length independent* if it operates with an amount of memory (and of processing time per input event) that does not depend on the length of the trace. Many logics admit trace-length independent algorithms for their past fragments, like for example LTL and TLTL [3] and MTL [38]. The notion of *efficient monitorability* in SRV [10,35], defined as the absence of positive (future) cycles in the dependency graph, guarantees a trace-length independent monitor. The dependency graph can also be used to build efficient runtime systems, by determining when a value stream variable is guaranteed to be resolved (the latency) and when a value can be removed because it will no longer be necessary (the back-reference). See [35] for longer formal definitions.

2.2 Haskell as a host language for an eDSL

An *embedded Domain Specific Language*[23] (eDSL) is a DSL that can be used as a language by itself, and also as a library in its host programming language. An eDSL inherits the language constructs of its host language and adds domain-specific primitives. In this work we implemented HLOLA as an eDSL in Haskell. In particular, we use Haskell's features as host language to implement static parameterization (see Section 3.4), a technique that allows the programmatic

definition of specifications. This is used to extend hLola to support many temporal logics proposed in RV. Other SRV implementations, in their attempt to offer expressive data theories in a standalone tool, require a long and costly implementation of features that are readily available in higher-order expressive languages like Haskell. Using an eDSL, we can effectively focus our development efforts on the temporal aspects of Lola.

We describe in the next section the *lift deep embedding*, which allows us to lift Haskell datatypes to Lola and then to perform a single deep embedding for all lifted datatypes. This technique fulfills the promise of a clean separation of time and data and eases the extensibility to new data theories, while keeping the amount of code at a minimum. Additionally, using eDSLs brings benefits beyond data theories, including leveraging Haskell's parsing, compiling, type-checking, and modularity. The drawback is that specifications have to be compiled with a Haskell compiler, but once a specification is compiled, the resulting binary is agnostic of the fact that an eDSL was used. Therefore, any target platform supported by Haskell can be used as a target of hLola. Moreover, improvements in the Haskell compiler and runtime systems will be enjoyed seamlessly, and new features will be ready to be used in hLola.

Haskell [28] is a pure statically typed functional programming language that has been reported to be an excellent host language for eDSLs [17]. Functions are values, and function application is written simply as a blank space without parentheses, which helps eDSLs look cleaner. Haskell also allows custom parametric polymorphic datatypes, which eases the definition of new data theories, and enables us to abstract away the types of the streams, effectively allowing the expression of generic specifications.

Haskell's ecosystem provides a plethora of frameworks for generic programming [22]. In particular, our engine implementation uses the *Typeable* class to incorporate new types without modification. However, we do not perform any kind of traversal over generic data, we employ the *Typeable* class as a mechanism to hide concrete types and implement heterogeneous lists. Members of the *Typeable* class have an associated type representation, which can be compared, and therefore employed to define a *Dynamic* datatype (which hides a *Typeable* datatype), and to define a type-safe cast operation. New datatypes developed by the active Haskell community can be incorporated immediately into hLola. The datatype members of the *Typeable* class encompass all sorts that are used in practice in SRV.

Haskell is declarative and statically typed, just like Lola. In Lola, functions are functions in the mathematical sense, that is, they do not have side effects. Lola does not make assumptions about when these functions will be called, and guarantees that a function yields the same result when applied to the same arguments twice. This is aligned with the Haskell purity of (total) functions.

Another feature that improves syntax readability is Haskell type classes, which allows overloading methods. We can redefine functions that are typically native in other languages, such as Boolean operators ($\vee$) and ($\wedge$), and the arithmetic operators ($+$), ($-$) and ($*$), as well as define and use custom infix oper-

ators. Such definitions are possible by extensions made by the de-facto Haskell compiler, GHC [29]. Haskell has let-bindings, list comprehensions, anonymous functions, higher-order, and partial function application, all of which improves specification legibility. Finally, hLola uses Haskell's module system to allow modular specifications and to build language extensions.

3 Implementation

3.1 Language design

We model input and output stream variables using:

- *Input Stream* declarations, which model Lola's input variables simply as a *name*. During evaluation, the engine can look up a name in the environment and fetch the corresponding value at a required time instant.
- *Output Stream* declarations, which model output streams in Lola. These declarations bind the name of the stream with its *Expression*, which represents the defining expression of a Lola output stream.

Revisiting the Lola specification in Ex. 1, in hLola, s will be an *Input Stream* declaration and *once_s* an *Output Stream* declaration.

We seek to represent many theories of interest for RV and to incorporate new ones transparently, so we abstract away concrete types in the eDSL. For example, we want to use the theory of *Boolean* without adding the constructors that a usual deep embedding would require. To accomplish this goal we revisit the very essence of functional programming. Every expression in a functional language—as well as in mathematics—is built from two basic constructions: *values* and *function applications*. Therefore, to implement our SRV engine we use these two constructions, plus two additional stream access primitives to capture offset expressions. The resulting datatype is essentially a de-functionalization [33] of the applicative interface. There is a limitation that some Haskell datatypes cannot be handled due to the use of *Dynamic* and *Typeable*, which we introduce within the engine to get a simple way to implement generic programming while preserving enough structure. However, this is not a practical limitation to represent theories and sorts of interests for monitoring.

We define expressions in Haskell as a parametric datatype *Expr* with a polymorphic argument *domain*. An $e :: Expr\ domain$ represents an expression e over the domain *domain*. The generic *domain* is automatically instantiated at static time by the Haskell compiler, effectively performing the desired lifting of Haskell datatypes to types of the theory in hLola. For example, the use of *Expr Int* will make the compiler instantiate *domain* as *Int*. The resulting concrete *Expressions* constitute a typical deeply embedded DSL. We call this two step technique a *lift deep embedding*. This technique avoids defining a constructor for all elements in the data theory, making the incorporation of new types transparent.

Here we present in more detail the *Expr* construction in Haskell. The first two constructors (**Leaf** and **App**) are the *data constructions* of the language, which are aligned with the notions of de-functionalization mentioned above, and

allow encoding terms from a LOLA theory seamlessly. The other two constructors (**Now** and (**:@**)) represent the offset expressions:

- The constructor **Leaf** :: *Typeable a* ⇒ *a* → *Expr a* models an element of the theory.
- The constructor **App** :: (*Typeable a*, *Typeable b*, *Typeable* (*b* → *a*)) ⇒ *Expr* (*b* → *a*) → *Expr b* → *Expr a* represents the application of a *function Expression* to a *value Expression*.
- A term **Now** :: *Stream a* → *Expr a* represents the value of a stream in the current instant.
- The *at* infix constructor, (**:@**) :: *Stream a* → (*Int*, *Expr a*) → *Expr a* models future and past offset expressions, specifying the default value to use if the access falls off the trace

These constructions allow us to lift operations from domain values to *Expressions* directly. For example, we can create an *Expression* that represents the sum of two *Expr Int* without defining a dedicated type of *Expression*.

Similarly, we define the *Stream* declarations in Haskell as a parametric datatype *Stream* with a polymorphic argument *domain*.

- The **Input** :: (*FromJSON a*, *Read a*, *Typeable a*) ⇒ *String* → *Stream a* constructor represents an input stream, and associates the name of the stream to the type of its values.
- The **Output** :: *Typeable a* ⇒ (*String*, *Expr a*) → *Stream a* constructor represents an output stream, and associates the name of the stream to the type of its values and its defining *Expression*, of the same type.

The LOLA specification from Ex. 1 can be written in HLOLA as follows:

```
once_s :: Stream Bool
once_s = Output ("once_s", App (App (Leaf (∨)) prevOnce_s) (Now s))
  where s = Input "s"
        prevOnce_s = once_s :@ (−1, False)
```

The expression of *once_s* takes the application of the (data theory) function (∨) to the value of *once_s* at −1, using *False* as the default value, and applying the result to the current value of *s*. We define the infix operator (**=:**) that builds an output stream from a name and an expression, and override the Boolean operator ∨; and the HLOLA *Output Stream* declaration looks almost like a LOLA expression:

```
once_s = "once_s" =: once_s :@ (−1, False) ∨ Now s
```

3.2 Static analysis

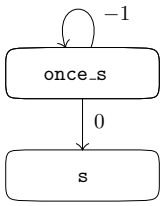
Not every grammatically correct LOLA specification is valid. Some errors like syntactic errors, missing references and type mismatches can be checked by the Haskell compiler. But to guarantee that a specification is well-defined we also need to examine the dependency graph to check that it does not contain closed paths of weight zero. This will ensure that the value of a stream at any point does not depend on itself. We first convert every *Expression a* and *Stream a* to

their equivalent *Expression* and *Stream* declaration of *Dynamic*, so *Stream* declarations and *Expressions* of different types can be mixed and used in the same specification. Then we obtain the dependency graph by traversing the stream definitions in the specification recursively. One drawback of this approach is that the Haskell type-system can no longer track the original type of an expression, but this step is made after Haskell has type-checked the specification, guaranteeing that the engine is forgetting the type information of a well-typed specification. The engine keeps the information on how to parse the input streams and how to show output values given a stream name, safely casting from and into *Dynamic*, and avoiding type mismatches when converting from dynamically-typed objects. We make the following claim:

Claim. Every conversion from a Dynamic *Expression* within the HLOLA engine returns a value *Expression* of the expected type.

The proof of this claim can be done using Liquid Haskell [39] and is ongoing research beyond the scope of this paper. Assuming the claim above, a runtime type error can only be produced when processing an input event whose value is not of the expected type.

During this stage, the tool also calculates the minimum weight of the paths in the dependency graph, a non-positive value that we call *minimum back reference* and note *minBackRef*, along with the maximum weight of the edges, which we call *latency* and note *maxLatency*. The dependency graph of the specification in Ex. 1 is shown on the right. The *minBackRef* is -1, because *once_s* depends on the previous value of itself, and the *maxLatency* is 0 because there are no references to future values of streams. The values of *minBackRef* and *maxLatency* indicate that the engine will only keep the values of the streams at the present and previous instants.



3.3 Runtime System

We now describe some key internal datatypes used in the implementation of the execution engine. An *Instant* is a map that binds the name of a stream to an *Expression*. Given a specification with m streams $s_1, \ldots, s_m$, an *Instant* can be interpreted as a vector of size m. A *Sequence* is an ordered collection of *Instant*s, one of which is said to be "in focus". The *Instant*s in the past of the one in focus are stored in the *Sequence* in an array of size $(maxLatency - minBackRef)$, which limits the amount of memory that the engine can consume. On the other hand, the *Instant*s in the future of the one in focus are stored as a list. Even though this list can be (implicitly) as long as the full trace, the elements in the list will not actually exist in memory until they are needed to compute a value, due to the laziness of Haskell evaluation. We can think of a *Sequence* as a matrix of expressions, where each column is an *Instant* vector, and one of them is in focus. The evaluation of a specification with m streams over n instants is conceptually an $n \times m$ matrix.

Given a specification and a list of values, we first create a *Sequence* with an empty past and the focus on the first instant. In this *Sequence*, the value of the cell (s_i, n) in the *Sequence* matrix for an *input* stream s_i and instant n, is a **Leaf** containing the value read for the stream of s_i at time instant n. Similarly, the value of every *output* stream t_j and instant n is the defining *Expression* for t_j in the specification, waiting to be specialized and evaluated. Note that these values do not actually exist in memory until they are needed. The goal of the engine is to compute a **Leaf** expression (this is, a ground value) at every position in the matrix, particularly for output streams.

Starting from the initial state, the engine will solve every output stream at the instant in focus, and then move the focus one step forward. This algorithm guarantees that all elements in the past of the focus are leaves. The figure on the right illustrates the *Sequence* of an execution at time instant 3, where some of the output expressions $e_{1,3} \dots e_{m,n}$ can be leaves too. At the end of the execution, the focus will be on the last column of the matrix, and all the elements in the matrix will be leaves.

$$
\begin{array}{c}
\begin{array}{cccccc} & 1 & 2 & 3 & \dots & n \end{array} \\
\begin{array}{c} s_1 \\ \vdots \\ s_k \\ t_1 \\ \vdots \\ t_m \end{array}
\begin{pmatrix}
\mathbf{Leaf}_{1,1} & \mathbf{Leaf}_{1,2} & \mathbf{Leaf}_{1,3} & \dots & \mathbf{Leaf}_{1,n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\mathbf{Leaf}_{k,1} & \mathbf{Leaf}_{k,2} & \mathbf{Leaf}_{k,3} & \dots & \mathbf{Leaf}_{k,n} \\
\mathbf{Leaf}_{k+1,1} & \mathbf{Leaf}_{k+1,2} & e_{1,3} & \dots & e_{1,n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\mathbf{Leaf}_{k+m,1} & \mathbf{Leaf}_{k+m,2} & e_{m,3} & \dots & e_{m,n}
\end{pmatrix} \\
\Delta
\end{array}
$$

The output streams will be calculated and output incrementally while new data is retrieved for the input streams. The engine will block when it needs the value of an input stream that has not been provided yet. These characteristics of the Haskell runtime system allow the monitor to run online processing events from the system under analysis on the fly, or offline over dumped traces.

A language that offers means to define new datatypes must not only provide the constructs to define them, but it also must implement the encoding and decoding of custom datatypes. Extensible encoding and decoding of datatypes in the theory is not trivial and might account for a large portion of the codebase. As an eDSL, hLola can rely upon Haskell's facilities to define how to encode and decode *Typeable* datatypes, sometimes even automatically from their definitions. This class encompasses many of the datatypes that are used in practice to encode values (observations and verdicts) when monitoring systems.

Input events are fed to hLola in JSON format, where each line is a string representation of a JSON object with one field per input stream. The types of the input streams have to be instances of the *FromJSON* class, meaning that a value of the corresponding type can be constructed from a serialized JSON *Object*. Output streams must be instances of the *ToJSON* class, which means that we can get a JSON *Object* from a value of the corresponding type.

Haskell allows defining custom datatypes via the **data** statement. Once defined, these types can be used just like any other type in Haskell. Most of the times, we can use Haskell's **deriving** mechanism to make our custom types instances of the corresponding classes, if needed. Section 4 contains examples of custom datatypes for input values.

3.4 Additional Features

The use of Haskell as a host language eases the implementation of many useful features of SRV in HLola. We show here two examples: anticipation and parameterized streams.

Anticipation Input event streams represent the trace of observations of a system, and output streams encode a property to be evaluated dynamically. The principle of *anticipation*, as presented in [11], states that once every (infinite) continuation of a finite trace leads to the same verdict, then the finite trace can safely evaluate to this verdict. This principle can be trivially implemented when functions know all their arguments, but it is not always possible to anticipate what the output of the function will be when some of the arguments will only be known in the future. Nevertheless, there are cases where a function can be evaluated with just a subset of its arguments. This property of some functions can be used to compute their values as soon as all the relevant information is retrieved, avoiding waiting for input values that are not strictly necessary to evaluate the function. This idea effectively brings us closer to strict anticipation as defined above.

The circumstances under which a function can be computed with missing arguments is data-specific information. Typical SRV implementations provide simplifications for some functions in the covered theories, but do not offer a way to provide new simplifications to their theories. Instead, we provide a framework to keep the simplifications extensible. To allow the use of functions off-the-shelf as well as simplifiable functions, we define a new datatype and a class of which the Haskell function constructor $(\rightarrow)$ is an instance, shown below:

data *LFunction a b* = *Pure* $(a \rightarrow b)$ | *Simplifier* $(Maybe\ a \rightarrow Maybe\ b)$
class *ILFunction x* **where** *toLFunction* $:: x\ a\ b \rightarrow LFunction\ a\ b$
instance *ILFunction* $(\rightarrow)$ **where** *toLFunction* = *Pure*

We then generalize the type of the function application constructor **App** :: *Expression* $(f\ b\ a) \rightarrow$ *Expression* $b \rightarrow$ *Expression* a, under the constraint that f be a member of the class *ILFunction*. In this way, users of the eDSL can define their own simplifiable functions using the *Simplifier* constructor, or just use off-the-shelf functions seamlessly; which will automatically be applied the *Pure* constructor by the compiler.

The language is shipped with simplifiers for the Boolean operators $\vee$ and $\wedge$; as well as the `if · then · else ·` operator and some numeric operators. These simplifiers have great impact in temporal logics with references to the future, where values can often be resolved at an instant with the information retrieved up to that point—without the need to wait until all future values are resolved. We show the simplifiers for the `if · then · else ·` operator in the extended version of the paper [6].

Parameterized streams Static parametrization is a feature of some SRV systems which allows instantiating an abstract specification. This is useful to reuse

repetitive specifications and capture the essence of a stream definition, abstracting away the specific values. Section 4 shows how this feature is used to concisely implement several monitoring languages as libraries in HLola. This feature is implemented in Lola2.0 [15] as well as in TeSSLa [9] using an ad-hoc macro feature in the tool chain. Here we show how static parametrization can be obtained directly using Haskell features. Consider again the specification of $\diamondsuit s$ shown in Ex. 1:

$once\_s :: Stream\ Bool$
$once\_s = \texttt{"once\_s"} =: once\_s :@ (-1, False) \vee \mathbf{Now}\ s$

If we want to define a stream to compute $\diamondsuit r$, we would have to define a stream $once\_r$ whose definition is almost identical to the definition of $once\_s$. This leads to code duplication and hard to maintain specifications.

Instead of defining an output stream $once\_s$ specifically for s, we aim to write a general stream *once* parameterized by a Boolean stream. We can use Haskell as a macro system to programmatically define specifications, effectively implementing static parameterization.

Example 2. The definition of *once* in HLola using static parameterization is:

$once :: Stream\ Bool \to Stream\ Bool$
$once\ s = \texttt{"once"} <: s =: once\ s :@ (-1, False) \vee \mathbf{Now}\ s$

Note that we simply abstracted away the occurrences of s. To avoid name clashes among different instantiations of *once*, we concatenate the string `"once"` with the name of the argument stream s, by using the operator $<:$. Static parametrization is used extensively to implement libraries as described in the next section.

4 Extensible libraries in HLola

One of the benefits of implementing an eDSL is that we can reuse the library system of the host language to modularize and organize the code. The Haskell module system allows importing third parties libraries, as well as developing new libraries; HLola ships with some predefined theories and stream-specific libraries. In this section we show an overview of the stream-specific libraries.

Past-LTL. The operators of Past-LTL [4] can be described using the Lola specification language (e.g. $\diamondsuit$ from Ex. 2). Given two Boolean streams p and q, the Boolean stream p ‘*since*‘ q is *True* if q has ever been *True*, and p has been *True* since the last time q became *True*. One can simply **import** *Lib.LTL* and then define streams like: $property = yesterday\ (p$ ‘*since*‘ $q)$.

Example 3. We show an example of a Past-LTL property for a sender/receiver model taken from [4]: $\square(snd.state = waitForAck \to \ominus\boxminus snd.state \neq waitForAck)$. Using HLola, we define a type to represent the possible states of the sender, deriving a *FromJSON* instance to use it as the type of an input stream *sndrState*:

data *SndrState* = *Get* | *Send* | *WaitForAck* **deriving** (*Generic*, *Read*, *FromJSON*, *Eq*)

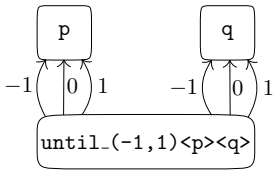
Then, we define the property as a Boolean output stream:

```
sndrState :: Stream SndrState
sndrState = Input "senderState"

sndrNotWaiting :: Stream Bool
sndrNotWaiting = "sndrNotWaiting" =: Now sndrState / == Leaf WaitForAck

prop :: Stream Bool
prop = let sndrWaitingAck = Now sndrState === Leaf WaitForAck
           startedWaiting = yesterday (historically sndrNotWaiting)
       in "prop" =: sndrWaitingAck `implies` Now startedWaiting
```

MTL. Metric Temporal Logic [24] is an extension of LTL with time constraints that give upper and lower bounds on the temporal intervals. The stream *until* is parameterized by two integers, which are the boundaries of the interval, and two Boolean streams to model the formula $p\,\mathcal{U}_{[a,b]}\,q$. We use recursion to programmatically define the *Expression* of *until*, which will be unfolded at compile time for the dependency graph sanity check. This expansion can be observed in the dependency graph of a specification that uses *until*, for example, *property* = *until* $(-1, 1)$ p q, which checks that a stream p is *True* until q is *True* in the interval $(-1, 1)$, which is shown on the right.



In [32], Reinbacher et al. introduce Mission-Time LTL, a projection of LTL for systems which are bounded to a certain mission time. They propose a translation of each LTL operator to its corresponding MTL operator, using $[0, mission_t]$ as the temporal interval, where $mission_t$ represents how the duration of the mission. The ability of HLOLA to monitor MTL can be used to monitor Mission-Time LTL through this translation.

Example 4. We show an example of an MTL property taken from [30]: $\square(alarm \rightarrow (\diamondsuit_{[0,10]} allclear \vee \diamondsuit_{[10,10]} shutdown))$
This property uses MTL to establish deadlines between environment events and the corresponding system responses. In particular, the property assesses that an *alarm* is followed by a *shutdown* event in exactly 10 time units unless *all clear* is sounded first. We consider three Boolean input streams *alarm*, *allclear* and *shutdown*—which indicate if the corresponding event is detected—and define an output stream that captures whether the property holds:

```
alarm = Input "alarm" :: Stream Bool
allclear = Input "allclear" :: Stream Bool
shutdown = Input "shutdown" :: Stream Bool

prop :: Stream Bool
prop = "prop" =: Now alarm `implies` Now willClear ∨ Now willShutdown
  where willClear = eventually (0, 10) allclear
        willShutdown = eventually (10, 10) shutdown
```

5 Implementation and Empirical evaluation

The implementation of hLola requires no code for the parser and type checker, since it reuses those from the Haskell compiler. The table below shows the number of lines for the full hLola implementation.

Language and input		Engine		Syntax		Libraries	
Files: ./	LoC	Files: Engine/	LoC	Files: Syntax/	LoC	Files: Lib/	LoC
DecDyn.hs	87	Engine.hs	176	Booleans.hs	37	LTL.hs	21
InFromFile.hs	51	Focus.hs	39	HLPrelude.hs	3	MTL.hs	29
Lola.hs	62			Num.hs	26	Pinescript.hs	41
StaticAnalysis.hs	78			Ord.hs	18	Utils.hs	13
Total	278	**Total**	215	**Total**	102	**Total**	104

In summary, the core of the tool has 493 lines, while the utils account for 206 lines, giving a total of 699 lines. This compares to the tens of thousands of lines of a parser and runtime system of a typical stand-alone tool. In the rest of this section we summarize how using Haskell enables the use of available tools, and then report on an empirical evaluation of hLola.

Haskell tools. The use of Haskell as a host language allows us to use existing tools to improve hLola specifications, such as LiquidHaskell and QuickCheck.

Liquid Haskell [39] enriches the type system with refinement types that allow more precise descriptions of the types of the elements in a Haskell program. In our case we can use Liquid Haskell to express specifications with more precision. For example, we can prevent a specification that adds the last n elements from being used with a negative n:

```
{- nsum :: Stream Int -> Nat -> Stream Int -}
```

$nsum :: \mathit{Stream\ Int} \to \mathit{Int} \to \mathit{Stream\ Int}$

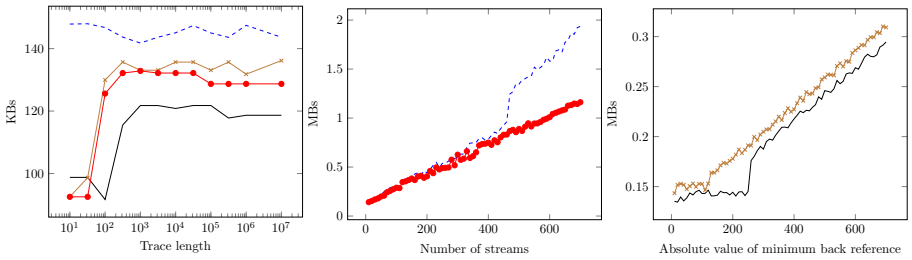
$nsum\ s\ n = \texttt{"n\_sum"} <\!\!: s <\!\!: n =\!\!: nsum\ s\ n$ **:@** $(-1, 0) + \mathbf{Now}\ s - s$ **:@** $(-n, 0)$

Then, given a stream r of type *Stream Int* we can attempt to define a stream s that computes the sum of the last 5 values on stream r as $s = nsum\ r\ 5$. Running LiquidHaskell with `--no-termination` allows the recursive definition of n over this specification, which yields no error, but running LiquidHaskell on $s' = nsum\ r\ (-1)$ produces a typing error.

QuickCheck [7] is a tool to perform random testing of Haskell programs, which we can easily use for hLola specifications. For example, we can assess that the first instant at which a Boolean stream p is *False* is exactly one instant after the last instant at which $\boxminus p$ is *True*, increasing our confidence on the implementation of the Past-LTL $\boxminus$ operator.

Empirical evaluation. We report now on an empirical evaluation performed to assess whether the engine behaves as theoretically expected in terms of memory usage. The hardware platform over which the experiments were run is a MacBook Pro with MacOS Catalina Version 10.15.4, with an Intel Core i5 at 2,5 GHz and 8 GB of RAM.

The first two *Stream* declarations calculate if an input *Boolean* stream p is periodic with period n. This is a simple, yet interesting property to assess in



(a) Memory wrt trace length (b) Memory wrt number of streams (c) Memory wrt *minBackRef*

Fig. 1. Empirical evaluation

embedded systems. We specify this property in two different ways. In the first *Stream* declaration, we define a single stream which compares the current value of p with its value n instants before:

$booleanPeriodWidth :: Int \rightarrow Stream\ Bool$

$booleanPeriodWidth\ n = \texttt{"periodic\_width"} =: \mathbf{Now}\ p === p\ \mathbf{:@}\ (-n, \mathbf{Now}\ p)$

The data of this experiment is represented by the solid, unmarked black curves in Fig. 1 (a) and (c).

In the second *Stream* declaration, we programmatically create $n+1$ streams *carrier* i, with $i = 0 \ldots n$ defined as a function that compares its argument with the value of p i instants before, which is bound by the partially applied equality function:

$booleanPeriodHeight :: Int \rightarrow Stream\ Bool$
$booleanPeriodHeight\ n = \texttt{"periodic\_height"} =: \mathbf{Now}\ (carrier\ n)\ \langle\star\rangle\ \mathbf{Now}\ p$
where
$carrier\ 0 = \texttt{"carrier\_prd"} <: 0 =: (\equiv)\ \langle\$\rangle\ \mathbf{Now}\ p$
$carrier\ n = \texttt{"carrier\_prd"} <: n =: carrier\ (n-1)\ \mathbf{:@}\ (-1, \mathbf{Leaf}\ (const\ True))$

The data of this experiment is represented by the solid, circle-marked red curves in Fig. 1 (a) and (b).

We also run a quantitative version of this n-period checker, whose value is 100 at a given instant if p at that instant is equal to the value of p n instants ago; 50 if it is equal to the value of p $n-1$ or $n+1$ instants ago; 25 if it is equal to the value of p $n-2$ or $n+2$ instants ago; and 0 otherwise. Note that this specification has a value closer to 100 when the specification is closer to being periodic, and closer to 0 when the specification is further from being periodic. This example illustrates how HLola can be used to define quantitative semantics of temporal logics, which is an active area of research in Runtime Verification. In this case we also define a version with a single stream (represented by the solid, cross-marked brown curves in Fig. 1 (a) and (c)), and a version with auxiliary streams, each of which has an offset of -1 at most (represented by the dashed blue curves in Fig. 1 (a) and (b)).

In the first experiment, we run all four specifications over traces with synthetic inputs of varying length. The results are shown in Fig. 1 (a), which suggest that the memory required is approximately constant, indicating that the mem-

ory used is independent of the trace length, and that monitors run in constant space, as theoretically predicted.

In the second experiment, we vary the period n to asses how the number of streams affects the memory usage for both period checkers. The outcome suggests that increasing the number of streams only impacts linearly on the memory required to perform the monitoring, as shown in Fig. 1 (b).

In the third experiment, we use different values for the period n to increase the absolute value of the *minBackRef* for the Boolean and quantitative period checkers to asses how increasing the absolute value of the *minBackRef* affects the memory required. The outcome again suggests that the memory required grows linearly, as shown in Fig. 1 (c). In both the second and third scenarios, we can observe that the memory required is unaffected by whether we are working with quantitative datatypes or *Boolean* values.

6 Final Discussions, Conclusion and Future work

Final discussions One alternative to *Typeable* is to use modular datatypes and evaluators [37]. However, this would break our goal of transparently borrowing datatypes in the lift deep embedding, by forcing HLOLA data sorts to be defined manually as Haskell datatypes.

Resource analysis is a central concern in RV and, in fact, in all real-time and critical systems. For example, aviation regulation forbids the use of runtime environments with garbage collection for critical systems. But this is still an option for soft-critical applications, where HLOLA has successfully been applied to improve mission software of autonomous UAVs [41]. As future work we plan to generate embedded C code from a restricted version of HLOLA, the Ivory framework [14] (see Copilot [31]).

An eDSL like HLOLA is a library within the host language, and can be used as a theory within HLOLA reflectively. This feature can greatly simplify writing specifications, used for example to express predictive Kalman filters as in [41] or quantitative semantics of STL and MTL.

Conclusions We have presented HLOLA, an engine for SRV implemented as a Haskell eDSL. We use the notion of lift deep embedding—folklore in advanced eDSLs (see [40])—in a novel way to fulfill the SRV promise of a clean separation between the temporal engine and the data manipulated, allowing the transparent incorporation of new types. Using Haskell makes readily available features like static parameterization—which allows implementing many logics with Boolean and quantitative semantics—, otherwise programmed in an ad-hoc manner in other SRV tools. The resulting system HLOLA is very concise. A well-known drawback of using an eDSL is that errors are usually cryptic. We are currently working on a front-end restriction of the language that enables better error reporting, while still allowing expert users to use all the advanced features.

Current work includes extending HLOLA to support time-stamped event streams, which allows monitoring real-time event sequences as in [18]. This extension will be to Striver [18] like HLOLA is to LOLA. From the point of view

of exploiting Haskell further, future work includes using LiquidHaskell more aggressively to prove properties of specifications and memory bounds, as well as proving formally the claim that our use of *Dynamic* is safe. We are also working on using QuickCheck to generate test traces from specifications and on studying how to use model-based testing to improve the test suites obtained.

References

1. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proc. of VMCAI'04*, LNCS 2937, pages 44–57. Springer, 2004.
2. E. Bartocci and Y. Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*. Springer, 2018.
3. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM T. Softw. Eng. Meth.*, 20(4):14, 2011.
4. M. Benedetti and A. Cimatti. Bounded model checking for past LTL. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pages 18–33. Springer, 2003.
5. G. Berry. *Proof, language, and interaction: essays in honour of Robin Milner*, chapter The foundations of Esterel, pages 425–454. MIT Press, 2000.
6. M. Ceresa, F. Gorostiaga, and C. Sanchez. Declarative stream runtime verification (hLola), 2020.
7. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of ICFP'00*, pages 268–279. ACM, 2000.
8. E. M. Clarke, O. Grunberg, and D. A. Peled. *Model checking*. MIT Press, 1999.
9. L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma. TeSSLa: Temporal stream-based specification language. In *Proc. of SBMF'18*, volume 11254 of *LNCS*. Springer, 2018.
10. 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 *Proc. of TIME'05*, pages 166–174. IEEE, 2005.
11. W. Dong, M. Leucker, and C. Schallhart. Impartial anticipation in runtime-verification. In *Proc. of ATVA'08*, volume 5311 of *LNCS*, pages 386–396. Springer, 2008.
12. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. of CAV'03*, volume 2725 of *LNCS 2725*, pages 27–39. Springer, 2003.
13. C. Eliot and P. Hudak. Functional reactive animation. In *Proc. of ICFP'07*, pages 163–173. ACM, 1997.
14. T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury. Guilt free ivory. *SIGPLAN Not.*, 50(12):189–200, Aug. 2015.
15. P. Faymonville, B. Finkbeiner, S. Schirmer, and H. Torfah. A stream-based specification language for network monitoring. In *Proc. of RV'16*, volume 10012 of *LNCS*, pages 152–168. Springer, 2016.
16. P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, and T. Hazem. StreamLAB: Stream-based monitoring of cyber-physical systems. In *Proc. of CAV'19*, volume 11561 of *LNCS*, pages 421–431. Springer, 2019.
17. A. Gill. Domain-specific languages and code synthesis using Haskell. *CACM*, 57:42–49, 06 2014.
18. F. Gorostiaga and C. Sánchez. Striver: Stream runtime verification for real-time event-streams. In *Proc. of RV'18*, volume 11237 of *LNCS*, pages 282–298. Springer, 2018.

19. N. Halbwachs, P. Caspi, D. Pilaud, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *Proc. of POPL'87*, pages 178–188. ACM Press, 1987.
20. K. Havelund and A. Goldberg. Verify your runs. In *Proc. of VSTTE'05*, LNCS 4171, pages 374–383. Springer, 2005.
21. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Proc. of TACAS'02*, LNCS 2280, pages 342–356. Springer, 2002.
22. R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*, pages 72–149. Springer, 2007.
23. P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), Dec. 1996.
24. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time Systems*, 2(4):255–299, 1990.
25. M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, and A. Schramm. TeSSLa: Runtime verification of non-synchronized real-time streams. In *Proc. SAC'18*, pages 1925–1933. ACM, 2018.
26. M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Logic Algebr. Progr.*, 78(5):293–303, 2009.
27. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.
28. S. Marlow. Haskell language report, 2010.
29. S. Marlow and S. Peyton Jones. *The Glasgow Haskell Compiler*. Lulu, the architecture of open source applications, volume 2 edition, January 2012.
30. J. Ouaknine and J. Worrell. Some recent results in metric temporal logic. In *Proc. of FORMATS'08*, volume 5215 of *LNCS*, pages 1–13. Springer, 2008.
31. L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A hard real-time runtime monitor. In *Proc. of RV'10*, LNCS 6418. Springer, 2010.
32. T. Reinbacher, K. Rozier, and J. Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Proc. of TACAS'14*, volume 8413 of *LNCS*. Springer, 2014.
33. J. C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(2):363–397, 1998.
34. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.
35. C. Sánchez. Online and offline stream runtime verification of synchronous systems. In *Proc. of RV'18*, volume 11237 of *LNCS*, pages 138–163. Springer, 2018.
36. K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. *ENTCS*, 89(2):226–245, 2003.
37. W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.
38. P. Thati and G. Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, pages 145–162, 2005.
39. N. Vazou, E. L. Seidel, and R. Jhala. LiquidHaskell: experience with refinement types in the real world. In *Proc. of Haskell'14*, pages 39–51. ACM, 2014.
40. O. Westphal and J. Voigtländer. Implementing, and keeping in check, a DSL used in E-learning. In *Proc. of FLOPS'20*, volume 12073 of *LNCS*, 2020.
41. S. Zudaire, F. Gorostiaga, C. Sanchez, G. Schneider, and S. Uchitel. Assumption monitoring using runtime verification for UAV temporal task plan executions. Under submission, 2020.