HStriver: A Very Functional Extensible Tool for the Runtime Verification of Real-Time Event Streams

Felipe Gorostiaga[1,2,3(✉)] and César Sánchez[1]

[1] IMDEA software institute, Madrid, Spain
{felipe.gorostiaga,cesar.sanchez}@imdea.org
[2] Universidad Politécnica de Madrid, Madrid, Spain
[3] CIFASIS, Rosario, Argentina

Abstract. We present HStriver, an extensible stream runtime verification tool for event streams. The tool consists of a runtime verification engine for (1) real-time events streams where individual observations and verdicts can occur at arbitrary times, and (2) rich data in the observations and verdicts. This rich setting allows, for example, encoding as HStriver specifications quantitative semantics of logics like STL, including different notions of robustness.

The keystone of stream runtime verification (SRV) is the clean separation between temporal dependencies and data computations. To encode the data values and computations involved in the monitoring process we borrow (almost) arbitrary data-types from Haskell. These types are transparently lifted to the specification language and incorporated in the engine, so they can be used as the types of the inputs (observations), outputs (verdicts), and intermediate streams. The resulting extensible language is then embedded, alongside the temporal evaluation engine (which is agnostic to the types) into Haskell as an embedded Domain Specific Langauge (eDSL). Morever, the availability of functional features in the specification language enables the direct implementation of desirable features in HStriver like parametrization (using functions that return stream specifications), etc. The resulting tool is a flexible and extensible stream runtime verification engine for real-time streams. We illustrate the use of the tool on many sophisticated real-time specifications, including realistic signal temporal logic (STL) properties of existing designs.

1 Introduction

Runtime Verification [4,25,29] is a lightweight dynamic technique for systems reliability that studies (1) how to generate monitors from formal specifications, and (2) algorithms to monitor the system under analysis, by processing one trace

The tool is available open source at http://github.com/imdea-software/hstriver. This work was funded in part by the Madrid Regional Government under project "S2018/TCS-4339 (BLOQUES-CM)", by Spanish National Project "BOSCO (PGC2018-102210-B-100)".

M. Huisman et al. (Eds.): FM 2021, LNCS 13047, pp. 563–580, 2021.
https://doi.org/10.1007/978-3-030-90870-6_30

at a time. Early RV formal specification languages were based on temporal logics like past LTL [31] adapted to finite traces [7,13,26], regular expressions [36], rewriting [34], fix-point logics [1], rule based languages [3]. In these languages, verdicts (and many times observations) are Boolean, because these logics were borrowed from static verification—where decidability is crucial.

Stream runtime verification (SRV) [12,35] attempts to generalize these monitoring algorithms to richer datatypes, including in observations and verdicts, which allows the computation of quantitative values and summaries, the computation of witnesses, models or the collection of representative data, etc. The keystone of SRV is to cleanly separate the temporal engine from the individual data operations, abstracting the temporal monitoring algorithms which can then be instantiated with generic data types. SRV offers declarative specifications where offset expressions allow accessing streams at different moments in time, including future instants. The first SRV developments [12] were based on similarly synchronous languages—like Esterel [8] or Lustre [21]. 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. Synchronous SRV languages have been extended in recent years to event-based systems for monitoring real-time event streams [10,15,18,20,28]. Most SRV efforts to date, synchronous and event-based, have focused on efficiently implementing the temporal engine, only offering a handful of hard-wired data-types. However, in practice, adding a new datatype requires modifying the parser, the internal representation and the runtime system, which becomes a cumbersome activity. More importantly, these tools are shipped as monolithic tools with a few hard-wired datatypes which the user of the tool cannot extend.

In this paper we describe the tool HStriver, an extensible implementation of an event-based SRV language. The core language is based on [18], and enables the extensions to arbitrary datatypes, implemented as an embedded DSL in Haskell. There are other RV tools implemented as eDSLs [2,24,37] but a main novelty of HStriver is the use of *lift deep embedding*, that allows borrowing Haskell types transparently and embedding the resulting language back into Haskell [9].

Most of the HStriver datatypes were introduced after the temporal engine was completed without any re-implementation, so users of the tool can also extend the datatypes easily. The second contribution of HStriver is the implementation of a novel efficient asynchronous engine for the temporal part, described in Sect. 2. Implementing HStriver as a Haskell eDSL enables the use of higher-order functions which in turn allows writing code that produces stream declarations from stream declarations, enabling features like stream parametrization, which requires costly ad-hoc implementations in previous tools. This is used to package HStriver libraries that describe logics like STL, etc. with Boolean and quantitative semantics in a few lines.

Related work. There have been runtime verification tools for the monitoring of event-based streams (see [14] for a survey). R2U2 [33] is based on a variation of metric interval temporal logic (MITL) for finite (real-time) traces. Since R2U2 uses logic as a specification formalism, the observations and verdicts are

based on Boolean values. BeepBeep [22,23] is a framework to build runtime verification tools based on connecting streaming blocks. Even though BeepBeep could be used as a programming framework for tools like HStriver, in comparison to HStriver, BeepBeep lacks semantics both in terms of the data-types, the assumptions on the temporal domain and lacks a way to compute the resources needed. MonPoly [5,6] is a monitoring tool based on first-order MITL. Even though the tool can produce witnesses for the quantifiers, in comparison with SRV, FO-MITL cannot compute values of arbitrary data-types like the computation of statistics and quantitative semantics of logics. Copilot [32] is a Haskell implementation that offers a collection of building blocks to transform streams, but Copilot does not offer explicit time accesses and offsets (and in particular future accesses). Also, Copilot is based on synchronous time. The closest tools to HStriver are RTLola [15,16] and TeSSLa [10] which are SRV tools extending Lola [12] with capabilities to real-time event streams. The main difference are that RTLola and TeSSLa come with a predefined collection of data-types, while HStriver enjoys the Haskell capabilities to import and create new types without changing HStriver. Also, HStriver incorporates an asynchronous pull engine and borrows flexible data-types and functional features from the host language. Additionally, HStriver allows event-generation, while RTLola is restricted to be event-driven or periodic events. Compared with TeSSLa, HStriver is an explicit timed language while TeSSLa uses stream transformers.

The main contributions of this tool paper are (1) to describe the implementation of HStriver, an SRV tool for real-time event streams using a lift deep embedding in Haskell; (2) the novel pull algorithm to implement an asynchronous temporal engine and; (3) to illustrate many of the HStriver features by example. The rest of the paper is structured as follows. Section 2 introduces SRV and describes the internals of HStriver, Sect. 3 illustrates many features by example, and finally Sect. 4 concludes.

2 The HStriver Tool

SRV generalizes monitoring algorithms to arbitrary data, where data-types are abstracted using multi-sorted first-order interpreted signatures. These data-types are called *data theories* in the SRV terminology. The signatures are interpreted in the sense that every functional symbol `f` used to build terms of a given type is accompanied with an evaluation function f (the interpretation) that allows the computation of values (given values of the arguments). In the context of event-streams, a specification not only needs to declare the values of output streams (based on input and output streams) but also the temporal instants at which there are events in the output streams.

The temporal core of the tool HStriver is based on the Striver specification language [18]. A specification $\langle I, O, E\rangle$ consists of (1) a set of typed input stream variables I, which correspond to the input observations; (2) a set of typed output stream variables O which represent outputs of the monitor and intermediate observations; and (3) defining equations, which associate every output $y \in O$

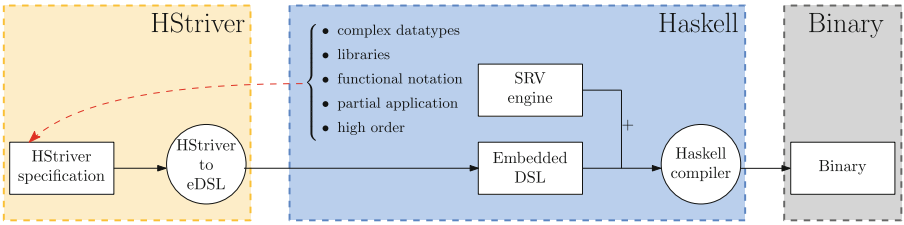


Fig. 1. Software Architecture of HStriver.

with two stream expressions: T_y, which describes when there is an event in y, and E_y which describes what the value is whenever there is an event. Tick expressions T_y are built from constant time-instants, and the union, and shift and delays of the ticking instants of other streams. Stream Expressions E_y are built from constant values, function symbols and offset expressions that allow referring to the previous and next-events in streams, according to the time-stamps of events. The online algorithm proposed in [18] is a *push* algorithm that processes input events in the order of their time-stamps and produces output events also in time order. The algorithm implemented in HStriver is a much more efficient *pull* algorithm, which attempts to compute events in output streams fetching the necessary events from other streams. We have empirically shown that HStriver is capable of processing tens of thousands of events per second [17].

HStriver Architecture. The architecture of HStriver follows the same approach as the tool HLola [19] and is shown in Fig. 1. An HStriver specification defines event streams using the syntax described below as well as Haskell datatypes and type members. A specification can also borrow Haskell notation and features such as list comprehension and let-clauses (represented by the red dashed arrow in Fig. 1). Then, a very simple translator generates Haskell code from the source specification. This translator does not parse and interpret the totality of the source code, but only performs simple rewritings introduced to make the specification cleaner. The resulting Haskell code is then combined with the execution engine described below, written in Haskell, and compiled using the GHC to obtain the binary for the specification monitor. In this manner, the HStriver tool can be easily extended with new data-theories, and Haskell programs can use HStriver specifications as part of their code.

The Language. A stream declaration in an HStriver specification can be either:

- An input declaration, which is bound to a name and a type using the following syntax:
  input <TypeConstraints>? <Type> <name> <ArgType argame>*, or
- An output declaration, which is bound to a name, a *TickExpression te* assigned to the field **ticks** and a *ValueExpression ve* assigned to the field **val**, using the following syntax:

```
output <TypeConstraints>? <Type> <name> <ArgType argame>* :
  ticks = te
  val = ve
```

where **<TypeConstraints>** is an optional set of constraints over the polymorphic types handled by the stream and expressed in Haskell notation, and **<ArgType argame>*** is an optional list of arguments. We can use **define** instead of **output** to define intermediate streams, whose values are not reported by the monitor but can be used by other streams. We can replace the last **:** with a **=** to define an output stream as the copy of another stream instead of indicating its *Tick* and *Value Expressions*.

The types of the streams have to be Haskell **Typeable** types, which is a very general class of types, enough for our the purpose of SRV data theories. The types of **input** streams have to be parseable from JSON using the Haskell **aeson** library (i.e., they have to be an instance of the **FromJSON** class), and the output streams have to be serializable to JSON using the **aeson** library (this is, they have to implement the **ToJSON** class). Also, the current HStriver frontend imposes some minor syntactic restrictions (the work reported in this tool paper focuses on an efficient implementation with rich data theories, while ongoing work includes bringing the specification language closer to Striver).

The *TickExpression* of an output stream indicates when it might produce an event, and is defined by the following recursive datatype:

- A single point in time t, which we write **{t}**,
- The instants at which stream s contains an event, written **ticksOf** s
- The instants of the events of s shifted by a constant c, written **shift** c s,
- The instants at which a stream of type **TimeDiff** ("Time Difference") s contains an event, delayed by the value in the event (unless s contains a new event meanwhile), which we write **delay** s, or
- The union of two *TickExpressions* te_1 and te_2, which we write te_1 **U** te_2

The *ValueExpression* of an output stream indicates if the stream will contain an event at a ticking point, and with which value. A *TickExpression* also "carries" the values of all events that made the stream wake up to facilitate the computation of the *ValueExpression*. The *ValueExpression* is defined as follows:

- The constructor **'** x encapsulates an element x from a data-theory. This constructor represents the *lift* stage of the *lift-deep embedding* technique [9].
- Function application is juxtaposition, has the greatest precedence and is automatically lifted for *ValueExpressions*. Parentheses are used to impose a different association between functions and values.
- The value **cv** contains the value carried by the tick expression.
- The constructor **:=>** is used to refrain from producing a value: it will return the value at the right hand of the operator provided that the expression at the left hand side holds, and will not produce a value otherwise.

Two additional constructors allow accessing timestamps and values of different streams:

- We use `timeOf` *te* to access the timestamp of a *TauExpression* (explained below) *te*, and
- We use the projection constructor *s*`[`*te*`|_]` to access the value of a stream *s* in a *TauExpression* *te*, either (1) providing a default value *v* with the same type as *s* in case the tau expression *te* falls off the trace as in *s*`[`*te*`|`*v*`]`; or (2) with no default value, delegating the obligation to check if the expression falls off the trace to the surrounding expression, as in *s*`[`*te*`|?]`; or (3) with no default value, indicating that the inner *TauExpression* does not fall off the trace, as in *s*`[`*te*`|]`.

Finally, the datatype for *TauExpression*, which allows offsets in time:

- The value `t` represents the current time.
- The constructor *s* `«` *te* allows us to refer to the last event in stream *s* strictly before the value of the *TauExpression* *te*.
- The constructor *s* `<~` *te* is like `«` but also considers the current `t`.
- The constructors `»` and `~>` are the future duals of `«` and `<~` respectively.

Sometimes, the offset expressions can allow us to express bounds on the time which enable a more efficient implementation (very useful to capture logics like STL). We rewrite the stream accesses to make them more compact and improve legibility. Thus, *s*`[`*s* `«` *te*`|_]` becomes *s*`[<`*te*`|_]`, *s*`[`*s* `<~` *te*`|_]` becomes *s*`[~`*te*`|_]`, *s*`[`*s* `»` *te*`|_]` becomes *s*`[`*te*`>|_]`, and *s*`[`*s* `~>` *te*`|_]` becomes *s*`[`*te*`~|_]`.

The language HStriver offers the possibility to work with two temporal domains: `Double` and `UTC`. The former uses the Haskell type `Double` as the time domain, while the latter uses `Data.Time` from package `time`.

We specify the time domain for a specification with the directive `time domain` followed by either `Double` or `UTC`. HStriver libraries and theories are imported with `use library/theory` `Name`, which allows the access to functions and streams from the imported file by prepending the name of the library or theory as in `Name.member`.

The main difference between a `library` and a `theory` file is that the former contains utilities for streams manipulation and definitions, while a theory is agnostic of the Striver concepts and comprises functions and constants from a specific application domain. Data theories, as described in Sect. 2, are implemented directly in the host language, which lets us use native types and functions, as well as third parties out of the shelf, and even define our own custom types and functions as data theories. In this manner, the syntactic name of a Haskell function definition (or its lambda expression in the case of anonymous functions) make up the functional symbols used to build terms, while their semantics in the Haskell language are the functions interpretations. This characteristic of the language shows the extensibility of the language in terms of data theories.

We can also import arbitrary Haskell libraries with the directive `use haskell` `Name`. Finally, we can access functions and constants in the Haskell Prelude by prepending `P` to their names.

Example 1. In this example we show the definition of an *output* stream *stock* to calculate the stock of a certain product based on two input event streams:

`sale` that represents the sales of such product, and `arrival` which represents the arrivals of the same product. The output stream `stock` is defined to tick when either `sale` or `arrival` (or both) tick. The value carried by the tick expression is of type `(Maybe Int, Maybe Int)` and represents the units of the product sold and received at a given point in time. Notice that at least one of the members will be a `Just` value.

```
time domain Double
use haskell Data.Maybe

input Int sale
input Int arrival

output Int stock:
  ticks = ticksOf sale U ticksOf arrival
  val = let
    (msal, marr) = cv
    sal = 1'(fromMaybe 0) msal
    arr = 1'(fromMaybe 0) marr
    in
    stock[<t|0] - sal + arr
```

In HStriver we can also define a stream as the transformation of another stream, which does not require the explicit definition of a *TickExpression* and a *ValueExpression*. We call this feature *Stream aliasing*. HStriver also allows the static parameterization of streams, which lets us reuse stream definitions and instantiate these for different parameters in static time. These two features are shown in the following example.

Example 2. The following specification generalizes Ex. 1 for multiple products. It uses the `delay` operator to set up a timer and raise an alarm in case the stock of any product has been low for too long.

```
time domain Double
use library Utils
use haskell Data.Maybe

#HASKELL
data Product = ProductA | ProductB | ProductC deriving (Show, Eq)
#ENDOFHASKELL

input Int sale <Product p>
input Int arrival <Product p>

define Int stock <Product p>:
  ticks = ticksOf (sale p) U ticksOf (arrival p)
  val = let
    (msal, marr) = cv
    sal = 1'(fromMaybe 0) msal
    arr = 1'(fromMaybe 0) marr
    in
    stock p [<t|0] - sal + arr

define Bool low_stock <Product p> = Utils.strMap "low" (lowval p) (stock p)
```

```
define Bool cp_low_stock <Product p> = Utils.changePointsOf (low_stock p)

define TimeDiff alarm_timer <Product p>:
  ticks = ticksOf (cp_low_stock p)
  val = if cv then tolerance p else (-1)

define () alarm <Product p>:
  ticks = delay (alarm_timer p)
  val = '()

output () any_alarm:
  ticks = ticksOf (alarm ProductA) U ticksOf (alarm ProductB)
          U ticksOf (alarm ProductC)
  val = '()

- Alternative alarm:
define TimeDiff alarm_timer2 <Product p>:
  ticks = ticksOf (cp_low_stock p) U ticksOf (arrival p)
  val = let
    (mls, marr) = cv
    ls = 1'fromJust mls
    in if (1'isJust marr) || not ls then (-1) else tolerance p
```

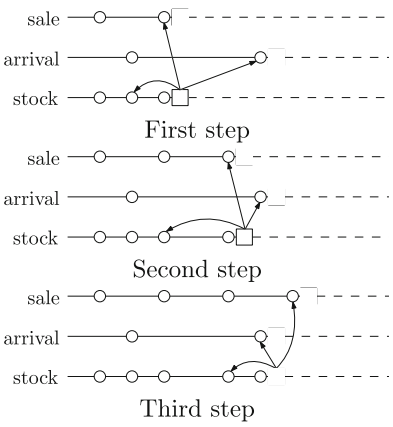
The Engine. In earlier work [18] we proposed an online monitoring algorithm, limited to past offsets only, that processes input events in strictly increasing time, producing outputs also in increasing time. We call this a *push* approach, because input events are pushed into the monitor. Instead, the implementation of HStriver follows a novel *pull* approach: the engine computes events for the output streams, which requires pulling events from other streams, and eventually pulling events from inputs. The performance of both execution approaches is similar for the common fragment of the language (i.e., the past-only fragment of Striver). Using a pull procedure, we gain expressivity in exchange for a somewhat more complex execution design. In this section we explain the pull algorithm in detail.

Input events are read from named pipes in JSON format. The main algorithm maintains the following *state* that updated at each step in the computation: (1) one *Leader* for each stream declared, and (2) one *Pointer* from one stream to another for every `timeOf` or projection used.

The task of the *Leader* is to fetch the next event in a stream when required. The *Leader* for an input stream will pull the next input event, while the *Leader* for an output stream will use its definition to calculate the next event, pulling from the pointers in the value and tick expressions if necessary. Leaders can also discover the lack of events, which is useful data for referring streams, and is necessary to prevent the system from hanging trying to calculate a real event.

A *Pointer* represents a relevant position in the sequence of events of a stream. Pointers advance from past to future over the events of a stream. The events in the past of a pointer have already been used, while the events in the future will be used later in the computation. When a Pointer needs an event that has not yet been computed, it will use the corresponding Leader to fetch it. When all

the pointers of a stream pass beyond an event, this event can be forgotten. For example, the Leader for the stream `any_alarm` in Example 2 maintains one pointer to each of the three `alarm` streams to determine when to generate the unit value. In particular, `any_alarm` will pull from every `alarm` stream at the beginning and then keep pulling from the pointer at the minimum position. Each of the `alarm` streams also maintains a pointer to its corresponding `alarm_timer`, to calculate if the corresponding `stock` is low for too long, so it produces a unit value. In particular, the leader will pull from `alarm_timer` one event to check the next timestamp and value; and one more to determine whether the timer is reset or not. The engine maintains an extra pointer for every output stream, which it uses to pull events and print them. The diagram on the right shows how the pointers are updated every time the output stream `stock` is pulled. The big box of each stream represents its Leader. Everything at the right of the leader has not yet been discovered, hence the dashed line. The leader of the stream `stock` maintains one pointer to the last event of each other stream, plus an extra pointer to its own last event (not considering the event that is being computed).



2.1 How to Run HStriver

To compile an HStriver specification or library we execute the `hstriverc` program —which is shipped with the tool— with a set of filenames, of which at most one can be a runnable specification, while the rest have to be library definitions. This will produce an executable monitor, with the name specified using the flag `-o` *filename*, or `a.out` if no output filename was specified.

To run our `monitor` over input data, it has to be executed with a parameter indicating the directory where the input data is located as its parameter: `monitor` *dir*. For every non-parameterized input stream s, the `monitor` will read its events from the file *dir*`/`s`.json`. For a parameterized input stream s with parameters $arg_0 \ldots arg_n$, the `monitor` will read the events for the instantiated input streams from the files *dir*`/`arg_0`/.../`arg_n`.json`. The input events have to be of the form `{"Time":` ts`, "Value":` val`}`, where val is the value of the stream at the instant ts, and there has to be one event per line, with a monotonically increasing timestamp. The monitor will then produce a list of events with the form `{"Id":` id`, "Time":` ts`, "Value":` val`}`, where val is the value of the stream id at the instant ts.

Note that the input files can be named pipes, which will be consumed when it is necessary to compute the next output event, following the pull model explained in Sect. 2, effectively allowing the monitor to be run over data generated in real

time. Also notice that it is easy to write a wrapper that acts as a sink for different input events and dispatches every event to its corresponding named pipe, if necessary.

Take for example the specification in Ex. 1, whose definition is in the file `stock.hstriver`, and suppose we want to execute with the input streams in the directory `ins` in the working directory. Then, we need to run:

```
$ hstriverc -o monitor stock.hstriver
$ ./monitor ins
```

There need to be two input stream files in the directory `ins`: `ins/sale.json` and `ins/arrival.json`. The monitor will print the events of `stock` to the standard output progressively when the information is available in the input stream files.

To run the specification in Ex. 2, where `paramstock.hstriver` contains the definitions and the input streams are in the sub-directory `paramins` of the current directory, we need to run

```
$ hstriverc -o monitor paramstock.hstriver
$ ./monitor paramins
```

and there need to be two directories in the directory `ins`: `ins/sale` and `ins/arrival`, with three input files inside each of them, namely

`ins/sale/ProductA.json`, `ins/arrival/ProductA.json`
`ins/sale/ProductB.json`, `ins/arrival/ProductB.json`
`ins/sale/ProductC.json`, and `ins/arrival/ProductC.json`

The monitor will print an event whenever there is a shortage of any product.[1]

3 Example Specifications and Libraries

In this section we show a selection of HStriver specifications, each of which illustrates a particular interesting feature of the language.

3.1 Example: Clock

The specification below demonstrates the use of the `delay` operator to define a specification with no input streams and one output streams `clock`, which generates a unit value at each instant multiple of 5. In this specific case, we could have used the `shift` operator instead with identical results. This example illustrates that HStriver is not only event-driven, and can generate ticks at instants where no input streams have an event.

```
time domain Double
output TimeDiff clock:
  ticks = {0} U delay clock
  val = 5
```

[1] See the tool webpage https://software.imdea.org/hstriver to find example specifications along with input and output data.

TeSSLa [10] can also implement this feature but most other systems, like RTLola [16] can only tick at periodic instants or at points at which inputs have events.

3.2 Libraries

We can use HStriver to collect reusable code and stream transformers in libraries (that do not have output streams). Libraries are declared with the directive **library** Name. Specifications can then import the definitions in the library to aid the stream definitions. Some libraries are time domain agnostic and do not require the definition of a time domain. We leverage the modules system of the host language to implement this feature. The libraries definitions contain many definitions of stream declarations from stream declarations, which shows the high-order nature of HStriver. Here we show the implementation of the library Utils, which contains useful stream operators that are used extensively in the rest of the examples.

```
library Utils

define (...) => b strMap <String funame> <(a->b) f> <Stream a s>:
  ticks = ticksOf s
  val = 1'f cv

define (Streamable a) => a filter <String funame> <(a->Bool) f> <Stream a x>:
  ticks = ticksOf x
  val = (1'f cv) :=> cv

define (Eq a, Streamable a) => a changePointsOf <Stream a s>:
  ticks = ticksOf s
  val = let
    prevMVal = s[<t|?]
    noprev = prevMVal === '-out
    prevVal = 1'getEvent prevMVal
    update = prevVal /== cv
    in noprev || update :=> cv
    where
    getEvent (Ev x) = x

define Streamable a => a firstEvOf <Stream a s>:
  ticks = ticksOf s
  val = let
    _this = firstEvOf s
    isfirst = timeOf (_this « t) === '-infty
    in isfirst :=> cv

define Streamable a => a shift <TimeDiff n> <Stream a x>:
  ticks = shift n x
  val = cv
```

We also show part of the implementations of the library STL which implements the operators of the Signal Temporal Logic STL [30], a temporal logic widely used to describe system properties of continuous signals, which are represented as timestamped event streams.

```
library STL
use library Utils
use haskell Data.Maybe

define Bool until <(TimeDiff, TimeDiff) (a,b)> <Stream Bool x> <Stream Bool y>:
  ticks = shift (-a) y U shift (-b) y U shift (-b) x U ticksOf x
  val = let
    tnow = 1'T now
    yT = filterId y
    min_yT = if Utils.shift (-a) y [~t|False] then tnow
             else timeOf (Utils.shift (-a) yT »_(b-a) t)
    xF = filterNot x
    min_xF = if not (x [~t|False]) then tnow else timeOf (xF »_b t)
    plus x tim = 2'timeDiffPlus x 'tim
    in
    min_yT `plus` a <= tnow `plus` b && min_yT `plus` a <= min_xF

define TimeDiff alwaysaux <TimeDiff n> <Stream Bool x>:
  ticks = ticksOf x
  val = let
    nextx = 1'unT (timeOf (x»t))
    frontier = 2'tDiffAdd nextx 'n
    in
    not cv :=> if not (x [t>|False]) then (-1) else 2'tDiff frontier now

define Bool statealways <TimeDiff n> <Stream Bool x>:
  ticks = let aaux = alwaysaux n x in delay aaux U ticksOf aaux
  val = 1'isNothing (snd cv)

define Bool always <(TimeDiff, TimeDiff) (a,b)> <Stream Bool x>:
  ticks = shift (-a) (statealways (b-a) x) U {0}
  val = let
    _this = always (a,b) x
    in 2'fromMaybe (_this [<t|True]) (fst cv)

define Bool eventually <(TimeDiff, TimeDiff) (a,b)> <Stream Bool x> =
  neg (always (a,b) (neg x))
```

This snippet illustrates the definitions of the $\mathcal{U}_{[a,b]}$ as `until`, $\Phi_{[a,b]}$ as `always` and $\Psi_{[a,b]}$ as `eventually`. These definitions are parametrized by the interval bounds and the streams of the sub-expressions.

3.3 STL

The next example illustrates a simple STL specification: if the input `speed` becomes `toofast`, then `speed` will decelerate continuously until reaching an admissible speed (`speedok`) within 0.5 time units (represented by the stream `slow_down`). This example shows a straightforward use of the STL library to define temporal properties as streams.

```
time domain Double

use library STL
use library Utils

const max_speed = 5
const ok_speed = 4

input Double speed

define Bool toofast = Utils.strMap "toofast" (P.>max_speed) speed

define Bool speedok = Utils.strMap "speedok" (P.<=ok_speed) speed

define Bool decel:
  ticks = ticksOf speed
  val = cv > speed[t>|0]

define Bool slow_down = STL.until (0,5) decel speedok

output Bool ok:
  ticks = ticksOf toofast U ticksOf slow_down
  val = toofast [~t|False] `implies` slow_down [~t|True]
```

3.4 Example: Cost Computation

The following example calculates the accumulated energy cost incurred by a monitor, based on a cost model for (a) waking up, (b) processing an event, (c) going to sleep, (d) being idle, (e) being awake, and also a patience parameter, which models how long to wait for a new event before going to sleep. This specification contains the definition of an output stream which is the quantitative result of a progressive computation, as opposed to typical Boolean output streams. In this example the event production is unpredictable and not governed by a predefined ratio[2]. This example uses custom datatype definitions, and event generation at instants where there is no input event.

```
time domain Double
use haskell Data.Maybe

input () wakeup

define TimeDiff sleep_delayer:
  ticks = ticksOf wakeup
  val = 'patience

define () sleep:
  ticks = delay sleep_delayer
  val = '()

define RunMode runMode:
  ticks = ticksOf wakeup U ticksOf sleep
  val = if 1'isJust (fst cv) then 'Alert else 'Sleeping
```

[2] The full code or all examples and libraries in this section can be found in https://software.imdea.org/hstriver.

```
output Cost cost:
  ticks = ticksOf runMode
  val = let
    previousRunMode = runMode[<t|Alert]
    currentRunMode = cv
    costOfTransitioning = 2'transitionCost previousRunMode currentRunMode
    getTimeT (T x) _ = x
    getTimeT _ y = y
    prevt = 2'getTimeT (timeOf (runMode « t)) now
    timediff = 1'(round.realToFrac) (2'tDiff now prevt)
    accum = cost [<t|0] + timediff * ('runCostPerSecond previousRunMode)
    in
    accum + costOfTransitioning
```

3.5 Example: PowerTrain

Our third example makes a heavy use of the STL library to implement STL properties for the verification of a powertrain control verification from [27], where input signals change asynchronously.

```
time domain Double
use library STL

input Double verification
input Double mode
input Double pedal

- phi = []_(taus ,simTime )(((low/\<>_(0, h) high)
-   \/ (high/\<>_(0,h) low)) -> []_[eta, zeta_min] (utr /\utl))
const ut2 = 0.02
output Bool opt2 =
   STL.always (taus, simTime) (((low `conj` STL.eventually (0,h) high)
     `disj` (high `conj` STL.eventually (0,h) low))
     `strImplies` STL.always (eta, zeta_min) (utl ut2 `conj` utr ut2))

- phi = <>_[simTime,simTime] utr
const ut3 = 0.05
output Bool opt3 = STL.eventually (simTime, simTime) (utr ut3)

- phi = []_(taus,simTime) utr
const ut4 = 0.1
output Bool opt4 = STL.always (taus, simTime) (utr ut4)
```

As in [27] we use input data computed from a MatLab simulation of the powertrain. This example shows how to import and use the STL operators to describe properties. We have aimed to keep the syntax of the original properties.

3.6 Example: Smart Home

This example is a smart home specification that uses the Orange4Home dataset [11]. The following monitor calculates how much time residents spend watching TV per day, assessing that every day the people living the house should

not watch more than three hours of TV (`exceeded3hPerDay`). More interesting is `exceededAvgPlus30m`, which states that residents should not watch thirty minutes more than the total average of TV watched historically. This threshold is dynamic, and requires declaring intermediate quantitative streams that compute the average and current day TV time.

```
time domain UTC

use library Utils
use haskell Data.Time

input TVStatus livingroom_tv_status
input TVStatus office_tv_status

define Bool any_tv_on:
  ticks = ticksOf office_tv_on U ticksOf livingroom_tv_on
  val = office_tv_on [~t|False] || livingroom_tv_on [~t|False]

output Bool exceeded3hPerDay:
  ticks = ticksOf any_tv_on
  val = howMuchTvToday[~t|] > 3*60*instantsPerMinute

define Int totalTVTime:
  ticks = ticksOf any_tv_on
  val = totalTVTime [<t|0] + if any_tv_on[~t|] then 1 else 0

define Int avgTvPast:
  ticks = ticksOf any_tv_on
  val = if isNewDay[~t|] then 2'div (totalTVTime[~t|]) (countDays[~t|])
    else avgTvPast [<t|0]

output Bool exceededAvgPlus30m:
  ticks = ticksOf any_tv_on
  val = howMuchTvToday[~t|] > avgTvPast[~t|] + 30 * instantsPerMinute
```

4 Conclusion

We have a presented HStriver, a stream runtime verification tool for real-time event-streams, implemented as an eDSL with Haskell as the host language, based on a technique called lift-deep embedding. The architecture of HStriver lets us use Haskell tools straightforwardly to aid improving the confidence on the correctness of the implementation with respect to the semantics of Striver, for example using unit tests, end-to-end tests and tools like Quickcheck and LiquidHaskell, as displayed in [9]. In this seminal paper we have focused on functionality, but future work includes the certification of HStriver, which is a more accessible endeavor for a concise language with a few constructs with clean semantics as Striver than for a general purpose language. HStriver has been used in (non-critical) UAV missions, where garbage collection is forbidden for critical applications, but we are exploring the generation of Misra-C from (a restricted set of) HStriver specifications. Also, future work includes a frontend that allows adapting the input syntax to particular use cases, offering a friendly syntax and the necessary types and features from HStriver.

References

1. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Proceedings of the 5th Int'l Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04). LNCS, vol. 2937, pp. 44–57. Springer (2004). https://doi.org/10.1007/978-3-540-24622-0_5
2. Barringer, H., Havelund, K.: Tracecontract: A scala DSL for trace analysis. In: Proceedings of the 17th Int'l Symposium on Formal Methods (FM'11). LNCS, vol. 6664, pp. 57–72. Springer (2011). https://doi.org/10.1007/978-3-642-21437-0_7
3. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from eagle to ruleR. In: Proceedings of the 7th Int'l Workshop on Runtime Verification (RV'07). LNCS, vol. 4839, pp. 111–125. Springer (2007). https://doi.org/10.1007/978-3-540-77395-5_10
4. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification. LNCS, vol. 10457. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5
5. Basin, D., Klaedtke, M.H.F., Zalinescu, E.: MONPOLY: monitoring usage-control policies. In: Proceedings of the 2nd Int'l Conference on Runtime Verification (RV'11). LNCS, vol. 7186, pp. 360–364. Springer (2011). https://doi.org/10.1007/978-3-642-29860-8_27
6. Basin, D.A., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: Proceedings of the Int'l Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES), pp. 19–28. Kalpa Publications in Computing, EasyChair (2017). https://doi.org/10.29007/89hs
7. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14 (2011). https://doi.org/10.1145/2000799.2000800
8. Berry, G.: Proof, language, and interaction: essays in honour of Robin Milner, chap. The foundations of Esterel, pp. 425–454. MIT Press (2000). https://doi.org/10.7551/mitpress/5641.001.0001
9. Ceresa, M., Gorostiaga, F., Sánchez, C.: Declarative stream runtime verification (hLola). In: Proceedings of the 18th Asian Symposium on Programming Languages and Systems (APLAS'20). LNCS, vol. 12470, pp. 25–43. Springer (2020). https://doi.org/10.1007/978-3-030-64437-6_2
10. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: temporal stream-based specification language. In: Proceedings of SBMF'18. LNCS, vol. 11254. Springer (2018). https://doi.org/10.1007/978-3-030-03044-5_10
11. Cumin, J., Lefebvre, G., Ramparany, F., Crowley, J.: A dataset of routine daily activities in an instrumented home. In: Ubiquitous Computing and Ambient Intelligence, pp. 413–425. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-67585-5_43
12. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: Proceedings of the 12th Int'l Symposium of Temporal Representation and Reasoning (TIME'05), pp. 166–174. IEEE CS Press (2005). https://doi.org/10.1109/TIME.2005.26
13. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: Proceedings of the 15th Int'l Conference on Computer Aided Verification (CAV'03). LNCS, vol. 2725, pp. 27–39. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_3

14. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: Proceedings of the 18th Int'l Conference on Runtime Verification (RV'18). LNCS, vol. 11237, pp. 241–262. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_14
15. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Hazem, T.: StreamLAB: stream-based monitoring of cyber-physical systems. In: Proceedings of the 31st Int'l Conference on Computer-Aided Verification (CAV'19). LNCS, vol. 11561, pp. 421–431. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_24
16. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring. CoRR abs/1711.03829 (2017). arxiv.org/abs/1711.03829
17. Gorostiaga, F., Danielsson, L.M., Sánchez, C.: Unifying the time-event spectrum for stream runtime verification. In: Proceedings of 20th Int'l Conference on Runtime Verification (RV'20). LNCS, vol. 12399, pp. 462–481. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_26
18. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: Proceedings of the 18th Int'l Conference on Runtime Verification (RV'18). LNCS, vol. 11237, pp. 282–298. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_16
19. Gorostiaga, F., Sánchez, C.: HLola: a very functional tool for extensible stream runtime verification. In: Proceedings of the 27th Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'21). Part II, pp. 349–356. LNCS, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_18
20. Gorostiaga, F., Sánchez, C.: Stream runtime verification of real-time event streams with the Striver language. Int. J. Softw. Tools Technol. Transfer **23**(2), 157–183 (2021). https://doi.org/10.1007/s10009-021-00605-3
21. Halbwachs, N., Caspi, P., Pilaud, D., Plaice, J.: Lustre: a declarative language for programming synchronous systems. In: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, pp. 178–188. ACM Press (1987). https://doi.org/10.1145/41625.41641
22. Hallé, S.: When RV meets CEP. In: Proceedings of RV'16. LNCS, vol. 10012, pp. 68–91. Springer (2016). https://doi.org/10.1007/978-3-319-46982-9_6
23. Hallé, S., Khoury, R.: Event stream processing with BeepBeep 3. In: Proceedings of the Int'l Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES), pp. 81–88. Kalpa Publications in Computing, EasyChair (2017). https://doi.org/10.29007/4cth
24. Havelund, K.: Rule-based runtime verification revisited. Int. J. Softw. Tools Technol. Transfer **17**(2), 143–170 (2014). https://doi.org/10.1007/s10009-014-0309-2
25. Havelund, K., Goldberg, A.: Verify your runs. In: Proceedings of VSTTE'05, pp. 374–383. LNCS 4171, Springer (2005). https://doi.org/10.1007/978-3-540-69149-5_40
26. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Proceedings of the 8th Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02). LNCS, vol. 2280, pp. 342–356. Springer-Verlag (2002). https://doi.org/10.1007/3-540-46002-0_24
27. Jin, X., Deshmukh, J.V., Kapinski, J., Ueda, K., Butts, K.: Powertrain control verification benchmark. In: Proceedings of the 17th Int'l Conference on Hybrid systems: Computation and Control (HSCC'14), pp. 253–262. ACM (2014). https://doi.org/10.1145/2562059.2562140

28. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams. In: Proceedings of the 33rd Symposium on Applied Computing (SAC'18). ACM (2018). https://doi.org/10.1145/3167132.3167338
29. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebr. Progr. **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004
30. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Proceedings of FORMATS/FTRTFT 2004. LNCS, vol. 3253, pp. 152–166. Springer (2004). https://doi.org/10.1007/978-3-540-30206-3_12
31. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Springer-Verlag (1995). https://doi.org/10.1007/978-1-4612-4222-2
32. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: Proceedings of the 1st Int'l Conference on Runtime Verification (RV'10). LNCS, vol. 6418, pp. 345–359. Springer (2010). https://doi.org/10.1007/978-3-642-16612-9_26
33. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Proceedings 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14). LNCS, vol. 8413, pp. 357–372. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_24
34. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. **12**(2), 151–197 (2005). https://doi.org/10.1007/s10515-005-6205-y
35. Sánchez, C.: Online and offline stream runtime verification of synchronous systems. In: Proceedings of the 18th Int'l Conference on Runtime Verification (RV'18). LNCS, vol. 11237, pp. 138–163. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_9
36. Sen, K., Roşu, G.: Generating optimal monitors for extended regular expressions. In: Electronic Notes in Theoretical Computer Science, vol. 89. Elsevier (2003). https://doi.org/10.1016/S1571-0661(04)81051-X
37. Stolz, V., Huch, F.: Runtime verification of concurrent Haskell programs. Electron. Notes Theor. Comput. Sci. **113**, 201–216 (2005). https://doi.org/10.1016/j.entcs.2004.01.026