Formal Methods in System Design
https://doi.org/10.1007/s10703-023-00428-9

ORIGINAL ARTICLE

Runtime verification of real-time event streams using the tool HStriver

Felipe Gorostiaga[1,2,3] · **César Sánchez**[1]

Received: 29 April 2022 / Accepted: 16 April 2023

Abstract
We present in this paper the tool HStriver, an extensible stream runtime verification tool for monitoring real-time event streams. Real-time event streams are formed by events that contain rich data and can occur at arbitrary times. The rich expressivity of HStriver allows the specifications of quantitative semantics of logics like signal temporal logic (STL), including different notions of robustness. Stream runtime verification is a specification family of languages based on the clean separation between temporal dependencies and data computations. To encode the data values contained in the events (both read as inputs and produced as the result of computation) HStriver borrows a large subset of data-types from Haskell. These types are transparently lifted into the HStriver specification language and incorporated in the temporal engine, so they can be used as the types of the input (observations), output (verdicts), and intermediate streams. The temporal evaluation engine is agnostic of the types used in the specification. The resulting extensible language is then embedded into Haskell as an embedded Domain Specific Langauge. 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 definitions), etc. The resulting tool, HStriver, is a flexible and extensible stream runtime verification engine for real-time streams. We illustrate the use of HStriver on many sophisticated real-time specifications, including realistic STL properties of existing designs.

Keywords Runtime verification · Stream runtime verification · Real-time streams

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)", and by FPU Grant FPU18/04362 granted by the Spanish Ministry of Science, Innovation and Universities.

✉ Felipe Gorostiaga
felipe.gorostiaga@imdea.org

César Sánchez
cesar.sanchez@imdea.org

[1] IMDEA Software Institute, Madrid, Spain

[2] Universidad Politécnica de Madrid, Madrid, Spain

[3] CIFASIS, Rosario, Argentina

Published online: 21 June 2023

1 Introduction

Runtime Verification (RV) [1–3] is a lightweight dynamic formal technique for systems reliability. The main concerns of RV are how to generate monitors from formal specifications, and algorithms that use the generated monitors to process, one at at time, traces of the system under analysis. The first RV specification languages, proposed almost twenty years ago, were based on temporal logics like past LTL [4] adapted to finite traces [5–7], regular expressions [8], rewriting [9], fix-point logics [10], rule based languages [11]. In these languages, verdicts (and many times observations) are Boolean, because these logics borrowed from static verification–-where decidability is crucial.

Stream runtime verification (SRV) [12, 13] attempts to generalize these monitoring algorithms to richer datatypes, including observations and verdicts. This richer setting allows the computation of quantitative values and summaries, the computation of witnesses, models for the collection of representative data, etc. The keystone of SRV is to cleanly separate the temporal engine (the *when*) that specifies the moments at which values are collected and processed during the computation, from the individual data operations (the *what*). Therefore, temporal monitoring algorithms are designed abstractly and then instantiated to concrete types and data operations. SRV languages offer declarative specifications where offset expressions allow accessing streams at different moments in time, including future instants. The first SRV developments [12] were synchronous, similar to the so-called *synchronous languages* like Esterel [14] or Lustre [15]. 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 [16–20]. 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 real-time SRV language[1]. The core language is based on Striver [18], and enables the extension to arbitrary datatypes, implemented as an embedded DSL in Haskell. There are other RV tools implemented as eDSLs [22–24] 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 [25].

Most of the HStriver datatypes were introduced after the temporal engine was completely implemented, requiring no re-implementation of the engine. Similarly, users of HStriver can also extend the collection of datatypes easily. The second contribution of HStriver is the implementation of a novel efficient asynchronous engine for the temporal part, described in Sect. 2.5. 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. In turn, this enables features like stream parametrization, which requires costly ad-hoc implementations in previous tools. This idea is used to pack HStriver libraries that describe complex logics like STL, both with Boolean and quantitative semantics, in just a few lines.

[1] An earlier short version of this paper appeared in [21].

1.1 Related work

There have been runtime verification tools for the monitoring of event-based streams (see [26] for a survey). R2U2 [27] 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 [28, 29] 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 [30, 31] 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 [19, 33] and TeSSLa [16] 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.

1.2 Contributions and structure

The main contributions of this paper are:

- a description of the implementation of HStriver, an SRV tool for real-time event streams using a lift deep embedding in Haskell;
- a novel pull algorithm to implement an asynchronous temporal engine and;
- examples that illustrate many of the HStriver features.

The additional contributions of this journal version with respect to the short version [21] are:

- a more detailed description of the implementation of the core of HStriver;
- a deep discussion of the lift-deep embedding;
- a new section about static analysis;
- nested specifications, which in turn leads to a new example (RobustSTL) and
- an empirical evaluation of the engine.

The rest of the paper is structured as follows. Section 2 introduces SRV and describes the internals of HStriver. Section 3 illustrates many features by example. Section 4 contains an empirical evaluation using HStriver. Finally, Sect. 5 concludes.

2 The HStriver Tool

Stream Runtime Verification generalizes monitoring algorithms (which are typically defined for Boolean observations and verdicts) to arbitrary data. Data is abstracted using multi-sorted

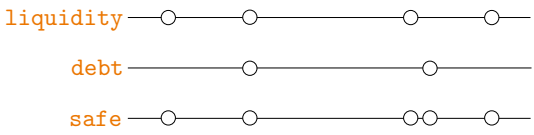


Fig. 1 Timeline of `safe` events

first-order interpreted signatures. The resulting 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. The main idea of a specification is to provide, declaratively, the relation between outputs (verdicts) and input (observations). The temporal dependencies in these declarations are used to determine how to traverse input streams, fetch individual data and ultimately produce each individual output event. These events consist of a *value* from a data theory, and a *timestamp* from a specification-fixed temporal domain. The concrete operations used in the specification are used for the details on how to compute the output data. 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.

Consider the following example in which we define a stream `safe` that computes if a company has acquired debt above 30% of its liquidity.

```
1 input Int liquidity
2 input Int debt
3 output Bool safe:
4   ticks = ticksOf liquidity U ticksOf debt
5   val = liquidity[~t|]*0.3 > debt[~t|]
```

This stream is updated whenever the input streams `liquidity` or `debt` produce new values, and computes if the current value of `debt` is greater than 0.3 times the current value of `liquidity`. Figure 1 on the right shows graphically the asynchrony of the event trace, and how `safe` produces values when the other streams do.

The temporal core of the tool HStriver is based on the Striver specification language, whose theoretical foundations are described in [18]. A specification $\langle I, O, E, T \rangle$ consists of

- a set of typed input stream variables I, which correspond to the input observations;
- a set of typed output stream variables O which represent outputs of the monitor and intermediate observations; and
- a collection of defining equations, which associate every output $y \in O$ 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 (that model the existence of an event at a specific time point), and operators for the union, shift and delays of the ticking instants of streams. Stream Expressions E_y are built from constants 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 language is explained in depth in Sect. 2.2. The simple 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 increasing time order. The algorithm implemented in the HStriver tool, and explained in Sect. 2.5, is a much more efficient *pull* algorithm, which attempts to compute events in output streams fetching the necessary events from other streams.

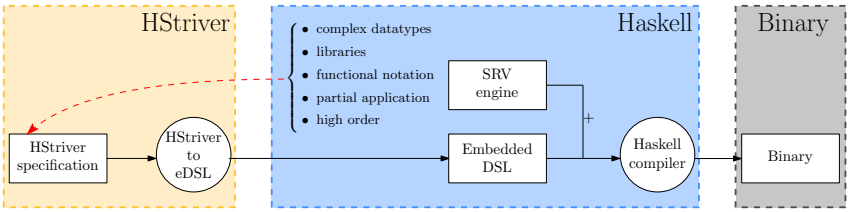


Fig. 2 Software architecture of HStriver

2.1 HStriver architecture

The architecture of HStriver is shown in Fig. 2.

An HStriver specification defines event streams following the syntax explained in Sect. 2.2 below, where one distinguishing feature is that it borrows Haskell datatypes and type members for the syntax of HStriver data expressions. A specification can also borrow Haskell notation and features such as list comprehension and let-clauses, represented by the red dashed arrow in Fig. 2. Then, a very simple translator generates Haskell code with the implementation explained in Sect. 2.3 from the source specification. This translator does not parse or interpret the source code, but only performs simple rewritings introduced to make the specification cleaner for HStriver programmers. The resulting Haskell code is then combined with the execution engine described in Sect. 2.5, written fully 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 directly use HStriver specifications as part of their code.

2.2 The tool HStriver

In this section we present informally the semantics of the language HStriver, which proceeds inductively over the trace. A more thorough formal description of the semantics can be found in [20]. We have chosen this inductive approach opposed to, for example, the search for a fixpoint using small-step semantics as in [34, 35] because we believe that an inductive semantics makes it easier to prove the correctness of the operational semantics. 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 Type name <args>*`
- An output declaration, which is bound to a name, a *tick expression te* assigned to the field `ticks` and a *value expression ve* assigned to the field `val`, using the following syntax:

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

where `TypeConstraints` is an optional set of constraints over the polymorphic types handled by the stream expressed in Haskell notation, and `<args>*` is an optional list of arguments of the form `Type name`. We can define `x` as an *alias* for the stream `y` with `output TypeConstraints? Type x = y`. We can also use `define` instead of `output` to define an intermediate stream, whose values are not reported if defined in a specification, and which are not accessible if defined in a library, just like in HLola.

The types of the streams have to be Haskell `Typeable` types, which is a very general class of types, enough for 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 paper focuses on an efficient implementation with rich data theories, while future work includes bringing the specification language closer to Striver).

The *tick expression* of an output stream **`x`** indicates when **`x`** might contain an event, and is defined by the following recursive datatype:

- A single point in time `t`, which we write `{t}`, where `t` is a value of type **`Time`**. This type represents the temporal domain of the specification. For example, the expression `{5}` indicates that **`x`** may contain an event with timestamp 5.
- The instants at which a stream `s` contains an event, written **`ticksOf`** `s`.
  For example, the expression **`ticksOf`** `s` indicates that **`x`** may contain events at the timestamps of the events of **`s`**.
- The instants of the events of `s` shifted by a constant `c`, written **`shift`** `c s`.
- The instants at which a stream `s` of type **`Time`** contains an event, delayed by the value in the event which we write **`delay`** `s`, **`delay+`** `s`, or **`delay-`** `s` depending on the sign of the values of `s`. Note that the **`shift`** operator works with a predefined constant, while **`delay`** can read the delay value from a stream.
- The union of two tick expressions te_1 and te_2, which we write te_1 **`U`** te_2

When writing specifications it is very convenient to be able to access the data contained in those events that make a stream tick according to its tick expression. For this reason, we add specific syntax to access the values *carried* by those events that made the stream wake up to facilitate the computation of the value expression. The *value expression* of an output stream indicates if the stream will contain an event at a ticking point, and with which value. The value expression is built with the following syntax:

- The constructor **`'`**`o` encapsulates an element *o* from a data-theory. This constructor represents the *lift* stage of the *lift-deep embedding* technique explained in Sect. 2.3. For example, we can lift the element *True* from the *Boolean* theory from Haskell using **`'`**`true`. We sometimes indicate the arity of the object *o* being lifted for clarity or to aid the type inference. For example, if `f` is a function of the theory that takes two arguments, we would write **`2'`**`f`. Constant values have an arity of 0, so we can write **`0'`**`true` if necessary. To improve readability, some operators have been overridden by default by their lifted version, such as `if · then · else ·`.
- Function application is juxtaposition and has the greatest precedence. Parentheses are used to impose a different association between functions and values.
- The value **`cv`** contains the value carried by the tick expression.
  For example, if the tick expression is **`ticksOf`** `s`, the value of **`cv`** will be the value of **`s`** at that time. The value carried by a singleton expression `{t}`, is unit.
- The constructor **`notick`** is used to refrain from producing a value. If at some point the value expression of a stream definition is **`notick`**, then the stream will not produce an event at all. This expression is typically used within `if · then · else ·` expressions and is useful to filter streams. For example, the expression `if cv > 10 then cv else notick` will generate events with the (numeric) carried value only when the carried value is greater than ten.

Tau Expression	Access Syntax sugar
s<<t	s[<t|·] = s[s<<t|·]
s<~t	s[~t|·] = s[s<~t|·]
s>>t	s[t>|·] = s[s>>t|·]
s~>t	s[t~|·] = s[s~>t|·]

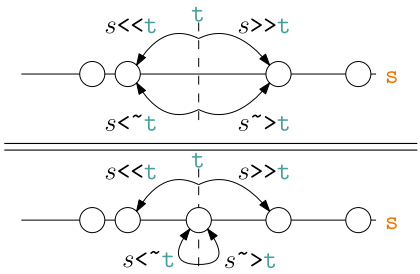


Fig. 3 Stream access operators and their behaviors when there is an event at `t` or not

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

- The expression `timeOf` *te* accesses the resulting timestamp of a *time expression* *te* (also called a tau expression), and
- The projection constructor *s*`[`*te*`|`*v*`]`, which accesses the value of a stream *s* at a tau expression *te*, and returns the value unless the time calculation results in "falling off the trace", this is, the result being lower than the minimum temporal value or greater than the maximum temporal value, in which case *v* is returned. The syntax also offers the variant *s*`[`*te*`|?]` which returns a supertype of *s* with additional values to indicate the falling off the trace, and the variant *s*`[`*te*`|]` which is only legal if the offset of the tau expression is guaranteed to exist (the type system guarantees this).

The value of the expression *s*`[`*te*`|?]` may be a value of the type of *s* or one of the special vualues `out` or `-out`, which represent that the access has fallen off the trace (i.e., that there is no previous/next value of *s*, depending on the operator). Similarly, the expression `timeOf` *te* can return either a value from the time domain or one of the special values `infty` or `-infty`.

Finally, the datatype for tau expressions, which allows offsets in time is:

- `t` which represents the current time.
- The constructor *s*`<<`*te*, which allows referring to the last event in stream *s* strictly before the value of the tau expression *te*.
- The constructor *s*`<~`*te*, which is like `<<` but also considers *te* as a candidate, this is, *s*`<~`*te* allows referring to the event in *s* exactly at time *te*, but it behaves like *s*`<<`*te* if such event does not exist.
- The constructors `>>` and `~>`, which are the future duals of `<<` and `<~` respectively.

We show a summary of the stream access operators in Fig. 3.

We also define versions of `>>` and `~>` that are decorated with a time limit *b* for the next event to be considered, which we write `»`$_b$ and `~>`$_b$ respectively, and are useful to efficiently capture logics like STL, as we show in Sect. 3.2. The syntax of HStriver contains some syntactic sugar for 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*`~|·]`. We also define `now` as a synonym of `timeOf t`. Also, HStriver allows the declaration of a top level constant value or function *x* (with definition *def*) using `const` *x* `=` *def* or `fun` *x* `=` *def* .

The current version of HStriver offers the possibility to work with two temporal domains: `Double` and `UTC`. The former option uses the Haskell type `Double` as the time domain, while the latter uses the Haskell library `Data.Time` of the Haskell 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. 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 are implemented directly in the host language, which let us use native types and functions, as well as third parties off-the-shelf implementations, and even define our own custom types and functions as data theories with the directive `data`. 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 illustrates the extensibility of HStriver in terms of data theories. We can also import arbitrary Haskell libraries with the directive `use haskell Name`.

When we import a library, a theory or a Haskell library `Lib`, we can include the reserved word `qualified` after `use` to avoid name clashes. This forces us to prepend the name of the library or theory before accessing a definition `member` from it, as in `Lib`.*member*. We can access functions and constants in the Haskell Prelude by prepending `P` to their names.

Example 1 The following specification defines a stream `y` that filters out the negative values of an integer input stream `x`. The stream `y` over-approximates its tick instants as the tick instants of `x`, and then delegates the filtering to its value expression.

```
1 input Int x
2 output Int y:
3   ticks = ticksOf x
4   val = if x[~t|] < 0 then notick else x[~ t|]
```

Example 2 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.

```
1  time domain Double
2  use haskell Data.Maybe
3  input Int sale
4  input Int arrival
5  output Int stock:
6    ticks = ticksOf sale U ticksOf arrival
7    val = let
8      (msal, marr) = cv
9      sal = 1'(fromMaybe 0) msal
10     arr = 1'(fromMaybe 0) marr
11     in
12     stock[<t|0] - sal + arr
```

We use the function `fromMaybe` from Haskell, we apply it to the Haskell integer `0`, and we lift the resulting (partially applied) function to safely get the number of sold and received products, defaulting to 0 if the corresponding input stream is not ticking.

HStriver also allows the static parameterization of streams, which lets us reuse stream definitions and instantiate these for different parameters in static time.

Example 3 The following specification generalizes Example 2 for multiple products. This example 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.

```
1  time domain Double
2  use library Utils
3  use haskell Data.Maybe
4  data Product= ProductA | ProductB | ProductC deriving (Show, Eq)
5  fun tolerance ProductA = 10
6  fun tolerance ProductB = 15
7  fun tolerance ProductC = 20
8  fun lowval x = x `leq` 0
9  input Int sale <Product p>
10 input Int arrival <Product p>
11 define Int stock <Product p>:
12   ticks = ticksOf (sale p) U ticksOf (arrival p)
13   val = let
14     (msal, marr) = cv
15     sal = 1'(fromMaybe 0) msal
16     arr = 1'(fromMaybe 0) marr
17     in
18     stock p [<t|0] - sal + arr
19 define Bool low_stock <Product p> = mapFun "low" lowval (stock p)
20 define Bool cp_low_stock <Product p> = changePointsOf (low_stock p)
21 define Time alarm_timer <Product p>:
22   ticks = ticksOf (cp_low_stock p)
23   val = if cv then tolerance p else (-1)
24 define () alarm <Product p>:
25   ticks = delay (alarm_timer p)
26   val = '()
27 output () any_alarm:
28   ticks = ticksOf (alarm ProductA) U ticksOf (alarm ProductB)
29           U ticksOf (alarm ProductC)
30   val = '()
```

2.3 Language implementation

The core of the HStriver language is implemented as an embedded Domain Specific Language (eDSL) in Haskell. In this section we explain the benefits and the drawbacks of this approach and the use of the novel technique of *lift-deep embedding*.

2.3.1 The host language Haskell

Haskell [36] is a pure statically typed functional programming language that allows custom parametric polymorphic datatypes, which eases the definition of new data theories in HStriver and enables us to abstract away the types of the streams, effectively allowing the expression of type-generic specifications.

As a design principle, and in order to facilitate type independent temporal engines, when a specification is processed, we drop the information about the types of the streams so streams of different types can be mixed and used in the same specification. One drawback of this approach is that the Haskell type-system can no longer track the original type of a stream, 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 HStriver engine keeps enough information to parse the values from input streams and to produce output values given a stream name, avoiding type mismatches when converting from and to dynamically-typed objects. As a result, a runtime type error can only be produced when processing an input event whose value received as input is not of the expected type. Otherwise, types are guaranteed to be respected during the computation.

Output streams of HStriver are defined using functions from data theories. These are functions in the mathematical sense, meaning that they do not have side effects and the tool does not make assumptions about when these functions will be called. Data theory functions are expected to yield the same result when applied to the same arguments twice, which is aligned with the Haskell purity of (total) functions.

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 user defined custom datatypes. Extensible encoding and decoding of datatypes in the theory is not trivial and such a feature might account for a large portion of the code-base in other implementations, draining implementation and maintenance effort from more fundamental aspects of the tool. Haskell allows defining custom datatypes via the `data` statement which once defined can be used just like any other type in Haskell. HStriver relies on Haskell's facilities to easily define how to encode and decode datatypes in JSON format, most of the times automatically from their definitions using Haskell's `deriving` mechanism.

Haskell allows redefining functions that are typically native in other languages, such as Boolean operators (||) and (&&), and the arithmetic operators (+), (−) and (∗), as well as define and use custom infix operators. Haskell also offers let-bindings, list comprehensions, anonymous functions, higher-order, and partial function application, all of which improves specification legibility. We use higher-order functions to describe transformations that produce stream declarations from stream declarations, obtaining static parameterization for free, which allows the programmatic definition of specifications.

Using eDSLs brings benefits beyond data theories, including leveraging Haskell's parsing, compiling, type-checking, and modularity. The HStriver engine uses Haskell's module system to allow modular specifications, to build language extensions, and to import third parties libraries. As a result, HStriver allows collecting reusable code and stream transformers in libraries, which specifications can then import to aid the stream definitions.

One drawback of using eDSLs 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 HStriver. Moreover, improvements in the Haskell compiler and runtime systems

will be enjoyed seamlessly, and new features will be ready to be used in the engines right away.

2.3.2 Lift-deep embedding

DSLs allow implementing a language by embedding a guest language (in our case HStriver) into a host language (in our case Haskell). A deep embedding is typically used for DSLs since the structure of the programs of the guest language is faithfully represented as a data-type in the host language. Typically, a DSL is designed as a complete language upfront, 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 programming language used in the implementation. However, our intention is to fulfill the promise in SRV of the clean separation between data theories and temporal engines. Therefore, we pursue a solution as a language where datatypes are not decided upfront but can be added on demand without requiring any re-implementation.

A DSL with user-defined data types has a general internal format for representing terms of these types. We define a lifting operator that takes Haskell terms into the term representation of HStriver. As a consequence, HStriver borrows (almost) arbitrary types from the host language Haskell, resulting in a tool that is agnostic from the types handled and even allows types to be defined and included after the implementation of the language. This technique allows us to incorporate Haskell datatypes into HStriver, and enables the use of many features from the host language in the SRV engine. We seek to represent many data 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 additional stream access primitives to capture the corresponding offset expressions in HStriver, as explained below.

The engine defines expressions in Haskell as parametric datatypes with a polymorphic argument domain. The generic domain is automatically instantiated in static time by the Haskell compiler, effectively performing the desired lifting of Haskell datatypes to types of the theory in the language. The resulting concrete *Expressions* constitute a typical deeply embedded DSL. This technique allows us to *lift* Haskell datatypes to HStriver and then perform a single *deep embedding* for all lifted datatypes, saving us from defining a constructor for all elements in the data theory, and making the incorporation of new types transparent. The application of 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.

2.3.3 Language internals

We define the stream declarations of HStriver in Haskell as a parametric datatype Stream with a polymorphic argument:

- The following constructor represents the definition of an **`input`** stream from its name and the names of its arguments:
  ```
  Input :: String → [String] → Stream a
  ```

- Similarly, the output constructor represents an **output** stream, and associates a stream with its name, its *tick expression* and its *value expression*:

```
Output :: String → TickExpr cv → ValExpr cv a → Stream a
```

We explain the datatypes that represent the tick and the value expressions respectively. First, *tick expressions* are modeled by a parametric datatype TickExpr whose polymorphic argument indicates the type of the carried value:

- The literal instant {·} constructor is:
  ```
  ConstTE :: Time → TickExpr ()
  ```
- The union constructor U is implemented by:
  ```
  (:+) :: TickExpr cv0 → TickExpr cv1 → TickExpr (Maybe cv0, Maybe cv1)
  ```
- The **shift** is implemented as follows:
  ```
  ShiftTE :: Time → Stream cv → TickExpr cv
  ```
  A special case of this constructor (specifying a deviation of 0) represents the **ticksOf** operator.
- Finally, the **delay** operator is implemented by the following constructor:
  ```
  DelayTE :: DelayDir → Stream Time →TickExpr ()
  ```
  The direction DelayDir indicates the sign of the delay stream, and can be either Positive or Negative. The **delay** and **delay+** operators are translated to a DelayTE expression with Positive direction, and the **delay-** operator generates a DelayTE expression with Negative direction.

Value expressions are implemented as a parametric datatype ValExpr cv a whose polymorphic arguments indicate the type of the carried value (cv) and the type of the expression itself (a):

- The lift ' expression is implemented by the following constructor:
  ```
  Leaf :: a → ValExpr cv a
  ```
  This operator constructs a value expression with the type of its argument, regardless of the type of the carried value.
- The carried value expression **cv** is implemented by the following constructor:
  ```
  CV :: ValExpr cv cv
  ```
  The CV operator constructs a value expression with the type of the carried value.
- The reserved value **notick** is implemented as:
  ```
  Notick :: ValExpr cv a
  ```
  The Notick operator constructs a value expression of any type.
- The special function `if · then · else ·` is implemented by the constructor:
  ```
  ITE :: ValExpr cv Bool → ValExpr cv a → ValExpr cv a → ValExpr cv a
  ```
- The following constructor allows obtaining the **timeOf** a *tau expression*:
  ```
  Tau :: TauExpr x → ValExpr cv ETime
  ```
  The type ETime includes the possibility that the expression returns an infinite value (**infty** or **-infty**).
- Similarly, the following constructor allows obtaining the event in a tau expression:
  ```
  Proj :: TauExpr x → ValExpr cv (MaybeOutside x)
  ```

The type MaybeOutside includes the possibility that the expression returns an outside value (`out` or `-out`).

Finally, *tau expressions* are represented as a parametric datatype TauExpr whose polymorphic argument indicates the type of the stream being accessed:

- The constructor TauT :: TauExpr a represents the reserved symbol `t`.
- The constructor (:<<) :: Stream a → TauExpr b → TauExpr a represents the tau expression `<<`.
- The constructor (:<~) :: Stream a → TauExpr b → TauExpr a represents the tau expression `<~`.
- The constructor (:>>) :: Stream a → TauExpr b → TauExpr a represents the tau expression `>>`.
- Finally, the constructor (:>~) :: Stream a → TauExpr b → TauExpr a represents the tau expression `~>`.

For example, the specification of Example 1, in which the stream `y` filters out the negative values of an input stream `x` is translated in the eDSL of HStriver as follows:

```
x :: Stream Int
x = Input "x" []

y :: Stream Int
y = let
  ticks = ShiftTE 0 x
  val   = ITE (App (Leaf (<0)) CV)
          Notick CV
  in Output "y" ticks val
```

The tick expression of y is the times at which x ticks, shifted by 0. Then, the value expression is Notick if the application of the (data theory) unary function (<0) to the current value of x returns *True*, and the current value of x otherwise. This illustrates how tick expressions are maintained simple, delegating the actual generation of the event to the value expression.

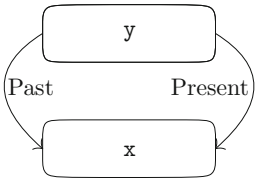
2.4 Static analysis

Even though the Haskell compiler takes care of most of the static checks that guarantee that the input specification is legal—like syntactic errors and type mismatches—lightening the burden of implementing these checks in HStriver, some remaining syntactic checks are specific to the language, and require us to run additional analyses over the specification to confirm its validity.

The work that introduced Striver [20] contains a detailed discussion on the syntactic conditions that ensure the semantic well-definedness of a Striver specification. The condition essentially corresponds to the lack of (certain kind of) cycles in the *dependency graph* of the specification, which indicates if a stream can depend on past, present or future values of another stream (including itself). Then, HStriver first computes the dependency graph of the specification by traversing the stream definitions recursively. After that, it assesses that the closed paths in every maximal strongly connected component are either all *past*, or all *future* paths, deciding the well-formedness condition of the input specification. Additionally, HStriver also checks that `notick` is used only in a branch of an `if · then · else ·` expression, possibly nested, but not as an argument of a function or as a function itself. Note that we could have used the type system of Haskell to enforce the correct usage of `notick` expressions

by requesting that the *Value Expressions* return a Maybe value, and interpreting the absence of value (Nothing) as a `notick`, but we believe that this alternative would have polluted the syntax and added an overhead that is irrelevant for most specifications.

The following is the dependency graph of the filter specification from Example 1.



We can see here that `y` depends on the present value of `x`, due to the use of `ticksOf` and `<~` and it also depends on past values of `x` due to the presence of `<~`.

2.5 The engine

The online monitoring algorithm proposed in [18] is limited to past offsets only, and it processes input events in strictly increasing time, producing outputs also in increasing time. We call this a *push* algorithm, 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., for the past-only fragment of Striver). Using a pull procedure, we gain expressivity in exchange for a slightly more complex execution design. In this section we focus on how the engine works and we explain the pull algorithm in detail. See [37, 38] for the proof of correctness of the engine algorithm, whose showcasing is the main goal of this tool paper.

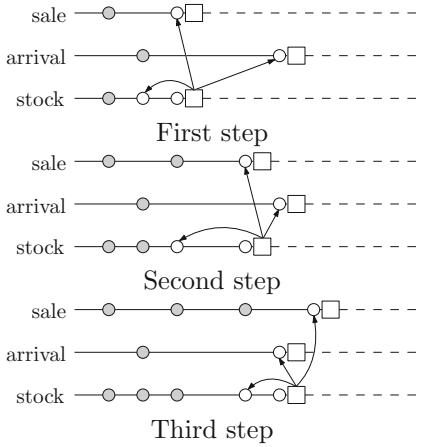
Input events are read from named pipes in JSON format. The main algorithm maintains the following *state* that is updated at each step in the computation:

- one *Leader* for each stream declared, and
- 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 traversing or generating 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 pointing to a stream pass beyond an event, this event can be forgotten, keeping the number of events in memory at the minimum. For example, the leader for the stream `any_alarm` in Example 3 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). The events shown in grey are events that can be forgotten by the engine because all the pointers that point to their corresponding streams have passed beyond those events.

2.5.1 Nested specifications

We describe now a feature of HStriver that simplifies specifications in many occasions by allowing the creation of new data theories as the result of the computation of a monitor, in order to use the results in higher monitoring activities. We call this feature nested monitoring or nested specifications.

The main function that implements the monitoring algorithm is *runSpec*, which takes an HStriver specification and an input trace for every `input` stream and returns the successive events of the `output` streams. As a by-product of developing HStriver as an eDSL in Haskell, the datatypes that constitute an HStriver specification and the function *runSpec* are defined in Haskell and as such can be lifted to be a theory of the language HStriver itself. This has enabled the use of HStriver specifications as a theory from within the language [37]. This feature does not extend the expressive power of HStriver, but it enables us to consider the language itself as a theory that can be used to define specifications, in the same way as static parameterization lets us reuse definitions without extending HStriver itself.

Nested specifications allow spawning and executing monitors dynamically, collecting the result in each invocation and using it as a value in the caller monitor. Defining an *inner* specification involves giving it a name and adding an extra clause: `return` x `when` y where x is a stream of any type and y is a `Boolean` stream. The type of the stream x determines the type of the value returned when the specification is invoked dynamically. Optionally, we can

provide parameters when defining the nested specification, which are considered constants within the monitor. Once we have defined an inner specification `spec`, we can execute it with the function *runSpec*, providing the necessary parameters and lists of values for the input streams, in the order in which they are defined in `spec`. When an inner specification with a return clause `return x when y` is executed, the computation will return the value of the stream x at the first time y becomes *true*, or the last value of x if y never holds in the execution. If x did not produce a value when y becomes *true* the value of the execution is the special value `-out`.

An inner specification is compiled by the frontend preprocessor to a special folder in the HStriver Haskell codebase so it is available for a caller monitor to import it as a module with the directive `use innerspec spec`.

Using the rich expressive power of HStriver we can define a stream $\texttt{win}_s$ that contains the events of a stream s in a window of length w as shown in the following program:

```
1 output [(Time, A)] win_s =
2   ticks = ticksOf s U shift (-w) s
3   val = let
4     (mold, mnew) = cv
5     prevList = win_s [<t| '[]]
6     nextList = if 1'isNothing mold then prevList else 1'tail prevList
7     in
8       if 1'isNothing mnew then nextList
9       else nextList ++ [(now+w, 1'fromJust mnew)]
```

The output stream $\texttt{win}_s$ updates the list of events when an event of s is leaving the sliding window of events (i.e., when s is producing a value). This definition also uses the `shift` operator to retrieve the future values of the stream s and incorporate them to the sliding window.

The following parametric auxiliary stream `slice` is offered by HStriver with the following signature `output (A, [(Time, A)]) slice <Int a> <Int b> <Stream A x> =` The stream returns the timestamped values of the stream `x` within the interval (`a`, `b`] along with the last value of `x` before `a`. We will usually use slices as input streams for inner specifications. The incorporation of nested specifications and slices as libraries in the language greatly simplifies some stream definitions when we let $x\texttt{[}a\texttt{:}b\texttt{]}$ be syntactic sugar to refer to slices.

2.6 How to run HStriver

To compile an HStriver specification or library we execute the `hstriverc` program–-which is shipped with the tool–-indicating a set of filenames, of which at most one can be a runnable specification, while the rest have to be library definitions or inner specifications. The result of successfully running `hstriverc` will be 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, the program generated has to be executed with a parameter indicating the directory where the input data is located: `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 $dir/arg_0/\ldots/arg_n$`.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 algorithm explained in Sect. 2.5, effectively allowing the monitor to be run over data generated in real time. 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.

Consider the specification in Example 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 needs 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 Example 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/sale/ProductB.json`, `ins/sale/ProductC.json`, and `ins/arrival/ProductA.json`, `ins/arrival/ProductB.json`, `ins/arrival/ProductC.json`. The monitor will print an event whenever there is a shortage of any product. The tool webpage https://software.imdea.org/hstriver contains several example specifications along with input and output data.

3 Example specifications

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

3.1 Example #1: clock

The following specification demonstrates the use of the `delay` operator to define a specification with no input streams and one output stream `clock`, which contains a unit value at each instant multiple of 5. ticks at instants where no input streams have an event.

```
1 output Time clock:
2   ticks = {0} U delay clock
3   val = 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 stream has an event. TeSSla [16] can also implement this feature but most other systems, like RTLola [33], 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 using the directive `library Name`. Specifications can then import the definitions in the library. Some libraries are time domain agnostic and do not require the definition of a time domain. We leverage the module system of the host language to implement this feature. Many libraries contain 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.

```
1  library Utils
2  output b mapFun <String funame> <(a->b) f> <Stream a s>:
3    ticks = ticksOf s
4    val = 1'f cv
5  output Eq a => a changePointsOf <Stream a s>:
6    ticks = ticksOf s
7    val = let
8      prevMVal = s[<t|?]
9      noprev = prevMVal === '-out
10     prevVal = s[<t|]
11     update = prevVal /== cv
12     in if noprev || update then cv else notick
13 output a shift <Time n> <Stream a x>:
14   ticks = shift n x
15   val = cv
```

The parametric stream `mapFun` applies a given function to every event in another stream. We use the stream `changePointsOf` to only replicate the events that represent a change of the current value of a signal. Finally, `shift` is a utility to apply an offset to the timestamps of all the events in a stream.

We also show part of the implementation of the library `STL` which implements the operators of the signal temporal logic [39], a temporal logic widely used to describe system properties of continuous signals, which are represented as timestamped event streams.

```
1  library STL
2  use qualified library Utils
3  use haskell Data.Maybe
4  output Bool mu <String funame> <Stream a x> <(a->Double) f>:
5    ticks = ticksOf x
6    val = 1'f cv > 0
7  define Bool keepTrues <Stream Bool s>:
8    ticks = ticksOf s
9    val = if cv then cv else notick
10 define Bool keepFalses <Stream Bool s>:
11   ticks = ticksOf s
12   val = if not cv then cv else notick
```

```
13 output Bool until <(Time,Time) (a,b)> <Stream Bool x> <Stream Bool y>:
14   ticks = shift (-a) y U shift (-b) y U shift (-b) x U ticksOf x
15   val = let
16     tnow = 1'T now
17     yT = keepTrues y
18     min_yT = if (Utils.shift (-a) y) [~t|False]
19              then tnow else timeOf (Utils.shift (-a) yT >>_(b-a) t)
20     xF = keepFalses x
21     min_xF = if not x[~t|False] then tnow else timeOf (xF >>_b t)
22     plus x tim = 2'timeAdd x 'tim
23     in
24     min_yT `plus` a <= tnow `plus` b && min_yT `plus` a <= min_xF
```

The parameterized stream `mu` represents the application of a function to the values of an input stream. We define the auxiliary streams `keepTrues` and `keepFalses` to filter only the *True* or *False* events of a stream, respectively.

The `until` operator is defined as follows. Given `Boolean` streams `x` and `y`, and given the window offsets `a` and `b`, the property $(x \mathcal{U}_{[a,b]} y)$ is *True* if there is a point t' in the window $[t+a, t+b]$ where y holds, and x holds continuously from t to t'. In particular, if y holds somewhere in the range $[t+a, t]$, the property is *True*. The definition of `until` finds the first time point at which `y` is *True* in the range $[t+a, \infty)$ (which we call `min_yT`) and compare it with the first time point at which `x` is *False* in the range $[t, \infty)$ (which we call `min_xF`). For the property to be true, two things must happen:

- that `min_yT` $\leq t+b$ (i.e., that `y` be *True* somewhere in the window $[t+a, t+b]$), and
- that `min_yT` $<$ `min_xF` (i.e., that `y` be *True* before the first time `x` is *False* after t)

The expression `min_yT` contains the earliest time at which `y` becomes *True* after `t`$+a$ (considering the possibility of (`t`$+a$) itself if `y` is already *True* at that point). Similarly, `min_xF` contains the earliest time at which `x` becomes *False* (considering the possibility of `t` itself if `x` is already *False*). With these auxiliary definitions, the value expression of `until` simply checks that `y` becomes *True* within $[a, b]$ and that `x` is *True* from `t` up-to that point.

The tick expression of the stream indicates the times at which its value can change, namely when a `y` event enters or leaves the sliding window defined by $[t+a, t+b]$, or when a `x` event enters or leaves the sliding window defined by $[t, t+b]$.

3.3 Example #2: 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* 10 *time units (represented by the stream* `slow_down`*)*."

We say that the vehicle is moving *too fast* if its speed is greater than 3, and we define a *safe speed* as a speed under 1. We can write this property in STL as follows:

$$\varphi : (\texttt{speed} > 3) \rightarrow \left(\texttt{decel}\ \mathcal{U}_{[0,10]}\ \texttt{speed} < 1\right)$$

We translate this property into HStriver using the following specification. As opposed to the original STL property, our specification does not assume the existence of a Boolean input signal `decel` that indicates whether the vehicle is decelerating. Instead, it uses the unique input signal `speed` to calculate if the vehicle is decelerating, comparing its value every time the signal changes with its next reported value.

```
1  time domain Double
2  use library STL
3  input Double speed
4  define Bool toofast:
5    ticks = ticksOf speed
6    val = cv > 3
7  define Bool speedok:
8    ticks = ticksOf speed
9    val = cv < 1
10 define Bool decel:
11   ticks = ticksOf speed
12   val = cv > speed[t>|0]
13 define Bool slow_down = until (0,10) decel speedok
14 output Bool ok:
15   ticks = ticksOf toofast U ticksOf slow_down
16   val = toofast [~t|False] `implies` slow_down [~t|True]
```

This example shows a straightforward use of the STL library to define temporal properties as streams.

3.4 RobustSTL

In [40] the authors propose a quantitative semantics of the Signal Temporal Logic, which they call Robust STL. The robustness of a formula φ relative to a trace w and a time t, denoted by $\rho(\varphi, w, t)$, is defined as follows:

$$\begin{aligned}
\rho(\mu, w, t) &\stackrel{\text{def}}{=} f(x_1[t], \ldots, x_n[t]) \text{ where } \mu \equiv f(x_1[t], \ldots, x_n[t]) \geq 0 \\
\rho(\neg\varphi, w, t) &\stackrel{\text{def}}{=} -\rho(\varphi, w, t) \\
\rho(\varphi_1 \wedge \varphi_2, w, t) &\stackrel{\text{def}}{=} \min(\rho(\varphi_1, w, t), \rho(\varphi_2, w, t))) \\
\rho(\varphi_1\, \mathcal{U}_{\mathcal{I}}\, \varphi_2, w, t) &\stackrel{\text{def}}{=} \max_{t' \in t + \mathcal{I}} (\min(\rho(\varphi_2, w, t), \min_{t'' \in [t, t')} \rho(\varphi_1, w, t)))
\end{aligned}$$

We use the extensions in HStriver to define these quantitative semantics of STL. The translation of the operators μ, $\neg$ and $\wedge$ are straightforward – they consist of the application of a function to one or two streams. We show here the definition of $\varphi\, \mathcal{U}_{[a,b]}\, \psi$.

```
1  library RobustSTL
2  use theory Striver
3  use haskell Data.Maybe
4  use innerspec robustuntilspec
5  output Int until <(Time,Time) (a,b)> <Stream Int phi> <Stream Int psi>:
6    ticks = ticksOf phis U ticksOf psis
7    val = 1'(fromMaybe 0.runSpec) (2'robustuntilspec phils psils)
8    where
9      phis = phi[0:b]
10     psis = psi[a:b]
11     phils = 2'stampFst now phis[~t|(Nothing, [])]
12     psils = 2'stampFst (2'tDiffAdd now 'a) psis[~t|(Nothing, [])]
13     stampFst _ (Nothing, r) = r
14     stampFst ts (Just v, r) = (ts,v):r
```

For the implementation of $\varphi\, \mathcal{U}_{[a,b]}\, \psi$ we define two slices: the slice of the events of `phi` in $[\mathtt{t}, \mathtt{t}+b]$ in line 9 and the slice of the events of `psi` in $[\mathtt{t}+a, \mathtt{t}+b]$, in line 10. Whenever an event enters or leaves any of the slices, we need to recompute the value of the stream, which is why we use the aforementioned slices `phis` and `psis` as the ticking points of `until`. Since we interpret the events of a stream as its change points, we use the value of the last event in the past of the slices, which is provided as the first element in the tuple of a slice, to access the value of the signal at the beginning of the sliding window. We override the time of the first event of `phis` with the current time (`now`) in line 11 and the first event of `psis` with $\mathtt{now}+a$ in line 12. The helper function `stampFst` is defined in the last two lines, namely 13 and 14, and simply stamps the previous value of the slice (if it exists) and prepends it to the list of events in the sliding window. Finally, we execute the nested specification `robustuntilspec` with the resulting slices.

The nested specification `robustuntilspec` is defined as follows:

```
1  innerspec Int robustuntilspec
2  input Int phis
3  input Int psis
4  define Bool never:
5    ticks = {0}
6    val = 'False
7  define Int phismins:
8    ticks = ticksOf phis
9    val = 2'min phismins[<t|maxBound] cv
10 define Int theMins:
11   ticks = ticksOf phismins U ticksOf psis
12   val = 2'min psis[~t|maxBound] phismins[~t|maxBound]
13 output Int theMaxMin:
14   ticks = ticksOf theMins
15   val = 2'max theMaxMin[<t|minBound] cv
16 return theMaxMin when never
```

The input streams `phis` and `psis` contain the successive changepoints of the original `phi` and `psi` within a sliding window. We first define in lines 4 to 6 a `Boolean` stream called `never` that is always *False*, which we will use to return the value of the last event of `theMaxMin` via the return statement in line 16. Note that the return clause would be equivalent if we define the value expression of `never` as `notick` instead of *False*.

The intermediate stream `phismins` defined in lines 7 to 9 maintains the successive *historical* minimums of `phi` within the slice, and the intermediate stream `theMins` defined in lines 10 to 12 computes the minimum between the current value of `psi` and the current historical minimum of `phi`.

Finally, `theMaxMin` in lines 13 to 15 keeps the maximum value that was ever taken by `theMins`. The last value of `theMaxMin` is returned as the result of the computation.

3.5 Example #3 : cost computation

The following example calculates the accumulated energy cost incurred by a monitor, based on a cost model for

(a) waking up the monitor,
(b) processing an event,
(c) going to sleep,
(d) being idle,
(e) being awake,

which are encoded in the return values of the functions `runningCost` of type `RunMode` → `Int` and `transiCost`, of type `RunMode` → `RunMode` → `Int`. The cost calculation also depends on how long does the monitor wait for a new event before going to sleep, represented by the constant *patience*. This specification contains the definition of an output stream `cost` 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. This example uses custom datatype definitions, and event generation at instants where there is no input event.

```
time domain Double
use haskell Data.Maybe

data RunMode= Alert | Sleeping deriving (Show,Generic,ToJSON,Eq)

const processEventCost = 20
const wakeUpCost = 100
const gotoSleepCost = 100
fun runningCost Alert = 100
fun runningCost Sleeping = 1

fun transiCost Alert Alert = processEventCost
fun transiCost Sleeping Alert = wakeUpCost + processEventCost
fun transiCost Alert Sleeping = gotoSleepCost
fun transiCost Sleeping Sleeping = error "going to sleep sleeping"

const patience = 10

input () newEvent

define Time sleep_delayer:
  ticks = ticksOf newEvent
  val = 'patience

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

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

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

The stream `sleep_delayer` produces the value `patience` every time there is a new event, and it is used to delay the value of `sleep` by exactly `patience` time units. Since `patience` is a constant in our specification, we could have used the stream `shift` from `Utils` to achieve the same result.

The stream `runMode` indicates if the system is `Alert` or `Sleeping`. When a new event is received, the system is `Alert`, and if the patience after the last event is consumed, then the system goes to sleep.

Finally, `cost` produces an event every time the `runMode` (potentially) changes, with the cost of transitioning from the previous state to the current one, plus the accumulated cost of having been in the previous state until now.

3.6 Example #4: powerTrain

Our fourth example makes a heavy use of the STL library to implement STL properties for the verification of a powertrain control verification from [41], where input signals change asynchronously.

```
time domain Double
use library STL
use library Utils

input Double verification
input Double mode
input Double pedal

const simTime = 50
const eta = 1
const taus = 10 + eta
const zeta_min = 5

define Bool low = mapFun "low" (geq 0.5) pedal
define Bool high = mapFun "high" (leq 0.5) pedal
define Bool utl <Double x> = mapFun "utl" (leq (-x)) verification
define Bool utr <Double x> = mapFun "utr" (geq x) verification
define Bool pwr = mapFun "pwr" (leq 0.5) mode
define Bool norm = mapFun "norm" (geq 0.5) mode

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

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

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

As in [41] 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 refrained from using let and where clauses to maintain the syntax of the original properties, which are commented in MatLab above each definition.

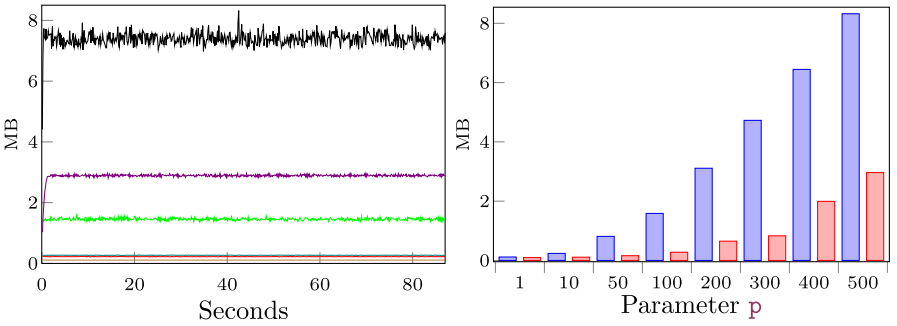
3.7 Example #5: smart home

This example models a smart home specification that uses the Orange4Home data-set [42]. 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`).

```
time domain UTC

use library Utils
use haskell Data.Time

input Bool livingroom_tv_on
input Bool office_tv_on

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]

define Int instantN:
  ticks = ticksOf any_tv_on
  val = instantN [<t|0] + 1

define Bool isNewDay:
  ticks = ticksOf any_tv_on
  val = let
    today = 1'utctDay now
    prev = 1'(utctDay.unT) (timeOf (any_tv_on << t))
    in
    if instantN [~t|] === 1 then 'True else today /== prev

define Int howMuchTvToday:
  ticks = ticksOf any_tv_on
  val = let
    prevVal = if isNewDay [~t|] then 0 else howMuchTvToday [<t|0]
    sumVal = if any_tv_on [~t|] then 1 else 0
    in
    prevVal + sumVal
```

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

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

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
```

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.



(a) Memory wrt time (b) Memory wrt parameter p

Fig. 4 Empirical evaluation of hypotheses (H1), (H2), and (H3)

4 Empirical evaluation

In this section, we report on an empirical evaluation of HStriver. This empirical evaluation is not aimed at comparing the performance of HStriver with similar tools, but it intends to demonstrate that the memory efficiency of HStriver is consistent with the theoretical analysis/prediction; and its memory footprint makes it good enough to be used in real life scenarios. We believe that the best choice of a tool is not exclusively based on its performance, but that expressivity and usability are more decisive, considering similar memory usages. All the experiments were executed on a MacBook Pro with MacOS Catalina Version 10.15.4, with an Intel Core i5 at 2.5 GHz and 8 GB of RAM.

We evaluate empirically the following hypotheses:

- (H1) The memory consumed is constant throughout an execution if we restrict the specification to use only past references, that is, where the memory required is trace length independent.
- (H2) The resources necessary to monitor a specification grows with respect to its number of streams.
- (H3) The resources necessary to monitor a specification grows with respect to the events it needs to keep in memory.
- (H4) The memory consumed is constant throughout the execution of an STL property with the efficient version of the $\mathcal{U}$ operator.
- (H5) The resources necessary to monitor an STL specification grows similarly with respect to variations in the interval size and with respect to variations in the event density.
- (H6) The version of the $\mathcal{U}$ operator that uses bounded future accesses, is more efficient and memory-wise stable than the version of the operator that uses unbounded future accesses.

We run experiments to measure the memory usage and assess hypotheses (H1), (H2), and (H3) for two collections of specifications:

- *Stocks.* The first collection generalizes Example 2 computing the stocks of $\mathtt{p}$ independent products, similar to Example 3, but without the alarms. These specifications contain a number of streams proportional to $\mathtt{p}$, where each defining equation is of the same size.

```
time domain Double
use haskell Data.Maybe

type Product= Int

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
```

- *Average.* The second collection computes the average of the last `p` sales of a fixed product, via streams that tick at the selling instants and compute the sum of the last `p` sales. The resulting specifications have depth proportional to `p`.

```
input Int sale

define Int denom <Int p>:
  ticks = ticksOf sale
  val = 1'(min p) ((denom p)[<t|0] + 1)

define Double sumlastp <Int p>:
  ticks = ticksOf sale
  val = (sumlastp p)[<t|0] + cv - sale[<sale<<sale<<...<<t|0]

output Double avgp <Int p>:
  ticks = ticksOf sale
  val = (sumlastp p)[~t|] / 1'fromIntegral (denom p)[~t|]
```

We instantiate `p` for values between 10 and 500 and run each resulting specification with a set of automatically generated random input traces.

For this experiment we ran the synthesized monitors over long traces, and we measured the memory consumption. The results shown in Fig. 4a illustrate that the memory needed to monitor each specification is independent of the length of the trace, since the curves are roughly constant throughout the entire executions, validating (H1). The bars in Fig. 4b indicate that the memory needed to monitor both specifications increases with the parameter `p`, with *stock* increasing more rapidly than *average*, valdating (H2) and (H3). The reason for this is that the number of streams in the *stock* specification gets higher when `p` gets larger and also has to keep in memory more events (the last event of every stream); while the *average* specification also has to remember a higher number of events when `p` gets larger, but the number of streams in the specification remains constant (even though the stream definition of `sumlastp` gets larger, this introduces negligible overhead compared to the overhead of the number of streams that affects the *stock* specification).

We have designed a set of experiments to assess the validity of the hypotheses (H4), (H5), and (H6), for which we consider the following STL specification from Section 3.3:

$$\varphi : (\texttt{speed} > 3) \rightarrow \left(\texttt{decel}\ \mathcal{U}_{[0,10]}\ \texttt{speed} < 1\right)$$

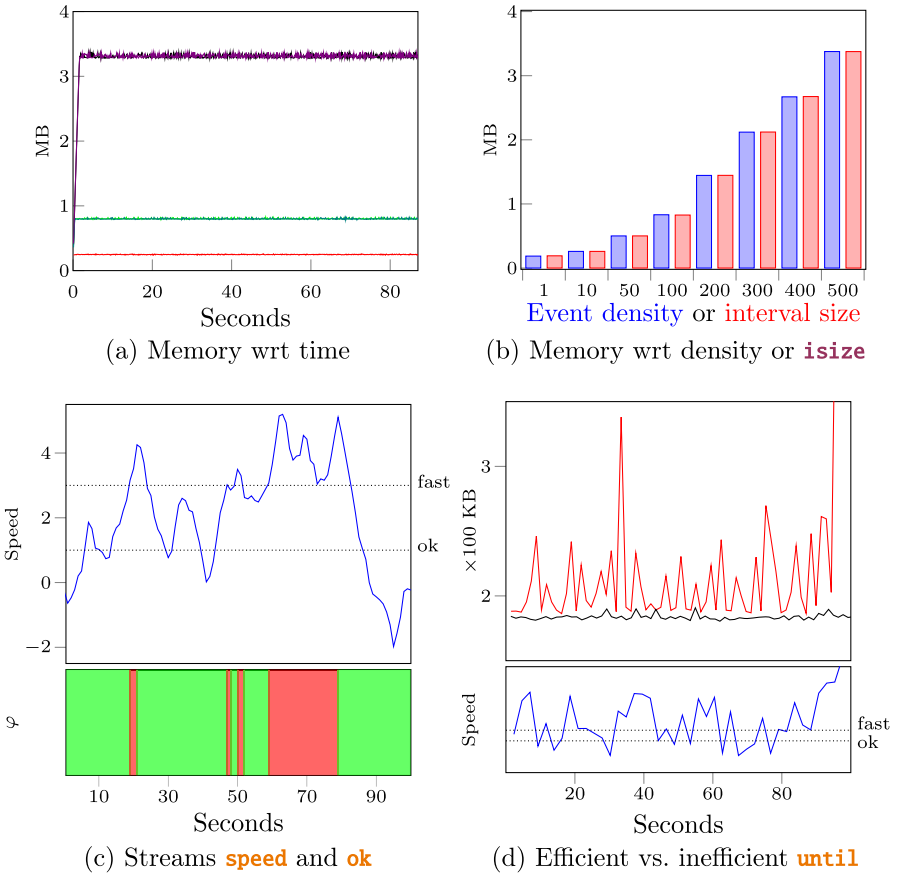


Fig. 5 Empirical evaluation of hypotheses (H4), (H5), and (H6)

In the first set of experiments, we vary the *size of the interval* for the $\mathcal{U}$ operator adding a parameter `isize` to the stream `slow_down` to analyse how it affects the performance of the engine:

```
define Bool slow_down <Time isize> = until (0,isize) decel speedok
```

We also vary the amount of events per second in the input stream, which we call the *density* of the signal, to assess its impact on performance. The input data for the experiments are generated programatically.

Figure 5a shows the memory consumption of the monitor of the specification with diverse interval sizes and event density, which in all cases remains constant throughout the execution, albeit with different constants, validating (H4).

Figure 5b shows the maximum memory consumed by the monitor of the specification with increasing event density and constant interval size (represented by the blue bars) and increasing interval sizes and constant event density (represented by the red bars), illustrating that the amount of memory necessary to monitor this specification increases with the number of events that fit in an interval, either because the interval is larger, or because there are more events per second, which validates (H5).

Figure 5c shows the input signal `speed` at the top, and underneath, the regions where φ holds (in green) or not (in red).

The definition of the parametric `until` stream in the library `STL` uses the $\gg_b$ operator, which ensures that the memory consumption is bounded by the interval size and the event density. If we use the implementation of `until` with unbounded future operators, then the memory requirement depends on the behavior of the input signal and is no longer solely determined by the interval size and input event density.

We have run the specification over the input data shown at the bottom of Fig. 5(d) using the *efficient* (bounded) and *inefficient* (unbounded) version of `until`, and we have measured their memory usage, which can be seen in the plot at the top of Fig. 5(d). The black curve represents the memory consumption of the efficient implementation of the $\mathcal{U}$ operator, and the red curve represents the memory consumption of the inefficient implementation of the operator. In the specification with unbounded future accesses, the $\mathcal{U}$ expression needs to retrieve the next instant at which the vehicle *decelerates* and also the next instant at which its speed is *safe*. If one of these instants is far in the future, the monitor needs to consume and store all the input up to that point. This causes a rapid increase in the memory consumption, which is now visible in the shape of the input signal and unknown beforehand. Once the relevant instants are found, the input signal is consumed up to that point, which causes a rapid decrease in memory consumption. As a result of this, we observe that memory usage presents peaks during the execution, in concordance with the behavior of the input signal. By the end of the execution, the input signal `speed` accelerates continuously, making the engine consume and store the input indefinitely, until it reaches the maximum memory allowed, which was set at 1 GB, without computing a value for the output stream `ok`, even though that much memory is not necessary for the computation, as demonstrated by the memory consumption of the efficient implementation of $\mathcal{U}$, in black. This experiment validates (H6).

5 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. One drawback of our approach is that the Haskell runtime system uses garbage collection, which is usually forbidden for critical applications. However, HStriver has been used in (non-critical) UAV missions, and we are exploring the generation of Misra-C code from (a restricted set of) HStriver specifications.

Another undesired byproduct of having developed HStriver as an eDSL is that we do not have full control over the syntax and we have to adapt it to that of Haskell. This also means that errors are reported by the Haskell compiler and are usually cryptic and hard to follow. Future work includes the development of a frontend that allows adapting the input syntax to particular use cases, offering a friendly syntax and the necessary types and features from HStriver, while also giving us complete control over error reports.

An alternative way of defining the semantics of the underlying language HStriver is by searching for a fixpoint using small-step semantics. Future work includes defining the semantics of HStriver following this alternative approach and analyse how this definition impacts on the design, development and assessment of the tool HStriver.

References

1. Havelund K, Goldberg A (2005) Verify your runs. In: Proceedings of verified software: theories, tools, and experiments (VSTTE'05), LNCS, vol 4171. Springer, Heidelberg, pp 374–383
2. Leucker M, Schallhart C (2009) A brief account of runtime verification. J Log Algebr Program 78(5):293–303
3. Bartocci E, Falcone Y (eds) (2018) Lectures on runtime verification—introductory and advanced topics. LNCS, vol 10457. Springer, Cham
4. Manna Z, Pnueli A (1995) Temporal verification of reactive systems. Springer, New York
5. Bauer A, Leucker M, Schallhart C (2011) Runtime verification for LTL and TLTL. ACM Trans Softw Eng Methodol 20(4):14
6. Eisner C, Fisman D, Havlicek J, Lustig Y, McIsaac A, Campenhout DV (2003) Reasoning with temporal logic on truncated paths. In: Proceedings of the 15th international conference on computer aided verification (CAV'03). LNCS, vol 2725. Springer, Heidelberg, pp 27–39
7. Havelund K, Roşu G (2002) Synthesizing monitors for safety properties. In: Proceedings of the 8th international conference on tools and algorithms for the construction and analysis of systems (TACAS'02). LNCS, vol 2280. Springer, Heidelberg, pp 342–356
8. Sen K, Roşu G (2003) Generating optimal monitors for extended regular expressions. In: Electronic notes in theoretical computer science, vol 89. Elsevier, Amsterdam
9. Roşu G, Havelund K (2005) Rewriting-based techniques for runtime verification. Automat Softw Eng 12(2):151–197
10. Barringer H, Goldberg A, Havelund K, Sen K (2004) Rule-based runtime verification. In: Proceedings of the 5th international conference on verification, model checking and abstract interpretation (VMCAI'04). LNCS, vol 2937. Springer, Heidelberg, pp 44–57
11. Barringer H, Rydeheard D, Havelund K (2007) Rule systems for run-time monitoring: from Eagle to RuleR. In: Proceedings of the 7th international workshop on runtime verification (RV'07). LNCS, vol 4839. Springer, Heidelberg, pp 111–125
12. D'Angelo B, Sankaranarayanan S, Sánchez C, Robinson W, Finkbeiner B, Sipma HB, Mehrotra S, Manna Z (2005) LOLA: runtime monitoring of synchronous systems. In: Proceedings of the 12th international symposium of temporal representation and reasoning (TIME'05). IEEE CS Press, Washington DC, pp 166–174
13. Sánchez C (2018) Online and offline stream runtime verification of synchronous systems. In: Proceedings of the 18th international conference on runtime verification (RV'18). LNCS, vol 11237. Springer, Cham, pp 138–163
14. Berry G (2000) The foundations of Esterel. Proof, language, and interaction: essays in honour of Robin Milner. MIT Press, Cambridge, MA, pp 425–454
15. Halbwachs N, Caspi P, Pilaud D, Plaice JA (1987) Lustre: a declarative language for programming synchronous systems. In: Proceedings of the 14th ACM symposium on principles of programming languages (POPL'87). ACM Press, New York, NY, pp 178–188
16. Convent L, Hungerecker S, Leucker M, Scheffel T, Schmitz M, Thoma D (2018) TeSSLa: temporal stream-based specification language. In: Proceedings of the 21st. Brazilian symposium on formal methods (SBMF'18). LNCS, vol 11254. Springer, Cham
17. Leucker M, Sánchez C, Scheffel T, Schmitz M, Schramm A (2018) TeSSLa: runtime verification of non-synchronized real-time streams. In: Proceedings of the 33rd symposium on applied computing (SAC'18). ACM Press, New York, NY
18. Gorostiaga F, Sánchez C (2018) Striver: stream runtime verification for real-time event-streams. In: Proceedings of the 18th international conference on runtime verification (RV'18). LNCS, vol 11237. Springer, Cham, pp 282–298
19. Faymonville P, Finkbeiner B, Schledjewski M, Schwenger M, Stenger M, Tentrup L, Hazem T (2019) StreamLAB: stream-based monitoring of cyber-physical systems. In: Proceeings of the 31st international conference on computer-aided verification (CAV'19). LNCS, vol 11561. Springer, Cham, pp 421–431
20. Gorostiaga F, Sánchez C (2021) Stream runtime verification of real-time event-streams with the Striver language. Int J Softw Tools Technol Transf 23:157–183
21. Gorostiaga F, Sánchez C (2021) HStriver: a very functional extensible tool for the runtime verification of real-time event streams. In: Proceedings of the 24th international symposium on formal methods (FM'21). LNCS, vol 13047. Springer, Cham, pp. 563–580. https://doi.org/10.1007/978-3-030-90870-6_30
22. Havelund K (2015) Rule-based runtime verification revisited. Int J Softw Tools Technol Transf 17(2):143–170

23. Barringer H, Havelund K (2011) TraceContract: a scala DSL for trace analysis. In: Proceedings of the 17th international symposium on formal methods (FM'11). LNCS, vol 6664. Springer, Heidelberg, pp. 57–72
24. Stolz V, Huch F (2005) Runtime verification of concurrent Haskell programs. Electronic notes on theoretical computer science, vol 113. Elsevier, Amsterdam, pp 201–216
25. Ceresa M, Gorostiaga F, Sánchez C (2020) Declarative stream runtime verification (hLola). In: Proc. of the 18th Asian Symposium on Programming Languages and Systems (APLAS'20). LNCS, vol 12470. Springer, Cham, pp 25–43
26. Falcone Y, Krstic S, Reger G, Traytel D (2018) A taxonomy for classifying runtime verification tools. In: Proceedings of the 18th international conference on runtime verification (RV'18). LNCS, vol 11237. Springer, Cham, pp 241–262
27. Reinbacher T, Rozier KY, Schumann J (2014) Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Proceedings 20th International Confer on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14). LNCS, vol 8413. Springer, Heidelberg, pp 357–372
28. Hallé S (2016) When RV meets CEP. In: Proceedings of the 16th international conference on runtime verification (RV'16). LNCS, vol 10012. Springer, Cham, pp 68–91
29. Hallé S, Khoury R (2017) Event stream processing with BeepBeep 3. In: Proceedings of the international workshop on competitions, usability, benchmarks, evaluation, and standardisation for runtime verification tools (RV-CUBES). Kalpa Publications in Computing. EasyChair, pp 81–88
30. Basin D, Harva M, Klaedtke F, Zalinescu E (2011) MONPOLY: monitoring usage-control policies. In: Proceedings of the 2nd international conference on runtime verification (RV'11). LNCS, vol 7186. Springer, Heidelberg, pp 360–364
31. Basin DA, Klaedtke F, Zalinescu E (2017) The MonPoly monitoring tool. In: Proceedings of the international workshop on competitions, usability, benchmarks, evaluation, and standardisation for runtime verification tools (RV-CUBES). Kalpa Publications in Computing. EasyChair, pp 19–28
32. Pike L, Goodloe A, Morisset R, Niller S (2010) Copilot: a hard real-time runtime monitor. In: Proceedings of the 1st international conference on runtime verification (RV'10). LNCS, vol 6418. Springer, Heidelberg, pp 345–359
33. Faymonville P, Finkbeiner B, Schwenger M, Torfah H(2017) Real-time stream-based monitoring. CoRR arXiv:1711.03829
34. Aguado J, Mendler M, Pouzet M, Roop P, von Hanxleden R (2018) Deterministic concurrency: a clock-synchronised shared memory approach. In: Ahmed A (ed) Programming languages and systems. Springer, Cham, pp 86–113
35. Talpin J-P, Brandt J, Gemünde M, Schneider K, Shukla S (2013) Constructive polychronous systems. In: Artemov S, Nerode A (eds) Logical foundations of computer science. Springer, Berlin, Heidelberg, pp 335–349
36. Marlow S (2010) Haskell language report
37. Gorostiaga F, Sánchez C (2021) Nested monitors: monitors as expressions to build monitors. In: Proceedings of the 21st international conference on runtime verification (RV'21). LNCS, vol 12974. Springer, Heidelberg, pp 164–183
38. Gorostiaga F (2022) Theory and practice of stream runtime verification for sequences and real-time event based systems. https://oa.upm.es/70504/
39. Maler O, Nickovic D (2004) Monitoring temporal properties of continuous signals. In: Proceedings of the joint international conference on formal techniques, modelling and analysis of timed and fault-tolerant systems, and formal modelling and analysis of timed systems (FORMATS/FTRTFT 2004). LNCS, vol 3253. Springer, Heidelberg, pp 152–166
40. Donzé A, Ferrère T, Maler O (2013) Efficient robust monitoring for STL. In: Proceedings of the 25th international conference on computer aided verification (CAV'13). LNCS, vol 8044. Springer, Heidelberg, pp 264–279
41. Jin X, Deshmukh JV, Kapinski J, Ueda K, Butts K (2014) Powertrain control verification benchmark. In: Proceedings of the 17th international conference on hybrid systems: computation and control (HSCC'14). ACM, New York, pp 253–262
42. Cumin J, Lefebvre G, Ramparany F, Crowley J (2017) A dataset of routine daily activities in an instrumented home. In: Proceedings of the 11th international conference on ubiquitous computing and ambient intelligence (UCAmI'17). LNCS, vol 10586. Springer, Cham, pp 413–425