International Journal on Software Tools for Technology Transfer (2023) 25:503–517
https://doi.org/10.1007/s10009-023-00710-5

GENERAL

Special Issue: RV 2020

MoonLight: a lightweight tool for monitoring spatio-temporal properties

Laura Nenzi[1,2] · Ezio Bartocci[2] · Luca Bortolussi[1] · Simone Silvetti[3] · Michele Loreti[4]

Accepted: 30 May 2023 / Published online: 22 June 2023

Abstract
We present MoonLight, a tool for monitoring temporal and spatio-temporal properties of mobile, spatially distributed, and interacting entities such as biological and cyber-physical systems. In MoonLight the space is represented as a weighted graph describing the topological configuration in which the single entities are arranged. Both nodes and edges have attributes modeling physical quantities and logical states of the system evolving in time. MoonLight is implemented in Java and supports the monitoring of Spatio-Temporal Reach and Escape Logic (STREL). MoonLight can be used as a standalone command line tool, such as Java API, or via Matlab™ and Python interfaces. We provide here the description of the tool, its interfaces, and its scripting language using a sensor network and a bike sharing example. We evaluate the tool performances both by comparing it with other tools specialized in monitoring only temporal properties and by monitoring spatio-temporal requirements considering different sizes of dynamical and spatial graphs.

Keywords Spatio-temporal logic · Specification-based monitoring

1 Introduction

Dynamical systems often display complex spatio-temporal behavioral patterns emerging as a collective and cooperative phenomenon of locally interacting components. Monitoring these behaviors plays a key role in predicting the overall system behavior at the macroscopic level or in understanding the underlying mechanisms occurring at the microscopic scale.

These patterns are ubiquitously present in nature: Turing patterns emerging in morphogenesis [6, 13], birds flying in V-formation [37], cooperative foraging in animal groups [32], and electrical spiral waves propagating in excitable media [5, 27] are typical fascinating examples. Spatio-temporal properties are likewise important in human-engineered artifacts such as Collective Adaptive Systems [36] (CAS) and Cyber-Physical Systems [49] (CPS), which often resemble many features of the natural ones.

CAS and CPS aggregate several heterogeneous and spatially distributed entities that are dynamically networked and can cooperate among themselves, with humans, or with other systems. Examples of these systems can be found in the internet of things, biking sharing systems, vehicular networks, and smart cities. CPS are also often safety-critical [49] systems where hardware/software failures can be responsible for tragic accidents such as loss of lives, the injury of people, or environmental damages.

Running example We consider as our running example throughout this paper the monitoring of a mobile ad hoc wireless sensor network [2] consisting of three different types of nodes: *coordinator*, *router*, and *end-device*.

The coordinator is the node responsible for initiating the network and routing protocol: each network can have only one single coordinator. Router nodes are responsible for forwarding data packets received from other devices, establishing a backbone of intermediate nodes necessary to reach all the other ones. End-devices are generally the nodes used to

✉ L. Nenzi
lnenzi@units.it

E. Bartocci
ezio.bartocci@tuwien.ac.at

L. Bortolussi
luca@dmi.units.it

S. Silvetti
simone.silvetti@gmail.com

M. Loreti
michele.loreti@unicam.it

1 University of Trieste, Trieste, Italy

2 TU Wien, Vienna, Austria

3 Esteco S.p.A., Trieste, Italy

4 University of Camerino, Camerino, Italy

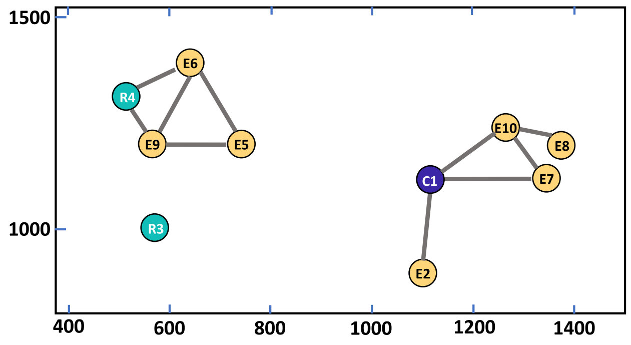


Fig. 1 Sensor network with 1 coordinator (C, in violet), 2 routers (R, in cyan), and 7 end devices (E, in yellow)

sense and control physical processes. They can only communicate directly with the coordinator and other routers, but they cannot relay data packets from other devices.

In our example, we assume that all the nodes participating in the network are battery-powered and equipped with sensors with which they can measure and collect data from the environment (e.g., pollution, temperature, etc.). Figure 1 illustrates a network with 10 nodes (1 coordinator (C) in violet, 2 routers (R) in cyan, and 7 end devices (E) in yellow). The color version of the article is given online. The nodes are distributed in an Euclidean space, i.e., the axes represent their coordinates in the space. The edges represent the connectivity graph of the network, expressing the fact that two devices can directly *interact* (i.e., they are within their communication range).

Specification-based monitoring Performing an exhaustive analysis at design time of complex networked and spatially distributed systems is generally impractical due to the state-space explosion. A more feasible but nonexhaustive approach is to instrument such systems and to test them using specification-based monitoring [10]: systems' execution traces are recorded and stored during their simulation or execution and are monitored offline with respect to a requirement specified in a formal language [44] such as, for example, Signal Temporal Logic (STL) [40].

Offline monitoring is typically performed on the numerical simulation (its digital twin) of the system behavior to test the correctness of its design under different initial conditions, parameter values, and input sequences [10, 11].

In the last years, specification-based monitoring [10] became the basic functionality upon which many other computer-aided methods are developed for the analysis and synthesis employed in CPS engineering such as falsification analysis [1, 51, 52, 54], fault-localization [12], failure explanation [14, 16], and parameter synthesis [7, 19, 23, 24].

The verdict of the monitoring process can be either a Boolean value (true/false) stating whether the execution trace satisfies or violates the formal requirement or a real value measuring how much the trace is close to satisfying or violating the specification according to a chosen notion of distance and formal specification semantics [9, 26, 30, 31, 50]. The majority of the currently available monitoring tools can only handle temporal requirements.

In this paper, we present MoonLight, a lightweight tool for monitoring temporal and spatio-temporal properties of spatially distributed entities, which can move in space and change their connectivity. Our implementation is available at https://github.com/MoonLightSuite/MoonLight.

MoonLight supports monitoring of Spatio-Temporal Reach and Escape Logic (STREL), a spatio-temporal specification language presented in [8, 45]. STREL extends STL [40] with a number of spatial operators, allowing us to describe spatio-temporal behaviors such as "there is always a path of routers that connect an end device with a coordinator with a battery level more than 50%" or "in a bike sharing system, if in a station there are no bikes, then I can find a bike in a close station at a distance less than 500 meters". MoonLight takes as input a STREL formula and a spatio-temporal trajectory. The space is represented as a weighted graph, describing the topological configurations in which spatially distributed entities are arranged. Nodes represent single entities (e.g., sensors in a sensor network or bike stations in a bike sharing system), and edges represent the connection between the entities. Both nodes and edges have attributes modeling physical and logical quantities that can change in time (e.g., values of the battery in the sensor, the Euclidean distance between the sensors, or the number of bikes in the bike station). Therefore a spatio-temporal signal, in the most general case, is described by a sequence of such weighted graphs, allowing both spatial arrangements and attributes to change in time. MoonLight monitors such a sequence of graphs with respect to a STREL formula, returning a Boolean or quantitative verdict according to the semantic rules of [45].

The main component of MoonLight is its Java Application Programming Interface (API), a set of specialized classes and interfaces to manage *data domains* and *signals*, to represent *spatial models* that can evolve in time, to *monitor temporal and spatio-temporal properties*, and to manage input/output to/from *generic data sources*. Moreover, it also contains a *compiler* that generates the necessary Java classes for monitoring from a MoonLight script. The latter are built and dynamically loaded to enable the monitoring of the specified properties. MoonLight provides also an interface that enables the integration of its monitoring features in Matlab™and Python. So MoonLight can be used as a standalone command line tool, as a Java API, via Matlab™ or Python interfaces.

This paper extends the paper presented at RV2020 [15] as follows:

- we provide a description of the Java architecture of the tool;
- we describe how to use MoonLight via console and as a Java API;
- we provide an extensive description of the MoonLight script language;
- we develop API to use MoonLight from Python;
- we provide a new bike sharing monitoring example in Python with real data using a static spatial graph;
- we extend the experimental evaluation of the spatial operators considering both dynamical and static spatial models.

Organization of the paper The rest of the paper is structured as follows. In Sect. 2, we present the related work. In Sect. 3, we introduce some background, in particular, the logic STREL. In Sect. 4, we give an overview of the implementation, and in Sect. 5, we describe the MoonLight script language. In Sect. 6, we present all different ways to use MoonLight, in Java API, in the console, and in MatLab™ and *Python* environments. In Sect. 7, we show the evaluation of the tool and in Sect. 8 the conclusion and future works.

2 Related work

The tools currently available for specification-based monitoring are generally limited to verifying temporal requirements over mixed/analog time series of data. They do not consider the spatial configuration of the entities such as sensors and computational units. Examples of monitoring tools for temporal properties include R2U2 [41] for Mission Linear Temporal Logic (MLTL) [41], S-Taliro [3] for Metric Temporal Logic (MTL) [33], AMT [47], RAMT [46], and Breach [21] for Signal Temporal Logic (STL) [39, 40], TeSSLa [35] and RTLola [17] for temporal stream-based specification languages, and Montre [53] for Timed Regular Expressions (TRE) [4]. However, the complex spatio-temporal patterns emerging in cyber-physical or collective adaptive systems cannot be expressed using temporal specification languages. As a consequence, in the last decade, there were many attempts to extend temporal specification logics such as STL to capture also spatial requirements: Spatial-Temporal Logic (SpaTeL) [28], the Signal Spatio-Temporal Logic (SSTL) [43], the Spatial Aggregation Signal Temporal Logic (SaSTL) [16, 38], and STREL [8] are some examples.

Although there are software prototypes supporting the monitoring for these specification languages, they are usually developed more with the aim to provide a proof-of-concept rather than becoming stable and usable tools. The only tool we are aware of is jSSTL [42], an offline monitoring tool for spatio-temporal properties. Whereas MoonLight can monitor STREL formulae over *dynamic networks*, jSSTL [42] can monitor SSTL formulae over a *static* topological space.

It is also worth mentioning VoxLogicA [18], a spatial model checking tool for image analysis. However, this tool is customized for medical imaging and does not take into consideration time. Thus it is not suitable for monitoring spatio-temporal properties.

3 Background material

3.1 Spatial model

In MoonLight the space is modeled as a graph, where each node represents a location while each edge represents a topological relation. Edges can be labeled with one or more attributes. Formally, we define a *spatial model* $\mathcal{S}$ as a pair $\langle L, \mathbf{W}\rangle$ where L is a set of *locations* and $\mathbf{W} \subseteq L \times \mathbb{R}^m \times L$ is a *proximity function* associating at most one label $w \in \mathbb{R}^m$ with each distinct pair $\ell_1, \ell_2 \in L$. The meaning of the weight w depends on the type of analysis. For instance, w can be the Euclidean distance between locations, the hops for connected nodes, or any tuple of values characterizing the *edge* between two locations. In the following, we will equivalently write

$$(\ell_1, w, \ell_2) \in \mathbf{W} \text{ as } \mathbf{W}(\ell_1, \ell_2) = w.$$

In the sensor network example, each device of the network represents a node/location. The edges are labeled with both their Euclidean distance and with the integer value 1. This last value is used to compute the hop (shortest path) count between two nodes, that is, the number of intermediate network nodes through which data must pass between a source node and the target one. A *dynamical spatial model* $\mathcal{S}(t)$ associates a different spatial model at each time.

3.2 Spatio-temporal trace

A spatio-temporal trace describes the evolution in time of a number of variables in each location of the spatial model. Formally, given the (dynamical) spatial model $\mathcal{S}(t) = \langle L, \mathbf{W}(t)\rangle$, we define a *spatio-temporal trace* as a function $\vec{x} : L \to \mathbb{T} \to D^n$ that associates with each location $\ell \in L$ a set of temporal signal $\vec{x}(\ell) = (v_1, \dots, v_n)$.

In our running example, each device (node/location of the network) contains three signals evolving in time: the type of node (coordinator, router, end-device), the level of battery, and the values of the temperature: $\vec{x}(\ell, t) = (v_{type}, v_{battery}, v_{temperature})$.

```
Formula ::=
          relExpression
        | ! Formula
        | Formula & Formula
        | Formula | Formula
        | Formula -> Formula
        | Formula until Interval Formula
        | Formula since Interval Formula
        | eventually Interval Formula
        | globally Interval Formula
        | once Interval Formula
        | historically Interval Formula
        | escape(distanceExpression)Interval Formula
        | Formula reach (distanceExpression)Interval Formula
        | somewhere(distanceExpression) Interval Formula
        | everywhere (distanceExpression) Interval Formula
        | { Formula }
relExpression ::= Expr <Expr | Expr <= Expr | Expr >= Expr | Expr == Expr | != Expr
Interval ::= [Expr, Expr]
Expr ::= baseExpr | Expr + Expr | Expr * Expr
        | UnaryMathFunction (Expr) | BinaryMathFunction (Expr,Expr)
baseExpr ::= true | false | INT | REAL | INF | literalExpr
```

Fig. 2 STREL syntax

3.3 Spatio-temporal reach and escape logic (STREL)

Moonlight evaluates properties specified in the linear-time spatio-temporal logic STREL over spatio-temporal signals and static or dynamical spatial model. It can also directly handle only temporal signals.

The syntax of STREL is summarized in Fig. 2. The atomic expressions (`relExpression`) consist of *relation expressions* on signal variables like, for instance, `(battery > 0.5)` or `(nodeType == 2)`. Formulas are built by using standard Boolean operators (negation `!`, conjunction `&`, disjunction `|`, and implication `->`) together with a set of temporal and spatial modalities.

Temporal properties are specified via the standard `until` and `since` operators (see, e.g., [39, 40]), from which we can derive the future `eventually` and `globally` operators and the past variants, `once` and `historically`. All these operators may take an interval of the form `Interval = [Expr, Expr]`, where `Expr` is a real expression that will be evaluated to a nonnegative value. The interval can be omitted in case of unbounded temporal operators.

Spatial modalities, instead, are `somewhere`, `everywhere`, `reach`, and `escape` operators. All these operators may be decorated with a *distance* `Interval` and a `distance expression`. The distance expression contains the *variables* associated with each edge and is used to compute the *length* of an edge. If omitted, the real value `1.0` is used. To describe the spatial operators, we consider some examples.

The `reach` operator allows us to express properties related to the existence of a path. Consider the following property:

```
P1 = (nodeType==3)reach(hop)[0,1]
{(nodeType==1) | (nodeType==2)}
```

`P1` holds if from a node of type `3` (an *end device*) we can reach a node of type `1` or `2` (a *coordinator* or a *router*) following a path in the spatial graph such that the `hop` distance along this path (i.e., its number of edges) is not greater than 1. This property specifies that *"end device should be directly connected to a router or the coordinator"*.

The `escape` operator can be used to express the ability to move away from a given point. Let us consider the following property:

```
P2 = escape(hop)[5,inf] (battery > 0.5)
```

P2 states that from a given location we can find a path of (hop) length of at least 5 such that all nodes along the path have a battery level greater than 0.5, i.e., that a message will be forwarded along with a connection with no risk of power failure.

To specify properties *around* a given location, the operators `somewhere` and `everywhere` can be used. For instance, we can consider the following property:

```
P3 = somewhere(dist)[0,250](battery>0.5)
```

P3 is satisfied (at a given location) whenever there is a node at a distance between `0` and `250` having a `battery` greater than `0.5`. In this formula the distance is computed by summing the value `dist` of traversed edges. The `everywhere` operator works in a similar way; however, it requires that its subformula holds in all nodes satisfying the distance constraints.

Note that both `reach` and `escape` are existential operators, as they predicate the existence of a path with certain properties, and all the properties are interpreted at a given location, at a given time. Temporal and spatial operators can be nested, for example, as

```
PT1 = (battery <= 0.5)reach(hop)[0, 10]
eventually(battery > 0.5)
```

PT1 holds if each node can reach a node in less than 10 hops where the battery is greater than 0.5 in at least one time step in the next 5 time units. We will show a second example later, but for more formal details and examples about STREL, we refer to [8] and the tool documentation.

STREL as STL [39, 40] has two semantics: the classical Boolean semantics and the quantitative ones. Given a (dynamical) spatial model $\mathcal{S}$, a spatio-temporal trace $\vec{x}$, and a STREL formula ϕ, it returns a Boolean spatio-temporal signal in case of the Boolean semantics, which means a spatio-temporal signal that gives the Boolean satisfaction of property ϕ in each location at each time. In case of the quantitative semantics, the tool returns a real-valued spatio-temporal signal with the quantitative satisfaction of property ϕ in each location at each time. As in STL, the satisfaction of the whole formula corresponds to the satisfaction at time 0. We will see in the description of the script language that the user can easily choose the semantics to use. In the Appendix, we provide the formal definition of the semantics, and we refer the reader to [45] for further details about the STREL logic.

4 Implementation overview

The architecture of MoonLight consists of three main components: *Core API*, a *MoonLight Compiler*, and a *Front-End Layer*.

The *Core API* provides all the Java classes and interfaces that are used to represent temporal and spatio-temporal signals and to manage the different kinds of input data, together with the classes that allow executing the monitoring algorithms. It provides also the classes used to read or write signals on external sources (such as files). We will see later that *Core API* can be used to integrated MoonLight features in Java applications.

MoonLight Compiler enables the use of our tool to users that are not familiar with Java programming. This compiler allows generating a monitor starting from a textual representation via a MoonLight *Script* (see Sect. 5).

Finally, the *Front-End Layer* encompasses components that can be used to interact with MoonLight with different tools. Currently, the layer provides a *console interface*, to use the tool from a terminal, and two modules that allow integrating MoonLight within MatLab and Python scripts.

The source code is publicly available in the *MoonLight GitHub Repository*. The building and testing processes of MoonLight are automated using gradle[1] framework that allows us to automatically download all the required libraries providing the necessary jar files. Moreover, gradle enables the automatic execution of all the tests integrated in MoonLight. Thus a *test suite* is used to simplify code maintenance and integration of new features.

In what follows, we describe the main features of MoonLight, whereas some of the technical details are omitted.

MoonLight core API As we have already anticipated, this module provides a Java API containing all the basic features of MoonLight, which can be included to use our tool in a Java application. By following the *open-close principle* and by relying on well-known design patterns, MoonLight Core API can be easily extended with new features. For instance, this allows us to integrate new operators and consider different sources for signals or a new representation of spatial models.

The module consists of three main packages:

- signal, containing interfaces and classes that can be used to represent and handle temporal and spatio-temporal *signals*;
- monitoring, providing interfaces and classes implementing temporal and spatio-temporal monitors;
- io, supplying the interfaces and classes that can be used to read/write signals from/to data sources, such as *files*.

The interface `Signal<S>` is used to represent a generic *temporal signal* associating each time value t with a value of type `S`. The default implementation of this interface is a *piecewise constant signal* represented as a sequence of time segments, each containing a value `v: S`.

The interface `SpatioTemporalSignal<S>` is used to represent a generic *spatio-temporal signal* associating each location that is univocally identified by an index with a `Signal<S>`. Like for the temporal case, the default implementation of `SpatioTemporalSignal<S>` associates its location with a *piecewise constant signal*.

Spatial models are represented in terms of interface `SpatialModel<V,E>`, identifying a *graph* where locations (vertexes) are associated with values of type `V`, whereas *edges* are labeled with values of type `E`. The exact implementation of this class can be selected according to the specific user's requirements. A number of default implementations, based on different representations of graphs, are provided in the module. To represent the evolution of the spatial model in time, the interface `LocationService<V,E>` can be used. This interface associates each time value t with a `SpatialModel<V,E>`.

Finally, the package `signal` also provides utility classes that enable the access to the values of a signal using Java *Iterators* and Java *Streams*.

[1] https://gradle.org/

Interfaces and classes defined in package signal are used in package `monitoring` to implement a monitoring algorithm. Temporal and spatio-temporal monitors are defined via the *functional interfaces*[2] `TemporalMonitor<S,T>` and `SpatiolTemporalMonitor<V,E,S,T>`.

The interface `TemporalMonitor<S,T>` has a single method `monitor` that, given an input signal `Signal<S>`, computes the output signal `Signal<T>`. Utility methods and classes are used to combine monitors to build new ones. For instance, the following code is used to build a monitor that computes the *conjunction* of two given monitors `m1` and `m2`:

```
<S,T> TemporalMonitor<S,T> andMonitor(
      TemporalMonitor<S,T> m1,
      SignalDomain<T> domain,
      TemporalMonitor<S,T> m2) {
          return new
                 TemporalMonitorBinary<S,T>( m1 ,
                 domain::conjunction , m2 );
}
```

The class `TemporalMonitorBinary<S,T>` combines the signals resulting from the evaluation of monitors `m1` and `m2` by using the given *binary operator* (that is an instance of `BinaryOperator<T>`). This operator depends on the used `SignalDomain<T>`. The latter is an interface representing the signal domain used to interpret formula operators on data of type `T`.

The behavior of *spatio-temporal monitoring* is similar. Indeed, the interface `SpatiolTemporalMonitor<V,E,S,T>` has a single method `monitor` that takes as parameters

a `LocationService<V,E>`

and

a `SpatioTemporalSignal<S>`

and returns

a `SpatioTemporalSignal<T>`.

The monitoring procedure recursively follows the syntax tree of the formula, considering as inputs the output signals of the subformula. Each operator has a specific monitoring algorithm. The monitoring algorithms for the temporal operators are implemented using the same approach of [22, 25], which leverages Lemire's algorithm [34]. The monitoring of the spatial operators follows the approach presented in [45], which uses a spatial flooding algorithm, i.e., the monitored values are propagated on the graph until a fixed point is reached. The algorithms for `escape`, `somewhere`, and `everywhere` operators need also to compute the matrix of minimum distances. The number of steps needed to evaluate the method `monitor` is linear in the size of the formula, in the length of the signal, and in the number of *edges* in the spatial model, and it is quadratic in the number of locations. For more detail about the monitoring algorithms and their correctness and complexity, we refer the reader to [45].

[2] We recall that in Java a *functional interface* is an interface containing a single abstract method.

Finally, the package `io` provides the classes that allow loading and saving temporal and spatio-temporal signals in different formats. Currently, JSon and CSV formats are supported.

MoonLight compiler The classes and interfaces described above can be used to integrate a MoonLight monitor in a Java application. However, often we are not interested in writing a Java program to monitor a set of trajectories. For this reason, to simplify the use of our tool, we have developed a simple compiler that, given a textual representation of a MoonLight monitor (a MoonLight script), produces the appropriate instances of the classes in the MoonLight Core API. The exact syntax of MoonLight script is reported in Sect. 5. Here we briefly outline the compiling procedure and its main components.

MoonLight Compiler is based on AntLR [48], a powerful and largely used parser generator. The generation process consists of three steps: parsing, validation, and generation.

In the parsing phase, the script is loaded, and an abstract syntax tree (AST) is generated. This activity is performed by relying on the classes generated by AntLR. In the validation phase, the generated AST is *visited* to verify the correct use of symbols and to implement a type-checking procedure. Finally, monitoring classes are instantiated and used by the *Front-End Layer*.

A MoonLight script can be either loaded from a file or from a string by relying on the utility methods `ScriptLoader.loadFromFile(String fileName)` and `ScriptLoader.loadFromString(String code)`.

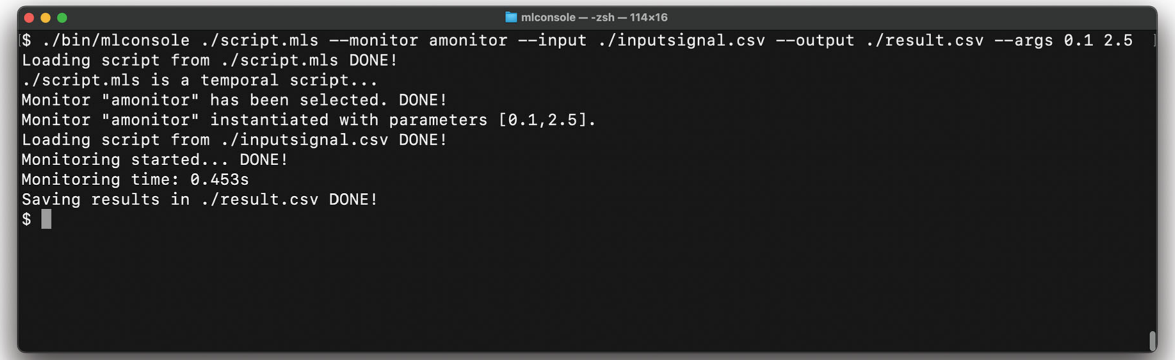
Front-end layer The *Front-End Layer* provides the components that allow using MoonLight in different contexts. Currently, this layer contains a Java *Console Application* that allows executing monitoring from a number of given trajectories. A screenshot of the execution is reported in Fig. 3. We refer to the MoonLight online documentation for a complete list of parameters and a detailed description of the use of MoonLight Console Application.

Moreover, in the *Front-End Layer*, two components are also available to integrate MoonLignt in MatLab and Python. A detailed description of the two modules and their use is available in Sect. 6.

5 Moonlight script

In this section, we present the script language. A Moonlight script consists of five main sections:

```
<TypeDeclaration>
<SignalDeclaration>
<SpaceDeclaration>
<DomainDeclaration>
<FormulasDeclaration>
```



```
$ ./bin/mlconsole ./script.mls --monitor amonitor --input ./inputsignal.csv --output ./result.csv --args 0.1 2.5
Loading script from ./script.mls DONE!
./script.mls is a temporal script...
Monitor "amonitor" has been selected. DONE!
Monitor "amonitor" instantiated with parameters [0.1,2.5].
Loading script from ./inputsignal.csv DONE!
Monitoring started... DONE!
Monitoring time: 0.453s
Saving results in ./result.csv DONE!
$
```

Fig. 3 A run of MoonLight Console Application

describing the custom types declared by the user, the type of monitored signals, the type of space (for spatial/spatio-temporal monitoring), the output domain, and the declaration of formulas.

Type declaration The user can declare his own type, which can be used later as types for signal and space.

```
type <name> = <name> | ... | <name>;
```

Signal declaration In the definition of the domains of input signal, we should define the type of each variable of our trajectory with which we associate a name:

```
signal{
    <VariableDeclaration>;
    ...
    <VariableDeclaration>;
}
```

where each <VariableDeclaration> takes the standard form <type> <name>. Three basic types are natively supported in our specification language: `bool` for Booleans, `int` for integers, and `real` for real values. Moreover, the name of one of the declared types can be used.

Space declaration Similarly to the variables, we can declare the type of labels on the edges:

```
space {
    edges {
        <VariableDeclaration>;
        ...
        <VariableDeclaration>;
    }
}
```

Note that if we are only interested in temporal properties, then this part is omitted in the script.

Domain declaration MoonLight, like STREL, supports different semantics for monitoring. A user can specify the desired one by indicating the specific `SemiringExpression`:

```
domain <SemiringExpression>
```

Currently, MoonLight supports qualitative (`boolean`) and quantitative (`minmax`) semantics of STREL.

Formula declaration Finally, the script contains the list of formulas that can be monitored:

```
formula <name> = <Formula>;
...
formula <name> = <Formula>;
```

A formula can be used within another formula. Furthermore, a formula can have parameters that are instantiated when monitoring is performed:

```
FormulaDeclaration::=
formula <name> (<VariableDeclaration>, ...,
    <VariableDeclaration>) = <Formula>;
```

The syntax of STREL `Formula` is reported in Fig. 2 and has been described in the background material.

Example 1
Figure 4 reports an example of a Moonlight script for the sensor network system. The script starts (lines 1–4) with the definition of the domains of input signals. In our scenario, these values are the type of node, the battery level, and the temperature level. As domains, the node type is represented by an integer (`int`), and the battery and temperature by real values (`real`).

The model for the space is described in lines 5–14. We can define a variable also on the location, which can be seen as a constant signal. In the example, we associate with each location the node type, which is represented by an integer (`int`). Spatial structures in STREL can change over time. This enables the modeling of the mobile network of sensors as in our example by updating edges and edge labels. The edges can have more than one label with different domains. In this

```
1  signal {
2      real battery;
3      real temperature;
4      int nodeType;
5  }
6  space {
7      edges {
8          int hop;
9          real dist;
10     }
11 }
12 domain boolean;
13 formula atom = (nodeType==3);
14 formula P1 = atom reach(hop)[0,1]{(nodeType==1)
15     |(nodeType==2)};
16 formula Ppar(int k) = atom reach(hop)[0, k]
17     (nodeType==1);
```

Fig. 4 The `sNetMonScript.mls` file, an example of Moonlight monitor script specification

case, we have two labels, `hop` having `int` domain and `dist` with type `real`. The user specifies the semantics in line 15. Lines 16–18 represent examples of STREL formulas. Line 16 defines an atomic proposition, which is also used in the next two formulas. Line 18 defines a parameterized formula by the parameter k.

6 Front-end

In this section, by two simple examples we will show how MoonLight can be integrated in MatLab and Python scripts.

6.1 Using MoonLight in MATLAB™

All the files and scripts needed to integrate MoonLight are available in the folder `distribution/matlab`. The installation script `install.m` contains the commands to save the right paths and should be run first. A detailed description of the installation process is available on the tool website. Folder `distribution/matlab/moonlight` contains the library `moonlight.jar` and the Matlab™ classes:

```
ScriptLoader.m
MoonlightScript.m
SpatialTemporalScriptComponent.m
TemporalScriptComponent.m
```

It contains all methods to use MoonLight in the Matlab™ environment.

`ScriptLoader.m` contains the methods to load the MoonLight `.mls` script. This can be loaded from file using the `loadFromFile(<filename>)` method, which takes as input the file name containing the script to load, or from a string using the `loadFromText(<stringArray>)` method. The method produces an object of the `MoonlightScript` class. After this operation is performed, a Java class is generated from the script and dynamically loaded. A reference to this object is returned to be used later.

Considering again our running example, we can load the Moonlight script of Fig. 4 with the code

```
myScript=
    ScriptLoader.loadFromFile("sNetMonScript");
```

`MoonlightScript` is a wrapper around the Java interface moonlight/MoonLightScript. It contains all the useful methods to get a monitor associated with formulas defined in the script. `MoonlightScript.isTemporal()` and `MoonlightScript.isSpatialTemporal()` return true if this is a temporal or spatio-temporal monitor, respectively; To change the domain of the script on the fly, we can use the methods `MoonlightScript.setBooleanDomain()` for the Boolean semantics and `MoonlightScript.setMinMaxDomain()` for the quantitative one. For example, we can specify in our script the monitoring semantics by writing

```
myScript.setMinMaxDomain();
```

`MoonlightScript.getMonitors()` returns the list of available formulas. In our example,

```
myScript.getMonitors();
```

returns the string list

```
"P1"
"Ppar"
```

`MoonlightScript.getMonitor(<formulaName>)` instantiates the monitor associated with the formula named `formulaName`. The method generates an object of the `TemporalScriptComponent` or of the `SpatialTemporalScriptComponent`, depending on whether the script is temporal or spatio-temporal. Considering again our running example, we can define the monitor of formula `"Ppar"` in the following way:

```
myMonitor = myScript.getMonitor("Ppar");
```

Once the monitor is generated, we can use it to verify the satisfaction of the property over a given temporal or spatio-temporal signal. We first describe the spatio-temporal case and later the temporal one.

`SpatialTemporalScriptComponent.m` is a wrapper around the Java interfaces moonlight/SpatialTemporalScriptComponent and contains the methods that can be used to monitor a specific spatial-temporal trajectory: `monitor` in case of a dynamical spatial model and `monitor_static` in case of a system with fixed spatial structure. These methods take four inputs:

- `<graph>` is an array of Matlab™ graph structures specifying the spatial structure at each point in time. In case of a static model, it is a single graph structure;
- `<time>` is an array of time points at which observations are provided;

- <values> is a map (a cell array) with a cell for each node. In each cell, there is an $n \times m$ matrix where each row represents the values of the signals at the time points specified by <time> (with n equal to the number of time points and m the number of the considered signals);
- <param> (optional) is an array of values used to instantiate the parameters of the formula. In case of a nonparametric formula, it is omitted.

The `monitor` method uses some private methods to convert the Matlab™ inputs <graph> and <values> in Java inputs: `toJavaGraphModel`, and `toJavaSignal`.

In our running example, we have

```
result =
    m.monitor(spatialModel,time,values,param);
```

where `spatialModel` is the array of Matlab graph structures where with each time step `i`, a `spatialModel{i} is associated. Edges` represents the adjacent list of the graph; `time` is the array of time points; `values` is a cell array, where each cell has a three-dimensional signal representing the node type, battery, and temperature. We represent different types of nodes using integer numbers 1, 2, and 3 to represent *coordinator*, *router*, and *end-device*, respectively. Finally, `param` is used to instantiate the parameter `k` of formula `Ppar`. The output `result` is similar to the input signal. It is a map that associates a Boolean signal (for the Boolean semantics) or a real-value signal (for the quantitative semantics) at each node, i.e., the Boolean or quantitative satisfaction at each time in each node.

In Fig. 5 (top), we can see the Boolean satisfaction at time zero of each node with respect to the formula `P1` of our script example in Fig. 4. The blue nodes (marked with a V) on the plot of Fig. 5 (top) correspond to the nodes that satisfy the property, i.e., the end devices that reach a router or a coordinator with at most one hop.

Figure 5 (bottom) shows the satisfaction of the formula

```
P4=(nodeType==3)reach(hop)[0,1]{
(nodeType==2)reach(hop)[0,5](nodeType==1)}
```

`P4` holds only in the nodes connected directly to the coordinator or to routers that can reach the coordinator through a maximum of four other routers. We can see that nodes 3, 4, 5, and 9 satisfy `P1` but not `P4`. The property

```
PT2 = globally P4
```

can be used to check that `P4` is true at each time step.

`TemporalScriptComponent.m` is wrapper around the Java interfaces moonlight/TemporalScriptComponent and contains the method `monitor(<time>, <values>, <parameters>)`, where:

- <time> is an array containing the trajectory time steps;
- <values> is an $n \times m$ matrix considering a trajectory with m signals and n time steps for each signal;

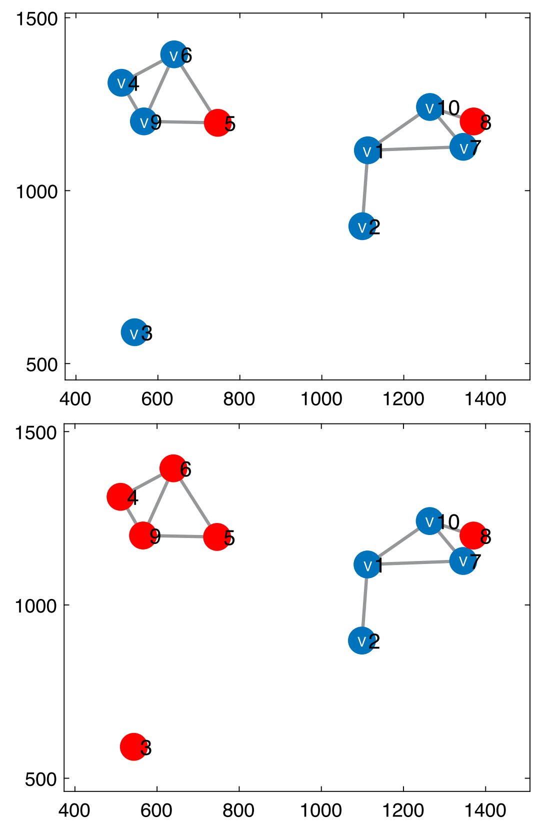


Fig. 5 (*top*) Boolean satisfaction of formula `P1` (blue nodes (V) satisfy the formula, red nodes do not satisfy the formula); (*bottom*) Boolean satisfaction of formula `P4` (blue nodes (V) satisfy the formula, red nodes do not satisfy the formula)

```
trajFunction = @(t)[sin(t);cos(t)];
time = 0:0.1:3.1;
values = trajFunction(time);
script = [
"signal { real x; real y}",...
"domain minmax;",...
"formula future = globally [0, 0.2]  (x >
    y);"...
"formula past = historically [0, 0.2]  (x > y);"
];
moonlightScript =
    ScriptLoader.loadFromText(script);
quantitativeMonitor =
    moonlightScript.getMonitor("past");
quantMonitorResult =
    quantitativeMonitor.monitor(time,values):

moonlightScript.setBooleanDomain();
booleanMonitor =
    moonlightScript.getMonitor("future");
boolMonitorResult =
    booleanMonitor.monitor(time,values);
```

Fig. 6 Example of monitoring a temporal trajectory in Matlab™

- <param>(optional) is an array containing the values of the formula parameters.

Example 2
Figure 6 describes a simple example of a temporal monitor

```
 1 from moonlight import *
 2 locationDb = ...
 3 timeGraph = [0.0]
 4 graph = locationDb.get_graph()
 5 timeSignalValues = locationDb.get_time()
 6 signalValues = locationDb.get_traces()
 7 script = """
 8 signal { int nBikes; }
 9 space {edges { real distance; }}
10 domain boolean;
11 formula service = globally[0,1200]{somewhere(distance) [0, 500] ( nBikes>=2 )};
12 """
13 moonlightScript = ScriptLoader.loadFromText(script)
14 booleanMonitor = moonlightScript.getMonitor("service")
15 booleanMonitorResult = monitor.monitor(timeGraph,graph,timeSignalValues,signalValues)
```

Fig. 7 Example of Python MoonLight script

loading the script as a string. We generate the trajectory simply as a two-signal `trajFunction = @(t)[sin(t); cos(t)]` of the sin and cos functions. We fix the time vector as the array `time = 0:0.1:3.1`, and then the input values of the monitor are the values of `trajFunction` for the `time` points. We define the script directly as a string (lines 4–9) and load it with the `loadFromText` method. The initialization of the monitor, considering the formula `past`, is done in line 11, and the monitoring in line 12. We then repeat the monitoring considering the Boolean semantics (line 14) and the formula `future` (line 15).

6.2 Using MoonLight in Python

The module `distribution/python` contains all files needed to use MoonLight in Python. In this case, we do not need an installation script. The framework requires the library `pyjnius`, which is a Python library for accessing to Java classes. The folder contains the library `moonlight.jar` and the Python file `moonlight.py`, which contains all classes and methods to interface Moonlight. Similarly to Matlab, we have the four classes `ScriptLoader`, `MoonlightScript`, `TemporalScriptComponent`, and `SpatialTemporalScriptComponent`.

The methods available in the Python interface are exactly the same available in the corresponding m-file of Matlab. Data are directly passed as inputs without needing a conversion.

Example 3
Figure 7 describes an example of a spatio-temporal monitor loading the script as a string. We get the spatio-temporal trajectory from a dataset containing the historical dock readings from sensors in Melbourne's Bike Share [20]. For each bike station, we know the geographical coordinates and the number of bikes (`nBikes`, line 8) available at specific times (`signalValues`, line 6), which are the same for each station.

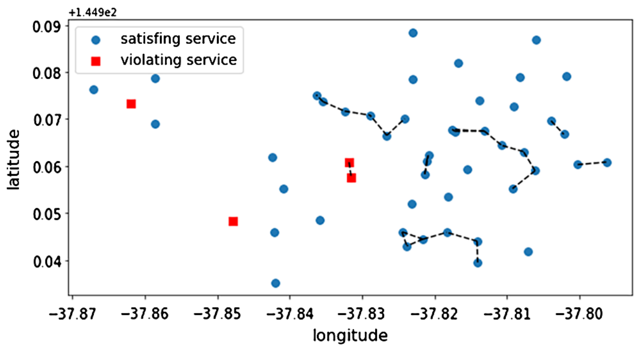


Fig. 8 Boolean satisfaction of formula `service`. Blue circular nodes satisfy the `service` formula, and red squared nodes do not satisfy the formula. Dashed lines connect locations with a mutual distance below 500 meters

The spatial model (`graph`, line 4) is a static fully connected graph representing the distance (in meters) from one location to another one. `locationDb` (line 2, implementation in `bike.ipynb`) is a wrapper around the dataset that we use to generate the spatial model (lines 3–4) and the temporal trajectories (lines 5–6). We define the script directly as a string (lines 7–12) and load it with the `loadFromText` method. The initialization of the monitor, considering formula `service`, is done in line 14, and the monitoring in line 15. The formula service is satisfied in a given location if there is another location within 500 meters with more than 2 bikes. Figure 8 shows the satisfaction of the formula.

7 Experimental evaluation

In this section, we evaluate the performance of the tool. First, we compare the performance of MoonLight with respect to S-Taliro [3] and Breach [21], the two most used tools to monitor STL formulas. Then we evaluate the scalability of the spatial operators by varying the size of the spatial model and the number of time steps.

Our experiments were performed on a workstation with an Intel Core i7-5820K (6 cores, 3.30 GHz) and 32 GB RAM,

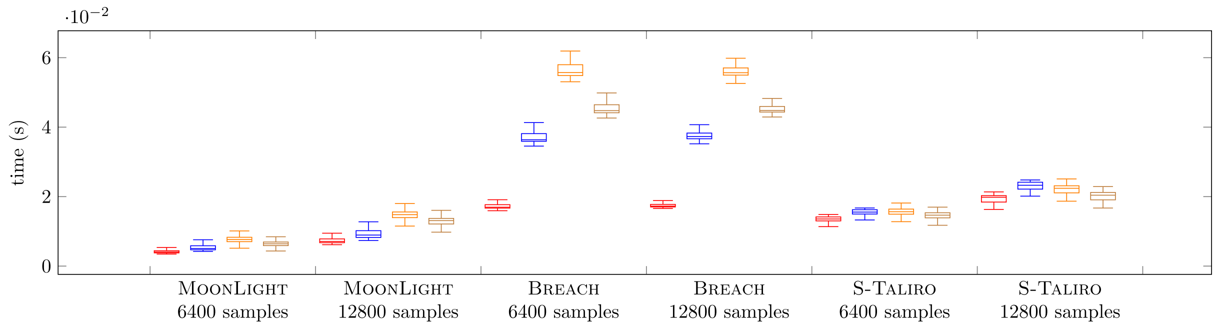


Fig. 9 The comparison of the computational time (in sec.) between MoonLight, Breach, and S-Taliro for simulation traces with different lengths. The different colors represent the result for different requirements (R1), (R2), (R3), and (R4), the color version of the article is given online

running Linux Ubuntu 16.04.6 LTS, Matlab™ R2020a and OpenJDK 64-Bit Server VM 1.8.0_252.

7.1 Temporal evaluation: monitoring signal temporal logic

We consider the *Automatic Transmission* example in [29]. This benchmark consists of a Matlab™/Simulink deterministic model of an automatic transmission controller. The model has two inputs (the throttle and break) and two outputs, the speed of the engine ω (RPM) and the speed of the vehicle v (mph). We monitor the robustness of four requirements in [29]:

(R1) The engine speed never reaches $\bar{\omega}$:

$$\texttt{globally}\ (\omega < \bar{\omega}).$$

(R2) The engine and vehicle speeds never reach $\bar{\omega}$ and $\bar{v}$, respectively:

$$\texttt{globally}\{(\omega < \bar{\omega})\ \&\ (v < \bar{v})\}.$$

(R3) If the engine speed is always less than $\bar{\omega}$, then the vehicle speed cannot exceed $\bar{v}$ in less than T seconds:

$$\texttt{!\{eventually[0,T]}\ (v > \bar{v})\ \&\ \texttt{globally}\ (\omega < \bar{\omega})\}.$$

(R4) Within T seconds, the vehicle speed is above $\bar{v}$, and from that point on the engine speed is always less than $\bar{\omega}$:

$$\texttt{eventually[0,T]\{}\ (v \geq \bar{v})\ \&\ \texttt{globally}\ (\omega < \bar{\omega})\ \}.$$

We randomly generated 20 different input traces with 6400 samples and other 20 with 12800 samples (0.01 sec. of sampling time). For each input trace, we simulated the model and monitored the robustness of the four requirements over the outputs by varying the parameters $\bar{v} \in \{120, 160, 170, 200\}$, $\bar{\omega} \in \{4500, 5000, 5200, 5500\}$, and $T \in \{4, 8, 10, 20\}$. For a fixed combination of parameters and output traces, we repeated the monitoring experiment 20 times and considered the mean of the execution times. In Fig. 9, we compare the performance of our MoonLight monitors with S-Taliro [3] and Breach [21] using boxplots, representing the quartiles of the execution times distribution for monitoring each requirement with each tool. The graph shows a good performance of MoonLight with respect to the other tools. However, it is important to note that Breach considers piecewise linear signals and computes the interpolation between two consecutive samples when necessary, whereas our tool and S-Taliro interpret the signal stepwise.

Table 1 List of STREL formulas used in the experimental evaluation

	STREL formula
P1	`(x<=0.5)reach(hop)[0, 30] (x>0.5)`
P2	`escape(hop)[5, inf] (x>0.5)`
P3	`somewhere(hop)[0, 3] (x>0.5)`
SPT1	`(x<=0.5)reach[0,30]eventually(x>0.5)`
TSP1	`globally((x<=0.5)reach[0,30](x>0.5))`
SPT2	`somewhere[0,30]eventually(x>0.5)`
TSP2	`eventually(somewhere[0,30](x>0.5))`

7.2 Spatio-temporal evaluation

We evaluate the scalability of the spatial operators considering a graph with N nodes and a signal consisting of K steps. We compute the Boolean (B) and quantitative (Q) semantics considering dynamic and static graphs (for spatio-temporal properties) for the STREL formulas reported in Table 1.

The first three formulas, which are named (P1), (P2), and (P3), contain only spatial properties, whereas the last four, named (SPT1), (TSP1), (SPT2), and (TSP2), are used to evaluate the interplay between temporal and spatial operators.

Tables 2 and 3 show the execution time of the monitoring procedure with different values of N and K.

First of all, we can observe that for property (P1), the one based on operator `reach`, the execution time does not change if a static or dynamic spatial model is considered. Moreover,

Table 2 The comparison of the computational time (in sec) with respect to N nodes and K time steps for formulas (P1), (P2), and (P3) described in Table 1, for Boolean and quantitative semantics. (S) indicates a static graph, otherwise, when it is not specified, it is a dynamic graph (edges' values can evolve in time)

	Boolean			Quantitative		
N \ K	1	10	100	1	10	100
	P1					
16	0.011	0.014	0.037	0.006	0.024	0.13
16 (S)	0.011	0.015	0.0370	0.005	0.024	0.124
256	0.009	0.037	0.239	0.030	0.140	1.139
256 (S)	0.009	0.036	0.251	0.029	0.13	1.113
1024	0.028	0.125	0.987	0.124	0.724	7.344
1024 (S)	0.022	0.136	0.990	0.123	0.704	6.706
	P2					
16	0.008	0.009	0.051	0.003	0.008	0.056
16 (S)	0.008	0.004	0.014	0.003	0.007	0.053
256	0.093	0.276	2.244	0.197	0.861	7.649
256 (S)	0.083	0.077	0.293	0.236	0.701	5.990
1024	1.211	5.422	47.698	4.266	23.277	215.398
1024 (S)	0.500	0.711	1.826	3.901	18.167	159.794
	P3					
16	0.007	0.007	0.042	0.001	0.002	0.013
16 (S)	0.005	0.002	0.009	0.001	0.001	0.007
256	0.094	0.264	1.916	0.044	0.212	1.892
256 (S)	0.088	0.070	0.404	0.031	0.072	0.533
1024	1.128	6.410	60.706	1.145	6.674	58.82
1024 (S)	0.655	1.118	5.011	0.804	1.799	4.620

Table 3 The comparison of the computational time (in sec) with respect to N nodes and K time steps for formulas (SPT1), (TSP2), (SPT2), and (TSP2) described in Table 1 for Boolean and quantitative semantics. (S) indicates a static graph; otherwise, when not specified, it is a dynamic graph (the values of edges can evolve in time)

	Boolean			Quantitative		
N \ K	1	10	100	1	10	100
			SPT1			
16	0.012	0.015	0.039	0.005	0.015	0.085
16 (S)	0.012	0.014	0.039	0.006	0.021	0.124
256	0.009	0.038	0.244	0.027	0.134	1.027
256 (S)	0.010	0.042	0.243	0.027	0.119	1.042
1024	0.022	0.134	0.997	0.124	0.691	6.276
1024 (S)	0.030	0.153	0.989	0.131	0.691	6.557
			TSP1			
16	0.014	0.015	0.037	0.006	0.024	0.093
16 (S)	0.013	0.016	0.042	0.005	0.016	0.091
256	0.008	0.041	0.24	0.029	0.124	1.13
256 (S)	0.008	0.040	0.242	0.029	0.129	1.101
1024 (S)	0.028	0.128	0.988	0.122	0.722	6.713
1024	0.03	0.144	0.997	0.126	0.731	6.927
			SPT2			
16	0.001	0.001	0.042	0.001	0.001	0.008
16 (S)	0.001	0.002	0.003	0.002	0.002	0.003
256	0.102	0.255	1.928	0.042	0.198	1.664
256 (S)	0.070	0.073	0.075	0.032	0.078	0.572
1024	1.117	6.242	54.588	1.19	5.824	62.253
1024 (S)	0.672	1.610	1.786	0.862	1.174	5.289
			TSP2			
16	0.001	0.007	0.041	0.001	0.008	0.043
16 (S)	0.001	0.002	0.007	0.001	0.002	0.008
256	0.106	0.255	1.904	0.044	0.22	1.810
256 (S)	0.066	0.089	0.243	0.033	0.068	0.509
1024	1.175	6.428	60.583	1.272	6.803	61.952
1024 (S)	0.553	1.224	4.476	0.826	2.086	5.121

the monitoring is faster when we consider the *Boolean* domain. This is because when a `reach` monitor is computed, at each time step the monitored values are propagated on the graph to compute a fixpoint. This means that at each time step the spatial model is reconsidered. Moreover, the Boolean monitoring terminates first due to the fact that, with this domain, in general, a fixpoint is reached with a limited number of iterations.

Computational times for property (P2) are different. This property involves operator `escape`, and its monitor consists of flooding of values over the spatial model and of the computation of the matrix of minimum distances. For this reason, the execution time of monitors for (P2) changes significantly if we consider static or dynamic models. This difference is amplified when the monitoring of property (P3) is considered. In this case (where operator `everywhere` is used), monitoring mainly consists of the computation of the distance matrix, which is computed only one time in a static model, whereas it is computed at each time step when a dynamic spatial model is considered. We can also observe that for property (P3), differently from (P1) and (P2), there is no difference between Boolean and quantitative monitoring, which is because for the `somewhere` and `everywhere` operators, we do not use a flooding algorithm, but we just need to select the nodes that satisfy the distance constraints.

Finally, the values reported in Table 3 show that the interplay between spatial and temporal properties does not affect the monitoring time. Indeed, the times needed to monitor properties (STP1) and (PST1), which use `reach` with `eventually`, have similar orders of magnitude to those reported in Table 2 for property (P1). Similarly, the time needed to monitor (STP2) and (PST2) is similar to the time needed to monitor (P3). Note that, as we have already observed for

Table A1 Monitoring function

$\mathbf{m}(\mathcal{S},\vec{x},\mu,t,\ell)=\iota(\mu,\vec{x}(\ell,t))$
$\mathbf{m}(\mathcal{S},\vec{x},\neg\varphi,t,\ell)=\neg\mathbf{m}(\mathcal{S},\vec{x},\varphi,t,\ell)$
$\mathbf{m}(\mathcal{S},\vec{x},\varphi_1\wedge\varphi_2,t,\ell)=\mathbf{m}(\mathcal{S},\vec{x},\varphi_1,t,\ell)\wedge\mathbf{m}(\mathcal{S},\vec{x},\varphi_2,t,\ell)$
$\mathbf{m}(\mathcal{S},\vec{x},\varphi_1\ \mathrm{U}_{[t_1,t_2]}\ \varphi_2,t,\ell)=\bigvee_{t'\in t+[t_1,t_2]}(\mathbf{m}(\mathcal{S},\vec{x},\varphi_2,t',\ell)\wedge\bigwedge_{t''\in[t,t']}\mathbf{m}(\mathcal{S},\vec{x},\varphi_1,t'',\ell))$
$\mathbf{m}(\mathcal{S},\vec{x},\varphi_1\ \mathrm{S}_{[t_1,t_2]}\ \varphi_2,t,\ell)=\bigvee_{t'\in t-[-t_2,-t_1]}\big(\mathbf{m}(\mathcal{S},\vec{x},\varphi_2,t',\ell)\wedge\bigwedge_{t''\in[t',t]}\mathbf{m}(\mathcal{S},\vec{x},\varphi_1,t'',\ell)\big)$
$\mathbf{m}(\mathcal{S},\vec{x},\varphi_1\ \mathcal{R}^{f:A\to A}_{[d_1,d_2]}\ \varphi_2,t,\ell)=\bigvee_{\tau\in Routes(\mathcal{S}(t),\ell)}\bigvee_{i:\left(d^f_\tau[i]\in[d_1,d_2]\right)}(\mathbf{m}(\mathcal{S},\vec{x},\varphi_2,t,\tau[i])\wedge\bigwedge_{j<i}\mathbf{m}(\mathcal{S},\vec{x},\varphi_1,t,\tau[j])$
$\mathbf{m}(\mathcal{S},\vec{x},\mathcal{E}^{f:A\to A}_{[d_1,d_2]}\ \varphi,t,\ell)=\bigvee_{\tau\in Routes(\mathcal{S}(t),\ell)}\bigvee_{\ell'\in\tau:\left(d^f_{\mathcal{S}(t)}[\ell,\ell']\in[d_1,d_2]\right)}\ \bigwedge_{i\leq\tau(\ell')}\mathbf{m}(\mathcal{S},\vec{x},\varphi,t,\tau[i]))$

(P3), monitoring of (STP2) and (PST2) is less expensive when a static spatial model is considered.

8 Conclusions

MoonLight provides a lightweight and very flexible monitoring tool for temporal and spatio-temporal properties of mobile and spatially arranged CPS. The possibility to use dedicated Matlab™ and Python interfaces enables us to easily integrate MoonLight as a component in other toolchains implementing more sophisticated computer-aided verification and synthesis techniques such as falsification analysis and parameter synthesis. The reader can find many other interesting examples in the *MoonLight GitHub Repository*. In the near future, we plan to extend the tool with new functionalities, such as supporting parallelization, to speed up the computation and online monitoring. Furthermore, we are considering other operators to increase the expressivity of the logic.

Appendix: STREL semantics

The semantics of STREL is reported in Table A1. We report the Boolean semantics; the quantitative semantics can be derived by substituting $\vee$, $\wedge$ with min, max; see [45] for more detail. The semantics is evaluated pointwise at each time and at each location, and it is defined in a recursive way. Given a formula ϕ, the function $\mathbf{m}(\mathcal{S},\vec{x},\phi,t,\ell)$ corresponds to the evaluation of the formula at time t in the location ℓ of the trajectory $\vec{x}$ and spatial model $\mathcal{S}$.

The function $\iota : AP\times D_1^n\to\mathbb{B}$ is the *signal interpretation function* and translates the input trace in a Boolean signal for each atomic proposition in AP.

The negation operator is interpreted with the negation function $\neg$ such that $\neg$ `true` = `false`. The conjunction operator $\varphi_1\wedge\varphi_2$ is interpreted as usual: $\mathbf{m}(\mathcal{S},\vec{x},\varphi_1\wedge\varphi_2,t,\ell)=$ `true` iff both $\mathbf{m}(\mathcal{S},\vec{x},\varphi_1 t,\ell)$ and $\mathbf{m}(\mathcal{S},\vec{x},\varphi_2 t,\ell)$ are equal to `true`.

The temporal operators work as usual: $(\mathcal{S}(t),\vec{x}(\ell,t))$ satisfies $\varphi_1\ \mathrm{U}_{[t_1,t_2]}\ \varphi_2$ iff it satisfies φ_1 from t until, in a time between t_1 and t_2 time units in the future, φ_2 becomes `true`, and $(\mathcal{S}(t),\vec{x}(\ell,t))$ satisfies $\varphi_1\ \mathrm{S}_{[t_1,t_2]}\ \varphi_2$ iff it satisfies φ_1 from now since, in a time between t_1 and t_2 time units in the past, φ_2 was `true`.

For the spatial operators, we denote by f the distance function (e.g., `int hop` or `real dist`), $d^f_\tau[i]$ is the distance over the route τ up to index i, and $d^f_{\mathcal{S}(t)}[\ell,\ell']$ is the minimum distance between locations ℓ and ℓ' with respect to the spatial model $\mathcal{S}(t)$. Then we have that $(\mathcal{S}(t),\vec{x}(\ell,t))$ satisfies $\varphi_1\ \mathcal{R}_{[d_1,d_2]}\ \varphi_2$ if and only if it satisfies φ_2 in a location ℓ' reachable from ℓ through a route τ, with length $d^f_\tau[\ell']$ belonging to the interval $[d_1,d_2]$, and such that $\tau[0]=\ell$ and all its elements with index less than $\tau(\ell')$ satisfy φ_1; $(\mathcal{S}(t),\vec{x}(\ell,t))$ satisfies $\mathcal{E}^f_{[d_1,d_2]}\ \varphi$ if and only if there exist a route τ, a location $\ell'\in\tau$, and an index $k\in\mathbb{Z}$ such that $\tau[0]=\ell$, $\tau[k]=\ell'$, and $d_{\mathcal{S}}[\ell,\ell']$ belongs to the interval $[d_1,d_2]$, whereas ℓ' and all the elements $\tau[0],\dots,\tau[k-1]$ satisfy φ.

Funding Open access funding provided by Università degli Studi di Trieste within the CRUI-CARE Agreement. This research has been partially supported by the Austrian FWF projects ZK-35 and W1255-N23, by the Italian PRIN project “SEDUCE” n. 2017TWRCNB and “IT-MaTTerS” n. 2017FTXR7S, and by the PNRR iNEST – Missione 4 Componente 2, Investimento 1.5 – D.D. 1058 23/06/2022, ECS_00000043.

References

1. Abbas, H., Fainekos, G.E., Sankaranarayanan, S., et al.: Probabilistic temporal logic falsification of cyber-physical systems. ACM Trans. Embed. Comput. Syst. **12**(s2), 95:1–95:30 (2013). https://doi.org/10.1145/2465787.2465797
2. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., et al.: A survey on sensor networks. IEEE Commun. Mag. **40**(8), 102–114 (2002). https://doi.org/10.1109/MCOM.2002.1024422
3. Annpureddy, Y., Liu, C., Fainekos, G.E., et al.: S-TaLiRo: a tool for temporal logic falsification for hybrid systems. In: Proc. of TACAS 2011: The 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 254–257 (2011). https://doi.org/10.1007/978-3-642-19835-9_21
4. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. J. ACM **49**(2), 172–206 (2002). https://doi.org/10.1145/506147.506151
5. Bartocci, E., et al.: Teaching cardiac electrophysiology modeling to undergraduate students: laboratory exercises and GPU programming for the study of arrhythmias and spiral wave dynamics. Adv. Physiol. Educ. **35**(4), 427–437 (2011). https://doi.org/10.1152/advan.00034.2011
6. Bartocci, E., Bortolussi, L., Milios, D., et al.: Studying emergent behaviours in morphogenesis using signal spatio-temporal logic. In: Proc. of HSB 2015. LNCS, vol. 9271, pp. 156–172. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-26916-0_9
7. Bartocci, E., Bortolussi, L., Nenzi, L., et al.: System design of stochastic models using robustness of temporal properties. Theor. Comput. Sci. **587**, 3–25 (2015). https://doi.org/10.1016/j.tcs.2015.02.046
8. Bartocci, E., Bortolussi, L., Loreti, M., et al.: Monitoring mobile and spatially distributed cyber-physical systems. In: Proc. of MEMOCODE 2017: The 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, pp. 146–155. ACM, New York (2017). https://doi.org/10.1145/3127041.3127050
9. Bartocci, E., Bloem, R., Nickovic, D., et al.: A counting semantics for monitoring LTL specifications over finite traces. In: Proc. of CAV 2018: The 30th International Conference on Computer Aided Verification. LNCS, vol. 10981, pp. 547–564. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-96145-3
10. Bartocci, E., Deshmukh, J., Donzé, A., et al.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Lectures on Runtime Verification. LNCS, vol. 10457, pp. 135–175. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-75632-5_5
11. Bartocci, E., Falcone, Y., Francalanza, A., et al.: Introduction to runtime verification. In: Lectures on Runtime Verification – Introductory and Advanced Topics. LNCS, vol. 10457, pp. 1–33. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-75632-5
12. Bartocci, E., Ferrère, T., Manjunath, N., et al.: Localizing faults in simulink/stateflow models with STL. In: Prandini, M., Deshmukh, J.V. (eds.) Proc. of HSCC 2018 the 21st International Conference on Hybrid Systems: Computation and Control, pp. 197–206. ACM, New York (2018). https://doi.org/10.1145/3178126.3178131
13. Bartocci, E., Gol, E.A., Haghighi, I., et al.: A formal methods approach to pattern recognition and synthesis in reaction diffusion networks. IEEE Trans. Control Netw. Syst. **5**(1), 308–320 (2018). https://doi.org/10.1109/TCNS.2016.2609138
14. Bartocci, E., Manjunath, N., Mariani, L., et al.: Automatic failure explanation in CPS models. In: Ölveczky, P.C., Salaün, G. (eds.) Proc. of SEFM 2019: The 17th International Conference on Software Engineering and Formal Methods. LNCS, vol. 11724, pp. 69–86. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-30446-1_4
15. Bartocci, E., Bortolussi, L., Loreti, M., et al.: Moonlight: a lightweight tool for monitoring spatio-temporal properties. In: Deshmukh, J., Nickovic, D. (eds.) Proc. of RV 2020: The 20th International Conference on Runtime Verification. LNCS, vol. 12399, pp. 417–428. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-60508-7_23
16. Bartocci, E., Manjunath, N., Mariani, L., et al.: CPSDebug: a tool for explanation of failures in cyber-physical systems. In: Khurshid, S., Pasareanu, C.S. (eds.) Proc. of ISSTA '20: The 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 569–572. ACM, New York (2020). https://doi.org/10.1145/3395363.3404369
17. Baumeister, J., Finkbeiner, B., Schwenger, M., et al.: FPGA stream-monitoring of real-time properties. ACM Trans. Embed. Comput. Syst. **18**(5s), 88:1–88:24 (2019). https://doi.org/10.1145/3358220
18. Belmonte, G., Ciancia, V., Latella, D., et al.: Voxlogica: a spatial model checker for declarative image analysis. In: Proc. of TACAS 2019: The 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 11427, pp. 281–298. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-17462-0_16
19. Bortolussi, L., Milios, D., Sanguinetti, G.: U-Check: model checking and parameter synthesis under uncertainty. In: Proc. of QEST 2015: 12th Inter. Conf. on Quantitative Evaluation of Systems. LNCS, vol. 9259, pp. 89–104. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-22264-6_6
20. City of Melbourne: Melbourne Bike Share Station Readings 2011-2017 [Dataset]. https://www.opendatanetwork.com/dataset/data.melbourne.vic.gov.au/74id-aqj9 (2018)
21. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Proc. of CAV 2010: The 22nd International Conference on Computer Aided Verification. LNCS, vol. 6174, pp. 167–170. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-14295-6
22. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Proc. of FORMATS, pp. 92–106. Springer, Berlin (2010)
23. Donzé, A., Clermont, G., Legay, A., et al.: Parameter synthesis in nonlinear dynamical systems: application to systems biology. In: Proc. of RECOMB 2009: The 13th Annual International Conference on Research in Computational Molecular Biology. LNCS, vol. 5541, pp. 155–169. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-02008-7_11
24. Donzé, A., Krogh, B., Rajhans, A.: Parameter synthesis for hybrid systems with an application to simulink models. In: Proc. of HSCC 2009: The 12th International Conference on Hybrid Systems: Computation and Control. LNCS, vol. 5469, pp. 165–179. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-00602-9_12
25. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Proc. of CAV 2013: The 25th International Conference on Computer Aided Verification. LNCS, vol. 8044, pp. 264–279. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-39799-8_19
26. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theor. Comput. Sci. **410**(42), 4262–4291 (2009). https://doi.org/10.1016/j.tcs.2009.06.021
27. Grosu, R., Smolka, S.A., Corradini, F., et al.: Learning and detecting emergent behavior in networks of cardiac myocytes. Commun. ACM **52**(3), 97–105 (2009). https://doi.org/10.1145/1467247.1467271
28. Haghighi, I., Jones, A., Kong, Z., et al.: SpaTeL: a novel spatial-temporal logic and its applications to networked systems. In: Proc. of HSCC'15: The 18th International Conference on Hybrid Systems: Computation and Control, pp. 189–198. IEEE, New York (2015). https://doi.org/10.1145/2728606.2728633

29. Hoxha, B., Abbas, H., Fainekos, G.E.: Benchmarks for temporal logic requirements for automotive systems. In: Proc. of ARCH@CPSWeek 2014: The 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems. EPiC Series in Computing, vol. 34, pp. 25–30. EasyChair (2015). https://doi.org/10.29007/xwrs
30. Jaksic, S., Bartocci, E., Grosu, R., et al.: Quantitative monitoring of STL with edit distance. Form. Methods Syst. Des. **53**(1), 83–112 (2018). https://doi.org/10.1007/s10703-018-0319-x
31. Jaksic, S., Bartocci, E., Grosu, R., et al.: An algebraic framework for runtime verification. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **37**(11), 2233–2243 (2018). https://doi.org/10.1109/TCAD.2018.2858460
32. Kane, A., Pirotta, E., Wischnewski, S., et al.: Spatio-temporal patterns of foraging behaviour in a wide-ranging seabird reveal the role of primary productivity in locating prey. Mar. Ecol. Prog. Ser. **646**, 175–188 (2020). https://doi.org/10.3354/meps13386
33. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (1990). https://doi.org/10.1007/BF01995674
34. Lemire, D.: Streaming maximum-minimum filter using no more than three comparisons per element. Nord. J. Comput. **13**(4), 328–339 (2006)
35. Leucker, M., Sánchez, C., Scheffel, T., et al.: Tessla: runtime verification of non-synchronized real-time streams. In: Proc. of SAC 2018: The 33rd Annual ACM Symposium on Applied Computing, pp. 1925–1933. ACM, New York (2018). https://doi.org/10.1145/3167132.3167338
36. Loreti, M., Hillston, J.: Modelling and analysis of collective adaptive systems with CARMA and its tools. In: Proc. of SFM 2016: Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems – 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems. LNCS, vol. 9700, pp. 83–119. Springer, Berlin (2016). https://doi.org/10.1007/978-3-319-34096-8
37. Lukina, A., Esterle, L., Hirsch, C., et al.: ARES: adaptive receding-horizon synthesis of optimal plans. In: Proc. of TACAS 2017: The 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 286–302 (2017). https://doi.org/10.1007/978-3-662-54580-5_17
38. Ma, M., Bartocci, E., Lifland, E., et al.: SaSTL: spatial aggregation signal temporal logic for runtime monitoring in smart cities. In: 11th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2020, Sydney, Australia, April 21–25, 2020, pp. 51–62. IEEE, New York (2020). https://doi.org/10.1109/ICCPS48487.2020.00013
39. Maler, O., Ničković, D.: Monitoring temporal properties of continuous signals. In: Proc. of FORMATS/FTRTFT. Lecture Notes in Computer Science, vol. 3253, pp. 152–166. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-30206-3_12
40. Maler, O., Ničković, D.: Monitoring properties of analog and mixed-signal circuits. Int. J. Softw. Tools Technol. Transf. **15**(3), 247–268 (2013). https://doi.org/10.1007/s10009-012-0247-9
41. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. Form. Methods Syst. Des. **51**(1), 31–61 (2017). https://doi.org/10.1007/s10703-017-0275-x
42. Nenzi, L., Bortolussi, L., Loreti, M.: jSSTL – a tool to monitor spatio-temporal properties. In: Proc. of VALUETOOLS 2016: The 10th EAI International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2016. ACM, New York (2016). https://doi.org/10.4108/eai.25-10-2016.2266978
43. Nenzi, L., Bortolussi, L., Ciancia, V., et al.: Qualitative and quantitative monitoring of spatio-temporal properties with SSTL. Log. Methods Comput. Sci. **14**(4), 1–38 (2018). https://doi.org/10.23638/LMCS-14(4:2)2018
44. Nenzi, L., Bartocci, E., Bortolussi, L., et al.: Monitoring spatio-temporal properties (invited tutorial). In: Proc. of RV 2020: The 20th International Conference on Runtime Verification. LNCS, vol. 12399, pp. 21–46. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-60508-7_2
45. Nenzi, L., Bartocci, E., Bortolussi, L., et al.: A logic for monitoring dynamic networks of spatially-distributed cyber-physical systems. Log. Methods Comput. Sci. **18**(1), 4:1–4:30 (2022). https://lmcs.episciences.org/8936. https://doi.org/10.46298/lmcs-18(1:4)2022
46. Nickovic, D., Yamaguchi, T.: RTAMT: online robustness monitors from STL. In: Proc. of ATVA 2020: The 18th International Symposium on Automated Technology for Verification and Analysis – 18th International Symposium. LNCS, vol. 12302, pp. 564–571. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-59152-6
47. Nickovic, D., Lebeltel, O., Maler, O., et al.: AMT 2.0: qualitative and quantitative trace analysis with extended signal temporal logic. In: Proc. of TACAS 2018: The 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 10806, pp. 303–319. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-89963-3
48. Parr, T.: The Definitive ANTLR 4 Reference, 2nd edn. Pragmatic Bookshelf, Raleigh (2013)
49. Ratasich, D., Khalid, F., Geissler, F., et al.: A roadmap towards resilient Internet of things for cyber-physical systems. IEEE Access **7**, 13260–13283 (2019). https://doi.org/10.1109/ACCESS.2019.2891969
50. Rodionova, A., Bartocci, E., Ničković, D., et al.: Temporal logic as filtering. In: Proc. of HSCC 2016, pp. 11–20. ACM, New York (2016). https://doi.org/10.1145/2883817.2883839
51. Sankaranarayanan, S., Kumar, S.A., Cameron, F., et al.: Model-based falsification of an artificial pancreas control system. SIGBED Rev. **14**(2), 24–33 (2017). https://doi.org/10.1145/3076125.3076128
52. Silvetti, S., Policriti, A., Bortolussi, L.: An active learning approach to the falsification of black box cyber-physical systems. In: Proc. of IFM 2017: The 13th International Conference on Integrated Formal Methods. LNCS, vol. 10510, pp. 3–17. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-66845-1
53. Ulus, D.: Montre: a tool for monitoring timed regular expressions. In: Proc. of CAV 2017: The 29th International Conference on Computer Aided Verification. LNCS, vol. 10426, pp. 329–335. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-63387-9
54. Yaghoubi, S., Fainekos, G.: Hybrid approximate gradient and stochastic descent for falsification of nonlinear systems. In: Proc. ACC 2017: The 2017 American Control Conference, pp. 529–534. IEEE, New York (2017). https://doi.org/10.23919/ACC.2017.7963007