Online and Offline Stream Runtime Verification of Synchronous Systems *

César Sánchez

IMDEA Software Institute, Spain
cesar.sanchez@imdea.org

Abstract. We revisit Stream Runtime Verification for synchronous systems. Stream Runtime Verification (SRV) is a declarative formalism to express monitors using streams, which aims to be a simple and expressive specification language. The goal of SRV is to allow engineers to describe both correctness/failure assertions and interesting statistical measures for system profiling and coverage analysis. The monitors generated are useful for testing, under actual deployment, and to analyze logs.

The main observation that enables SRV is that the steps in the algorithms to monitor temporal logics (which generate Boolean verdicts) can be generalized to compute statistics of the trace if a different data domain is used. Hence, the fundamental idea of SRV is to separate the temporal dependencies in the monitoring algorithm from the concrete operations to be performed at each step.

In this paper we revisit the pioneer SRV specification language Lola and present in detail the online and offline monitoring algorithms. The algorithm for online monitoring Lola specifications uses a partial evaluation strategy, by incrementally constructing output streams from input streams, maintaining a storage of partially evaluated expressions. We identify syntactically a class of specifications for which the online algorithm is trace length independent, that is, the memory requirement does not depend on the length of the input streams. Then, we extend the principles of the online algorithm to create an efficient offline monitoring algorithm for large traces, which consist on scheduling trace length independent passes on a dumped log.

Keywords: Runtime verification, formal verification, formal methods, stream runtime verification synchronous systems, dynamic analysis, monitoring.

This is a post-peer-review, pre-copyedit version of an article published at LNCS vol 11237, Springer (2018).The final authenticated version is available online at: https://doi.org/10.1007/978-3-030-03769-7_9.

* This research has been partially supported by: the EU H2020 project Elastest (num. 731535), by the Spanish MINECO Project "RISCO (TIN2015-71819-P)" and by the EU ICT COST Action IC1402 ARVI (*Runtime Verification beyond Monitoring*).

1 Introduction

Runtime Verification (RV) is an applied formal method for software reliability that analyzes the system by processing one trace at a time. In RV a specification is transformed automatically into a monitor, and algorithms are presented to evaluate monitors against traces of observations from the system. There are two kinds of monitoring algorithms in RV depending on when the trace is generated and processed. In online monitoring the monitor checks the trace while the system runs, while in offline monitoring a finite collection of previously generated traces are analyzed. Online monitoring is used to detect violations of the specification when the system is in operation, while offline monitoring is used in post-mortem analysis and for testing large systems before deployment.

Static verification techniques like model checking intend to show that every (infinite) run of a system satisfies a given specification, while runtime verification is concerned only with a single (finite) trace. Thus, RV sacrifices completeness to provide an applicable formal extension of testing. See [24, 21] for modern surveys on runtime verification and the recent book [4].

The first specification languages studied for runtime verification were based on temporal logics, typically LTL [22, 13, 6], regular expressions [28], timed regular expressions [1], rules [3], or rewriting [27]. In this paper we revisit the Stream Runtime Verification specification formalism, in particular the Lola specification language for synchronous systems [12]. The Lola language can express properties involving both the past and the future and their arbitrary combination. In SRV, specifications declare explicitly the dependencies between input streams of values—that represent the observations from the system—and output streams of values—that represent monitoring outputs, like error reports and diagnosis information. The fundamental idea of SRV is to cleanly separate the temporal reasoning from the individual operations to be performed at each step. The temporal aspects are handled in a small number of constructs to express the offsets between observations and their uses. For the data, SRV uses off-the-self domains with interpreted functions so function symbols can be used as constructors to create expressions, and their interpretation is used for evaluation during the monitoring process. The domains used for SRV are not restricted to Booleans and allow richer domains like Integers, Reals (for computing quantitative verdicts) and even queues, stacks, etc. These domains do not involve any reasoning about time. The resulting expressiveness of SRV surpasses that of temporal logics and many other existing formalisms including finite-state automata. The restriction of SRV to the domain of Booleans is studied in [10], including the expressivity, the comparison with logics and automata and the complexity of the decision problems.

The online monitoring problem of past specifications can be solved efficiently using constant space and linear time in the trace size. For future properties, on the other hand, the space requirement depends on the length of the trace for rich types (even though for LTL, that is for the verdict domain of the Booleans, one can use automata techniques to reduce the necessary space to exponential in the size of the specification). Consequently, online monitoring of future temporal

formulas quickly becomes intractable in practical applications with long traces. On the other hand, the offline monitoring problem for LTL-like logics is known to be easy for purely past or purely future properties. We detail in the paper a syntactic characterization of *efficiently monitorable* specifications (introduced in [12]), for which the space requirement of the online monitoring algorithm is independent of the size of the trace, and linear in the specification size. This property was later popularized as *trace length independence* [5] and is a very desirable property as it allows online monitors to scale to arbitrarily large traces. In practice, most properties of interest in online monitoring can be expressed as efficiently monitorable properties. For the offline monitoring problem, we show an efficient monitoring strategy in the presence of arbitrary past and future combinations by scheduling trace length independent passes. We describe here the algorithm and results using the Lola specification language. An execution of the monitor extracted from a Lola specification computes data values at each position by evaluating the expressions over streams of input, incrementally computing the output streams.

Two typical specifications are properties that specify correct behavior, and statistical measures that allow profiling the system that produces the input streams. One important limitation of runtime verification is that liveness properties can never be violated on a finite trace. Hence, most of these properties have been typically considered as non-monitorable (for violation) as every finite prefix can be extended to a satisfying trace, at least if the system is considered as a black box and can potentially generate any suffix. An appealing solution that SRV supports is to compute quantitative measures from the observed trace. For example, apart from "there are only finitely many retransmissions of each package," which is vacuously true over finite traces, SRV allows to specify "what is the average number of retransmissions." Following this trend, runtime monitors can be used not only for bug-finding, but also for profiling, coverage, vacuity and numerous other analyses. An early approach for combining proving properties with data collection, which inspired SRV, appeared in [16].

In the present paper we present a simplified semantics of Lola [12] together with a detailed presentation of the monitoring algorithms as well as the necessary definitions and proofs. In the rest of the paper we use SRV and Lola interchangeably.

Related Work. The expressions that declare the dependencies between input streams and output streams in SRV are functional, which resemble synchronous languagues—which are also functional reactive stream computation languages—like Lustre [20], Esterel [9] and Signal [17], with additional features that are relevant to monitoring. The main difference is that synchronous languages are designed to express behaviors and therefore assume the causality assumption and forbid future references, while in SRV future references are allowed to describe dependencies on future observations. This requires additional expressiveness in the language and the evaluation strategies to represent that the monitor cannot decide a verdict without observing future values. These additional verdicts were

also introduced for this purpose in LTL-based logics, like LTL_3 and LTL_4 [7, 6, 8], to encode that the monitor is indecisive.

An efficient method for the online evaluation of *past* LTL properties is presented in [22], which exploits that past LTL can be recursively defined using only values in the previous state of the computation. The *efficiently monitorable* fragment of SRV specifications generalize this idea, and apply it uniformly to both verification and data collection. One of the early systems that most closely resembles Lola is Eagle [3], which allows the description of monitors using greatest and least fixed points of recursive definitions. Lola differs from Eagle in the descriptive nature of the language, and in that Lola is not restricted to checking logical formulas, but can also express numerical queries.

The initial application domain of Lola was the testing of synchronous hardware by generating traces of circuits and evaluating monitors against these traces. Temporal testers [26] were later proposed as a monitoring technique for LTL based on Boolean streams. Copilot [25] is a domain-specific language that, similar to Lola, declares dependencies between streams in a Haskell-based style, to generate C monitors that operate in constant time and space (the fragment of specifications that Copilot can describe is efficiently monitorable). See also [18].

The simple version of Lola presented here does not allow to quantify over objects and instantiate monitors to follow the actual objects observed, like in Quantified Event Automata [2]. Lola2.0 [14] is an extension of Lola that allows to express parametrized streams and dynamically generates monitors that instantiate these streams for the observed data items. The intended application of Lola2.0 is network monitoring.

Stream runtime verification has a also been extended recently to asynchronous and real-time systems. RTLola [15] extends SRV from the synchronous domain to timed streams. In RTLola streams are computed at predefined periodic instants of time, collecting aggregations between these predefined instants using a library of building blocks. TeSSLa [11] also offers a small collection of primitives for expressing stream dependencies (see also [23]) but allows to compute timed-streams at arbitrary real-time instants. The intended application of TeSSLa is hardware based monitoring. Striver [19] offers a Lola-like language with time offsets, that allows to express explicit instants of time in the expressions between streams. Striver is aimed at testing and monitoring of cloud based systems.

The rest of the paper is structured as follows. Section 2 revisits the syntax and semantics of SRV. Section 3 presents the online monitoring of SRV specifications, including the notion of efficient monitorability. Section 4 presents the algorithm for offline monitoring, and finally Section 5 concludes.

2 Overview of Stream Runtime Verification

In this section we describe SRV using the Lola specification language. The monitoring algorithms will be presented in Sections 3 and 4.

2.1 Specification Language: Syntax

We use many-sorted first order interpreted theories to describe data domains. A theory is given by a finite collection T of types and a finite collection F of function symbols. Since our theories are interpreted every type T is associated with a domain D of values and every symbol f is associated with a computable function that, given elements of the domains of the arguments compute a value of the domain of the resulting type. We use sort and type interchangeably in this paper.

For example, the theory *Boolean* uses the type *Bool* associated with the Boolean domain with two values $\{\top, \bot\}$, and has constant symbols *true* and *false*, and binary function symbols $\wedge$, and $\vee$, unary function symbol $\neg$, etc all with their usual interpretations. A more sophisticated theory is *Naturals*, the theory of the Natural numbers, that uses two types *Nat* and *Bool*. The type *Nat* is associated with the domain $\{0, 1, \ldots\}$ of the Natural numbers, and has constant symbols 0, 1, 2, ... and binary symbols $+$, $*$, etc of type $Nat \times Nat \to Nat$. Other function symbols in this theory are predicates $<$, $\leq$, ... of type $Nat \times Nat \to Bool$. All our theories include equality and also, for every type T, a ternary predicate `if` $\cdot$ `then` $\cdot$ `else` $\cdot$ of type $Bool \times T \to T$. For simplicity we restrict the rest of the paper to types *Nat* and *Bool*.

Definition 1 (Stream Expression). *Given a finite set Z of stream variable (each with a given type) the set of stream expressions is defined as follows:*

- *Variable: If s is a stream variable of type T, then s is a stream expression of type T;*
- *Function Application: Let $f : T_1 \times T_2 \times \cdots \times T_k \mapsto T$ be a k-ary function symbol. If for $1 \leq i \leq k$, e_i is an expression of type T_i, then $f(e_1, \ldots, e_k)$ is a stream expression of type T.*
- *Offset: If v is a stream variable of type T, c is a constant of type T, and k is an integer value, then $v[k, c]$ is a stream expression of type T.*

We use $Expr(Z)$ for the set of stream expressions using stream variables Z.

Constants c (that is, 0-ary function symbols) and stream variables v are called *atomic stream expressions*. Stream variables are used to represent streams. Informally, the offset term $v[k, c]$ refers to the value of v offset k positions from the current position, where a negative offset refers to a past position in the stream and a positive offset refers to a future position in the the stream. The constant c is the *default* value of type T assigned to positions from which the offset is past the end or before the beginning of the stream. For example $v[-1, true]$ refers to the previous position of stream v, with the value *true* when v does not have a previous position (that is when $v[-1, true]$ is evaluated at the begining of the trace).

A Lola specification describes a relation between input streams and output streams. A *stream* σ of type T and length N is a *finite* sequence of values from the domain of T; $\sigma(i)$, $i \geq 0$ denotes the value of the stream at time step i.

Definition 2 (Lola specification). *A* Lola specification $\varphi : \langle I, O, E \rangle$ *consists of:*

- *a finite set I of typed* independent *stream variables;*
- *a collection O of typed* dependent *stream variables; and*
- *a collections E of* defining expressions*, with one expression $E_y \in Expr(I \cup O)$ for each output variable $y \in O$, where y and E_y must have the same type.*

We write $y := E_y$ to denote that the stream y is defined by its defining expression E_y, which can use every stream variable in $I \cup O$ (including y itself) as atomic terms. Sometimes, LOLA specifications include a collection of *triggers* defined by expressions of type *Bool* over the stream variables, with the intended meaning of informing the user when the corresponding expressions become true, but we do not use triggers in the presentation in this paper.

Independent variables refer to input streams and dependent variables refer to output streams. It is often convenient to partition the dependent variables into output variables and intermediate variables to distinguish streams that are of interest to the user from those that are used only to facilitate the description of other streams. However, for the semantics and the algorithm this distinction is not important, and hence we will ignore this classification in the rest of the paper.

Example 1. Let x_1 and x_2 be stream variables of type Boolean and x_3 be a stream variable of type integer. The following is an example of a LOLA specification with $I = \{x_1, x_2, x_3\}$ as independent variables, $O = \{y_1, \ldots, y_{10}\}$ as dependent variables and the following defining equations:

$$
\begin{array}{ll}
y_1 := \mathbf{true} & y_6 := \texttt{if } x_1 \texttt{ then } x_3 \leq y_4 \texttt{ else } \neg y_3 \\
y_2 := x_3 & y_7 := x_1[+1, \mathit{false}] \\
y_3 := x_1 \vee (x_3 \leq 1) & y_8 := x_1[-1, \mathit{true}] \\
y_4 := ((x_3)^2 + 7) \ \mathit{mod}\ 15 & y_9 := y_9[-1, 0] + (x_3 \ \mathit{mod}\ 2) \\
y_5 := \texttt{if } y_3 \texttt{ then } y_4 \texttt{ else } y_4 + 1 & y_{10} := x_2 \ \vee \ (x_1 \ \wedge \ y_{10}[1, \mathit{true}])
\end{array}
$$

Stream variable y_1 denotes a stream whose value is *true* at all positions, while y_2 denotes a stream whose values are the same at all positions as those in x_3. The values of the streams corresponding to $y_3, \ldots, y_6$ are obtained by evaluating their defining expressions place-wise at each position. The stream corresponding to y_7 is obtained by taking at each position i the value of the stream corresponding to x_1 at position $i+1$, except at the last position, which assumes the default value *false*. Similarly for the stream for y_8, whose values are equal to the values of the stream for x_1 shifted by one position, except that the value at the first position is the default value *true*. The stream specified by y_9 counts the number of odd entries in the stream assigned to x_3 by accumulating (x_3 *mod* 2). Finally, y_{10} denotes the stream that gives at each position the value of the temporal formula $x_1 \mathcal{U} x_2$ with the stipulation that unresolved eventualities be regarded as satisfied at the end of the trace. □

To present formal results, it is sometimes convenient to work with a simpler class of specifications.

Definition 3 (Flat). *A specification is flat if each defining expression E_y is one of the following*

- *A constant* c
- *A stream variable* v
- *A constructor over stream variables* $f(v_1, \ldots, v_n)$
- *An offset expression* $v[k, c]$.

Definition 4 (Normalized). *A specification is normalized if it only contains offsets* 1 *or* -1.

Any Lola specification can be converted into a flat specification by introducing extra stream variables as place-holders for complex sub-expressions. Similarly, any Lola specification can be converted into a normalized specification by introducing additional stream variables defined to carry value $n-1$ for offsets of $n > 1$ (and $n+1$ for offsets of $n < -1$). This transformation also preserves flatness so every Lola specification can be converted into a normalized flat specification.

Example 2. Consider the Lola specification with $I = \{x_1, \ldots, x_5\}$, $O = \{y\}$ and

$$y := x_1[1, 0] + \texttt{if } x_2[-1, \mathit{true}] \texttt{ then } x_3 \texttt{ else } x_4 + x_5.$$

The normalized specification uses $O = \{y, y_1, \ldots, y_4\}$ with equations:

$$y := y_1 + y_2 \qquad \begin{array}{ll} y_1 := x_1[1, 0] & y_3 := x_2[-1, \mathit{true}] \\ y_2 := \texttt{if } y_3 \texttt{ then } x_3 \texttt{ else } y_4 & y_4 := x_4 + x_5 \end{array}$$

□

2.2 Specification Language: Semantics

In order to define the semantics of SRV specifications we first define how to evaluate expressions. Consider a map σ_I that assigns one stream σ_x of type T and length N for each input stream variable x of type T, and a map $\sigma_O : \{\ldots, \sigma_y, \ldots\}$ that contains one stream σ_y of length N for each defined stream variable y (again of the same type as y). We call (σ_I, σ_O) an interpretation of φ, and use σ as the map that assigns the corresponding stream as σ_I or σ_O (depending on whether the stream variable is an input variable or an output variable).

Definition 5 (Valuation). *Given an interpretation* (σ_I, σ_O) *a valuation is a map* $[\![\cdot]\!]$ *that assigns to each expression a stream of length* N *of the type of the expression as follows:*

$$\begin{array}{ll} [\![c]\!](j) & = c \\ [\![v]\!](j) & = \sigma_v(j) \\ [\![f(e_1, \ldots, e_k)]\!](j) & = f([\![e_1]\!](j), \ldots, [\![e_k]\!](j)) \\ [\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!](j) & = \texttt{if } [\![e_1]\!](j) \texttt{ then } [\![e_2]\!](j) \texttt{ else } [\![e_3]\!](j) \\ [\![v[k, c]]\!](j) & = \begin{cases} [\![v]\!](j+k) & \mathit{if}\ 0 \le j + k < N \\ c & \mathit{otherwise} \end{cases} \end{array}$$

We now can define when an interpretation (σ_I, σ_O) of φ is an *evaluation model*, which gives denotational semantics to Lola specifications.

Definition 6 (Evaluation Model). *An interpretation (σ_I, σ_O) of φ is an evaluation model of φ whenever*

$$[\![y]\!] = [\![E_y]\!] \quad \textit{for every } y \in O$$

In this case we write $(\sigma_I, \sigma_O) \models \varphi$.

For a given set of input streams, a Lola specification may have zero, one, or multiple evaluation models.

Example 3. Consider the Lola specifications (all with $I = \{x\}$ and $O = \{y\}$) where x has type *Nat* and y has type *Bool*.

$$\begin{aligned}\varphi_1 &: y := (x \leq 10)\\ \varphi_2 &: y := y \wedge (x \leq 10)\\ \varphi_3 &: y := \neg y\end{aligned}$$

For any given input stream σ_x, φ_1 has exactly one evaluation model (σ_x, σ_y), where $\sigma_y(i) = \mathit{true}$ if and only if $\sigma_x(i) \leq 10$, for $1 \leq i \leq N$. The specification φ_2, however, may give rise to multiple evaluation models for a given input stream. For example, for input stream $\sigma_x : \langle 0, 15, 7, 18\rangle$, both $\sigma_y : \langle \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}\rangle$ and $\sigma_y : \langle \mathit{false}, \mathit{true}, \mathit{false}, \mathit{true}\rangle$ make (σ_x, σ_y) an evaluation model of φ_2. The specification φ_3, on the other hand, has no evaluation models, because there is no solution to the equations $\sigma_y(i) = \neg\sigma_y(i)$. □

2.3 Well-definedness and Well-formedness

SRV specifications are meant to define monitors, which intuitively correspond to queries of observations of the system under analysis (input streams) for which we want to compute a unique answer (the output streams). Therefore, the intention of a specification is to define a function from input streams to output streams, and this requires that there is a unique evaluation model for each instance of the input streams. The following definition captures this intuition.

Definition 7 (Well-defined). *A* Lola *specification φ is* well-defined *if for any set of appropriately typed input streams σ_I of the same length $N > 0$, there exists a unique valuation σ_O of the defined streams such that $(\sigma_I, \sigma_O) \models \varphi$.*

A well-defined Lola specification maps a set of input streams to a unique set of output streams. Unfortunately well-definedness is a semantic condition that is hard to check in general (even undecidable for rich types). Therefore, we define a more restrictive (syntactic) condition called *well-formedness*, that can be easily checked on every specification φ and implies well-definedness. We first add an auxiliary definition.

Definition 8 (Dependency Graph). *Let φ be a* Lola *specification. The* dependency graph *for φ is the weighted directed multi-graph $D = \langle V, E\rangle$, with vertex set $V = I \cup O$. The set E contains an edge $y \xrightarrow{0} v$ if v is occurs in E_y and an edge $y \xrightarrow{k} v$ if $v[k, d]$ occurs in E_y.*

An edge $y \xrightarrow{k} v$ encode that y at a particular position potentially depends on the value of v, offset by k positions. Note that there can be multiple edges between x and y with different weights on each edge. Also note that vertices that correspond to input variables do not have outgoing edges.

A *walk* of a graph is a sequence $v_1 \xrightarrow{k_1} v_2 \xrightarrow{k_2} v_3 \cdots v_n \xrightarrow{k_n} v_{n+1}$ of vertices and edges. A walk is *closed* if $v_1 = v_{n+1}$. The weight of a walk is the sum of the weights of its edges. A simple walk is a walk in which no vertex is repeated. A *cycle* is a simple closed walk.

Definition 9 (Well-Formed Specifications). *A* Lola *specification φ is well-formed if its dependency graph has no closed walk with weight zero.*

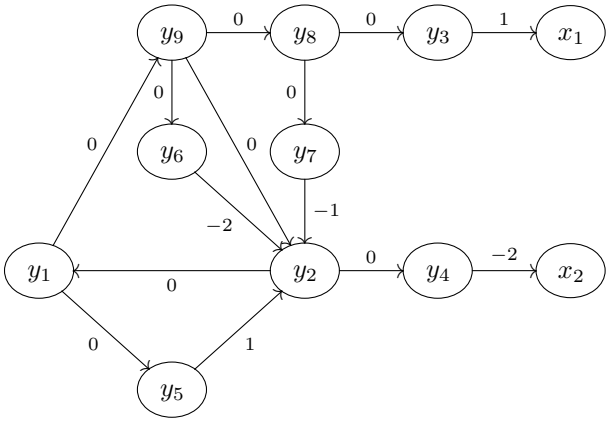
Example 4. Consider the Lola specification with $I : \{x_1, x_2\}$ and $O : \{y_1, y_2\}$ and the following defining equations:

$$y_1 := y_2[1,0] + \texttt{if } (y_2[-1,7] \leq x_1[1,0]) \texttt{ then } (y_2[-1,0]) \texttt{ else } y_2$$
$$y_2 := (y_1 + x_2[-2,1]).$$

Its normalized specification is

$$y_1 := y_5 + y_9 \qquad y_2 := y_1 + y_4 \qquad y_3 := x_1[1,0]$$
$$y_4 := x_2[-2,1] \qquad y_5 := y_2[1,0] \qquad y_6 := y_2[-1,0]$$
$$y_7 := y_2[-1,7] \qquad y_8 := y_7 \leq y_3 \qquad y_9 := \texttt{if } y_8 \texttt{ then } y_6 \texttt{ else } y_2$$

The dependency graph of the normalized specifications is:



This specification has a zero-weight closed walk, namely $y_1 \xrightarrow{0} y_9 \xrightarrow{0} y_2 \xrightarrow{0} y_1$, and hence the specification is not well-formed. □

To prove that well-formedness implies well-definedness, we first define the notion of an evaluation graph which captures the dependencies for a given input length N.

Definition 10 (Evaluation graph). *Given a specification φ and a length N, the evaluation graph is the directed graph $G_N : \langle V, E \rangle$ where V contains one vertex v^j for each position j of each stream variable v and*

- *there is an edge* $y^j \to v^j$ *if* E_y *contains* v *as an atom, and*
- *there is an edge* $y^j \to v^{j+k}$ *if* E_y *contains* $v[k, c]$ *and* $0 \leq j + k < N$.

The vertices v^j are called *position variables* as they encode the value of stream variable v at position j. We will prove later that a specification is guaranteed to be well-defined if no evaluation graph for any length contains a cycle, because in this case the value of each position variable can be uniquely determined. The following lemma relates this acyclicity notion with the absence of zero-weight cycles in the dependency graph.

Lemma 1. *Let* φ *be a specification with dependency graph* D*, let* N *be a trace length and* G_N *the explicit dependency graph. If* G_N *has a cycle then* D *has a zero-weight closed walk.*

Proof. Assume G_N has a cycle

$$y_1^{j_1} \to y_2^{j_2} \to \cdots \to y_k^{j_k} \to y_1^{j_1}$$

The corresponding closed walk in D is

$$y_1 \overset{j_2 - j_1}{\longrightarrow} y_2 \to \cdots \to y_k \overset{j_1 - j_k}{\longrightarrow} y_1$$

with weight $\sum_{i=1}^{k} (j_{i \oplus 1} - j_i) = 0$. □

Note that the closed walk induced in D needs not be a cycle since some of the intermediate nodes may be repeated, if they correspond to the same y_k for different position j.

Lemma 2. *Let* φ *be a specification and* N *a length. If* G_N *has no cycles, then for every tuple* σ_I *of input streams of length* N*, there is a unique evaluation model.*

Proof. Assume G_N has no cycles, so G_N is a DAG. Then we can define a topological order $>$ on G_N by taking the transitive closure of $\to$. We prove by induction on this order that the value of each vertex is uniquely determined, either because this value is obtained directly from an input stream or constant value in the specification, or because the value can be computed from values computed before according to $>$.

For the base case, the value of a vertex v^j without outgoing edges does not depend on other streams. The only possible value is either the value of an input stream (if v is an input stream variable) at position j, or a value obtained from an equation with no variables or offsets as atoms. In all these cases the value is uniquely determined.

For the inductive case, the value of v^j can be computed uniquely from the values of its adjacent vertices in G_N. Indeed, by Definition 10, if the value of v^j depends on the value of v^k then there exists an edge $v^j \to v^k$ in G_N and thus $v^j > v^k$ and, by the inductive hypothesis, the value of v^k is uniquely determined. Then, since every atom in $[\![E_v]\!](j)$ is uniquely determined, the value of $[\![E_v]\!](j)$ is uniquely determined. Since this value has been computed only from inputs, this is the only possible value for $\sigma_v(j)$ to form an evaluation model. □

Consider now a well-formed specification φ. Then, by Lemma 1, no evaluation graph has cycles, and thus by Lemma 2 for every set of input streams, there exists a unique solution for the output streams, and hence there is exactly one evaluation model.

Theorem 1. *Every well-formed* Lola *specification is well-defined.*

Note that the converse of Theorem 1 does not hold. First, the absence of cycles in G_N does not imply the absence of a zero-weight closed walk in D. For example, the evaluation graph for the specification

$$y_1 := y_2[-k, c]$$
$$y_2 := y_1[k, c]$$

for $N < k$ has no cycles (since it has no edges), but it is easy to see that D has a zero-weight closed walk. Second, a cycle in G_N does not necessarily imply that φ is not well-defined. For example, the evaluation graph of the specification

$$y := (z \ \vee \ \neg z) \ \wedge \ x$$
$$z := y$$

has a cycle for all N, but for every input stream, φ has exactly one evaluation model, namely $\sigma_y = \sigma_z = \sigma_x$, and thus, by definition, the specification is well-defined.

2.4 Checking Well-Formedness

A Lola specification φ is well-formed if its dependency graph D has no closed walks, so checking well-formedness is reduced to construct D and check for closed walks. In turn, this can be reduced to checking for cycles as follows.

Let a gez-cycle be a cycle in which the sum of the weight of the edges is greater than or equal to zero, and let a gz-cycle be a cycle in which the sum of the weight of the edges is strictly greater than zero. Similarly, a lez-cycle is a cycle where the sum is less than or equal zero and a lz-cycle is one where the sum is less than zero. The reduction is based on the observation that a graph has a zero-weight closed walk if and only if it has a maximally strongly component (MSCC) with both a gez-cycle and a lez-cycle.

Lemma 3. *A weighted and directed multigraph D has a zero-weight closed walk if and only if it has a vertex v that lies on both a gez-cycle and a lez-cycle.*

Proof. ($\Rightarrow$) Assume v is part of gez-cycle C_1 and lez-cycle C_2, with weights $w_1 \geq 0$ and $w_2 \leq 0$, respectively. The closed walk consisting of traversing w_1 times C_2 and then traversing $|w_2|$ times C_1 has weight $w_1 w_2 + |w_2| w_1 = 0$, as desired.

($\Leftarrow$) Assume D has a zero-weight closed walk. If D has a zero-weight cycle C we are done, as C is both a gez-cycle and a lez-cycle and any vertex in C has the desired property.

For the other case, assume D has no zero-weight cycles. It is easy to show by induction in the length of W that every closed walk can be decomposed into cycles that share one vertex. If one of these cycles is a lez-cycle or a gez-cycle the result follows. Now, not all the cycles can be strictly positive, because then the total weight of W would not be zero. Consequently there must a positive cycle and a negative cycle, and therefore there must be two consecutive cycles C_1 and C_2 that share one node and C_1 is positive and C_2 is negative. □

Theorem 2. *A directed weighted multigraph D has no zero-weight closed walk if and only if every MSCC has only gz-cycles or only lz-cycles.*

Proof. ($\Rightarrow$) Consider an arbitrary MSCC with only gz-cycles (the case for only lz-cycles is analogous). By the proof of Lemma 3, a closed walk is the multiset union of one or more cycles with weight the sum of the weights of the cycles. Hence the weight of any closed walk within the MSCC must be strictly greater than zero. Since any closed walk must stay within an MSCC, the weight of any closed walk must be strictly greater than zero.

($\Leftarrow$) Assume D has no zero-weight closed walk. Then, by Lemma 3, D has no vertex that lies on both a gez-cycle and a lez-cycle. Suppose D has an MSCC with a gz-cycle C_1 and a lz-cycle C_2. Consider an arbitrary vertex v_1 on C_1 and v_2 on C_2. If $v_1 = v_2 = v$, v lies on both a gez-cycle and a lez-cycle, a contradiction. If $v_1 \neq v_2$, since v_1 and v_2 are in the same MSCC, there exists a cycle C_3 that contains both v_1 and v_2. C_3 is either a zero-weight cycle, a gz-cycle or a lz-cycle. In all three cases either v_1 or v_2 or both lie on both a gez-cycle and a lez-cycle, a contradiction. □

Thus to check well-formedness of a SRV specification φ it is sufficient to check that each MSCC in G has only gz-cycles or only lz-cycles. This can be checked efficiently, even for large dependency graphs.

3 Online Monitoring

We distinguish two situations for monitoring—*online* and *offline* monitoring. In online monitoring, the traces from the system under observation are received as the system run, and the monitor works in tandem with the system. This leads to the following restriction for online monitoring: the traces are available a few points at a time starting at the initial instant on-wards, and need to be processed to make way for more incoming data. In particular, random access to the traces is not available. The length of the trace is assumed to be unknown upfront and very large.

In offline monitoring, on the other hand, we assume that the system has run to completion and the trace of data has been dumped to a storage device. Offline monitoring is covered in Section 4.

3.1 Monitoring Algorithm

We start by exhibiting a general monitoring algorithm for arbitrary LOLA specifications, and then study its efficiency. Let φ be a LOLA specification with independent stream variables I, dependent stream variables O and defining expressions E. Let j be the current position, at which the latest data is available from all input streams. The monitoring algorithm maintains two sets of equations as storage:

- *Resolved* equations R of the form (v^k, c) for a given position variable v^k (with $k \in \{1, \ldots, j\}$) and concrete value c.
- *Unresolved* equations U of the form (y^k, e) for position variable y^k expression e (for e different from a constant).

An equation (v^k, c) stored in R denotes that stream variable v at position k in the trace has been determined to have value c. This happens in two cases: input streams whose reading has been performed, and dependent stream variables whose value has been computed. Equations in U relate position variables y^k—where y is a dependent stream variable—with a (possibly partially simplified) expression over position variables whose values have not yet been determined. Note that if (y^k, e) is in U then e must necessarily contain at least one position variable, because otherwise e is a ground expression and the interpreted functions from the domain can transform e into a value.

The monitoring algorithm is shown in Algorithm 1. After initializing the U and R stores to empty and j to 0, the monitoring algorithm executes repeatedly the main loop (lines 5 to 11). This main loop first reads values for all inputs at the current position and adds these values to R (line 6). Then, it instantiates the defining equations for all outputs and adds these to U (line 7). Finally, it propagates new known values (v^k, c) in R by substituting all occurrences of v^k in unresolved equations by c and then simplifies resulting equations (procedure PROPAGATE). This procedure simply uses all the information in R to substitute occurrences of known values in unresolved equations. In some cases, these equations become resolved (the term becomes a value) and the corresponding pair is moved to R (lines 23 and 24). Then, the procedure PRUNE is used to eliminate unnecessary information from R as described below. Finally, procedure FINALIZE is invoked at the end of the trace. This procedure is used to determine whether a given offset expression that remains in an unresolved equation falls beyond the end of the trace, which is converted into its default value. This procedure also performs a final call to PROPAGATE, which is guaranteed (see below) to resolve all position variables, and therefore U becomes empty.

Procedure INST, shown in Algorithm 2, instantiates the defining equation for v into the corresponding equation for v^j at given position j by propagating the value into the atomic stream variable references and offsets atoms, which become instance variables. Note that the default value c is recorded in line 57 in case the computed position $k + j$ falls beyond the end of the trace N, which is not known at the point of the instantiation. Whether $k + j$ is inside the trace will be determined after k steps or resolved by FINALIZE.

Algorithm 1 Monitoring algorithm

```
1: procedure MONITOR
2:     U ← ∅
3:     R ← ∅
4:     j ← 0
5:     while not finished do
6:         R ← R ∪ {(y^j, σ_y(j)) | for every y ∈ I}          ▷ Add new inputs to R
7:         U ← U ∪ {(x^j, INST(e_x, j)) | for every x ∈ O}   ▷ Add output instances to U
8:         PROPAGATE()
9:         PRUNE(j)
10:        j ← j + 1
11:    end while
12:    N ← j + 1
13:    FINALIZE(N)
14: end procedure

15: procedure PROPAGATE
16:    repeat
17:        change ← false
18:        for all (v^k, e) ∈ U do                          ▷ Try to resolve every v^k in U
19:            e' ← simplify(subst(e, R))
20:            U.replace(v^k, e')                           ▷ update v^k
21:            if e' is value then
22:                change ← true                            ▷ v^k is resolved
23:                R ← R + {(v^k, e')}                      ▷ add v^k to R
24:                U ← U − {(v^k, e)}                       ▷ remove v^k from U
25:            end if
26:        end for
27:    until ¬change
28: end procedure

29: procedure PRUNE(j)
30:    for all (v^k, c) ∈ R do
31:        if ∇v + k ≤ j then                               ▷ Prune R
32:            R ← R − {(v^k, c)}
33:        end if
34:    end for
35: end procedure

36: procedure FINALIZE(N)
37:    for all (v^k, e) ∈ U do
38:        for all u_c^l subterm of e with l ≥ N do
39:            e ← e[u_c^l ← c]
40:        end for
41:        U.replace(v^k, e)
42:    end for
43:    PROPAGATE()
44: end procedure
```

Algorithm 2 Instantiate a defining expression for position j

```
45: procedure INST(expr, j)
46:     switch expr do
47:         case c
48:             return c
49:         case f(e_1, ..., e_n)
50:             return f(INST(e_1, j), ..., INST(e_n, j))
51:         case v
52:             return v^j
53:         case v[k, c]
54:             if k + j < 0 then
55:                 return c
56:             else
57:                 return v_c^{k+j}
58:             end if
59: end procedure
```

We show now how the resolved storage R can be pruned by removing information that is no longer necessary. The back reference distance of a stream variable represents the maximum time steps that its value needs to be remembered.

Definition 11 (Back Reference Distance). *Given a specification φ with dependency graph D the* back reference distance ∇v *of a vertex v is*

$$\nabla v = max(0, \left\{ k \mid s \xrightarrow{-k} v \in E \right\})$$

Example 5. We illustrate the use of back reference distances for pruning R (lines 31 and 32) revisiting Example 4. The back reference distances are $\nabla y_1 = \nabla y_{10} = \nabla y_{11} = \nabla y_{12} = \nabla y_{13} = \nabla y_{14} = \nabla y_{15} = \nabla y_{16} = \nabla x_1 = 0$ and $\nabla y_2 = \nabla x_2 = 2$. Consequently, all equations (v^j, c) are removed from R in the same time step that they are entered in R, except for y_2^j and x_2^j, which must remain in R for two time steps until instant $j + 2$. □

Example 6. Consider the following specification

$$y := q \vee (p \wedge z)$$
$$z := y[1, \mathit{false}]$$

which computes $p \,\mathcal{U}\, q$. For input streams $\sigma_p : \langle \mathit{false}, \mathit{false}, \mathit{true}, \mathit{false} \rangle$ and $\sigma_q : \langle \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false} \rangle$ the equations in stores R and U at the completion of

step (3) of the algorithm at each position are:

j	0	1	2	3
R	$p^0 = \mathit{false}$ $q^0 = \mathit{true}$ $y^0 = \mathit{true}$	$p^1 = \mathit{false}$ $q^1 = \mathit{false}$ $y^1 = \mathit{false}$ $z^0 = \mathit{false}$	$p^2 = \mathit{true}$ $q^2 = \mathit{false}$	$p^3 = \mathit{false}$ $q^3 = \mathit{false}$ $z^3 = \mathit{false}$ $y^3 = \mathit{false}$ $z^2 = \mathit{false}$ $y^2 = \mathit{false}$ $z^1 = \mathit{false}$
U	$z^0 = y^1$	$z^1 = y^2$	$y^2 = z^2$ $z^2 = y^3$ $z^1 = y^2$	$\emptyset$

Since the back reference distance of all stream variables is 0, all equations can be removed from R at each position. □

Theorem 3 (Correctness). *Let φ be a specification and σ_I be input streams of length N. If φ is well-formed, then Algorithm 1 computes the unique evaluation model of φ for σ_I. That is, at the end of the trace the unique value has been computed for each y^k, and U is empty.*

Proof. Assume φ is well-formed. By Definition 9 the dependency graph D has no zero-weight closed walks and hence by Lemma 1, the evaluation graph G_N has no cycles, and we can define a topological order $<$ in G_N.

As in the proof of Lemma 1, every vertex of G_N can be mapped to the corresponding value of the unique evaluation model. We prove by induction on G_N that at the end of the trace each of these values has been computed and that each value has been available in R at some point $j < N$ during the computation.

For the base case, leaf vertices v^j correspond to either input stream variables or values from equations of the form $x = c$ or $x = y[k, c]$ such that $j + k < 0$. In both cases the value is uniquely obtained and the corresponding equation is added to R.

For the inductive case, the value for vertex v^j is uniquely computed from the values for vertices w^k such that $v^j \to w^k$, and hence $w^k < v^j$ and by the inductive hypothesis, the value for w^k is uniquely computed or obtained and is at some point available in R. It remains to be shown that these values are available in R for substitution. We distinguish three cases:

1. $j = k$. In this case (v^j, e) and (w^k, e') are added to U (or R) at position j (either in line 6 or in line 7). If (w^k, e') is added to R, the value of w^k in e is substituted in e' in line 19. If (w^k, e') is added to U, by the inductive hypothesis, it is available at some later point in the computation. Then it must be moved to R in line 23, and hence in the same step it is substituted in e.
2. $j < k$. In this case (w^k, e') is added to U (or R) after (v^j, e) is added to U. Again, by the inductive hypothesis, (w^k, c) will be resolved and become

available in R at some position $l < N$ and thus at that same position is substituted in e if (v^j, e) is still in U.

3. $j > k$. In this case E_v contains $w[i, c]$ and thus $k = j + i$ (i.e. $i < 0$). Now, (w^k, c) is added to R or U before (v^j, e) is added to U. Again, by the inductive hypothesis, w^k will be resolved at some position $l \leq N$, which must be after k. By the definition of k, (w^k, c) will be in R at least until $k + \nabla w$ which is guaranteed to be at j or after and hence be available when v^j is added to U.

This finishes the proof. □

3.2 Efficiently Monitorable Specifications

In the general case the algorithm MONITOR described above is linear in both time and space in the length of the trace and the size of the specification. In these bounds, we assume that the value of a type can be stored in a single register of the type, and that a single function is computed in a single step.

In online monitoring, since the traces are assumed to be large, it is generally assumed that a specification can be monitored efficiently only if the memory requirements are independent of the trace length.

Example 7. Consider the following specification with $I = \{x\}$ and $O = \{y, last, w, z\}$:

$$\begin{aligned} y &:= false \\ last &:= y[1, true] \\ w &:= z[1, 0] \\ z &:= \texttt{if } last \texttt{ then } x \texttt{ else } w \end{aligned}$$

For the input stream σ_x $\langle 37, 31, 79, 17, 14\rangle$ the unique evaluation model is

σ_x	37	31	79	17	14
σ_y	*false*	*false*	*false*	*false*	*false*
σ_{last}	*false*	*false*	*false*	*false*	*true*
σ_w	14	14	14	14	0
σ_z	14	14	14	14	14

In general, for any input stream σ_x, output stream σ_z has all its values equal to the last value of σ_x. However, for all j, equations

$$(w^j, z_0^{j+1}) \quad \text{and} \quad (z^j, \texttt{if } last^j \texttt{ then } x^j \texttt{ else } w^j)$$

remain unresolved until the end of the trace, and thus the memory requirements of Algorithm 1 for this specification are linear in the length of the trace. □

The worst-case memory usage of a LOLA specification for a given trace length can be derived from the evaluation graph with the aid of the following definitions.

Definition 12 (Fan and Latency). *The fan of a vertex v^j of an evaluation graph G_N is the set of vertices reachable v^j:*

$$\mathit{fan}(v^j) \stackrel{def}{=} \left\{ w^k \mid v^j \rightarrow^* w^k \right\}$$

The latency of a position variable v^j is the difference between j and the position of the furthest vertex in $\mathit{fan}(v^j)$:

$$\mathit{lat}(v^j) \stackrel{def}{=} \mathit{max}\left(0, \left\{ k \mid w^{j+k} \in \mathit{fan}(v^j) \right\}\right).$$

The fan of v^j is an over-approximation of the set of vertices on which the value of v^j depends. The latency is an upper-bound on the number of trace steps it takes before a value at a given position is guaranteed to be resolved.

Theorem 4. *If a vertex v^j has latency k, then the corresponding equation (v^j, e) will be fully resolved by* Monitor *at or before step $j + k$.*

Proof. Since the specification is well-formed the evaluation graph is acyclic. We show the results by induction on a topological order $<$ of the evaluation graph. Note that if $v^j \rightarrow w^i$ then $\mathit{lat}(v^j) \geq \mathit{lat}(w^i)$ directly by the definition of latency. Then, at position $j+k$ it is guaranteed that w^i is resolved. Since all atoms in the expression e of equation (v^j, e) are resolved at $j+k$ or before, the corresponding values are subsituted in e (line 19) at step $j + k$ or before, so e is simplified into a value at $j + k$ or before. □

Example 8. Consider again the specification of Example 7. The latency of z^2 is $N - 2$, so equations for z^2 may reside in U for $N - 2$ steps, so this specification cannot be monitored online in a trace-length independent manner. □

Definition 13 (Efficiently Monitorable). *A* Lola *specification is efficiently monitorable if the worst case memory usage of* Monitor *is independent of the length of the trace.*

Some specifications that are not efficiently monitorable may be rewritten into equivalent efficiently monitorable form, as illustrated by the following example.

Example 9. Consider the specification "*Every request must be eventually followed by a grant before the trace ends*", expressed as φ_1 as follows:

$$\begin{aligned} \mathit{reqgrant} &:= \texttt{if } \mathit{request} \texttt{ then } \mathit{evgrant} \texttt{ else } \mathit{true} \\ \mathit{evgrant} &:= \mathit{grant} \ \vee \mathit{nextgrant} \\ \mathit{nextgrant} &:= \mathit{evgrant}[1, \mathit{false}] \end{aligned}$$

This specification encodes the temporal assertion $\square(\mathit{request} \rightarrow \diamondsuit \mathit{grant})$. Essentially, *evgrant* captures $\diamondsuit \mathit{grant}$ and *reqgrant* corresponds to $\square(\mathit{request} \rightarrow \diamondsuit \mathit{grant})$ (see [12] and [10] for a description of translation from LTL to Boolean SRV). An alternative specification φ_2 of the same property is

$$\begin{aligned} \mathit{waitgrant} &:= \neg \mathit{grant} \wedge (\mathit{request} \vee \mathit{nextgrant}) \\ \mathit{nextgrant} &:= \mathit{waitgrant}[-1, \mathit{false}] \\ \mathit{ended} &:= \mathit{false}[1, \mathit{true}] \end{aligned}$$

It is easy to see that, for the same input, *ended* $\wedge$ *waitgrant* is true at the end of the trace (for φ_2) if and only if $\neg$*nextgrant* is true at the beginning of the trace for φ_1. Hence, both specifications can report a violation at the end of the trace if a request was not granted. The second specification, however, is efficiently monitorable, while the first one is not. □

Similar to the notion of well-definedness, checking whether a specification is efficiently monitorable is a semantic condition and cannot be checked easily in general. Therefore we define a syntactic condition based on the dependency graph of a specification that guarantees that a specification is efficiently monitorable.

Definition 14 (Future Bounded). *A well-formed specification φ is future bounded if its dependency graph D has no positive-weight cycles.*

We show that every future bounded specification is efficiently monitorable by showing that in the absence of positive-weight cycles every vertex in the dependency graph can be mapped to a non-negative integer that provides an upper-bound on the number of trace steps required to resolve the equation for the corresponding instance variable.

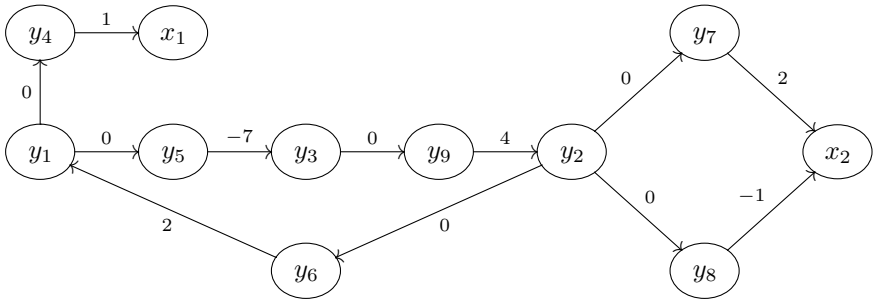
Definition 15 (Look-ahead Distance). *Given a future bounded specification with dependency graph D, the look-ahead distance Δv of a vertex v is the maximum weight of a walk starting from v (or zero if the maximum weight is negative).*

Note that the look-ahead distance is well defined only in the absence of positive-weight cycles. The look-ahead distance of a vertex can be computed easily using shortest path traversals on the dependency graph D.

Example 10. Consider the specification

$y_1 := y_4 \wedge y_5$ $\quad$ $y_4 := p[1, \mathit{false}]$ $\quad$ $y_7 := q[2, 0]$
$y_2 := \texttt{if } y_6 \texttt{ then } y_7 \texttt{ else } y_8$ $\quad$ $y_5 := y_3[-7, \mathit{false}]$ $\quad$ $y_8 := q[-1, 2]$
$y_3 := y_9 \leq 5$ $\quad$ $y_6 := y_1[2, \mathit{true}]$ $\quad$ $y_9 := y_2[4, \mathit{true}]$

The dependency graph D of this specification is:



Consequently, the values of the look-ahead distance are:

$$\begin{array}{lll} \Delta y_1 = 1 & \Delta y_4 = 1 & \Delta y_7 = 2 \\ \Delta y_2 = 3 & \Delta y_5 = 0 & \Delta y_8 = 0 \\ \Delta y_3 = 7 & \Delta y_6 = 3 & \Delta y_9 = 7 \end{array}$$

which are easily computer from D. □

The look-ahead distance provides an upper-bound on the number of equations that may simultaneously be in U.

Lemma 4. *For every vertex v^j in an evaluation graph G_N of a future bounded specification $lat(v^j) \leq \Delta c$.*

Proof. Consider a vertex v^j in an evaluation graph, with latency $lat(v, j) = d$. Then, there exists a sequence of vertices

$$v^j \to y_1^{j_1} \to \ldots \to y_n^{j_n}$$

with $j_n - j = d$. The walk in the dependency graph of the corresponding vertices

$$v \overset{j_1-j}{\longrightarrow} y_1 \overset{j_2-j_1}{\longrightarrow} \ldots \overset{j_n-j_{n-1}}{\longrightarrow} y_n$$

has total weight

$$\sum_{i=1}^{n} j_{i+1} - j_i \;=\; j_n - j_i \;=\; d$$

and hence $lat(v^j) \leq \Delta v$. □

Theorem 5 (Memory Requirements). *Let φ be a future bounded specification. Algorithm 1 requires to store in U and R, at any point in time, a number of equations linear in the size of φ.*

Proof. From the description of the algorithm and Lemma 4 it follows that the maximum number of equations in U is less than or equal to

$$\sum_{y \in O} \Delta y \; + |O|$$

where the second term reflects that all equations for the dependent variables are first stored in U in line 7 and after simplification moved to R in line 23.

Moreover, the maximum number of equations stored in R is bounded by ∇v and the number of stream variables v. □

Example 11. Consider again the specification of Example 10. The back reference distance is 0 for all variables except for x_2 and y_3, which are $\nabla x_2 = 1, \nabla y_3 = 7$. Hence, at the end of every main loop, R only contains one instance of x_2 and seven instances of y_3. Additionally, the look-ahead distance of a stream variable v bounds linearly the number of instances of v in U. □

Corollary 1. *Every future-bounded specification is efficiently monitorable.*

Note that the converse does not hold. In practice, it is usually possible to rewrite an online monitoring specification with a positive cycle into one without positive cycles, as illustrated in Example 9.

4 Offline Monitoring

In offline monitoring we assume that all trace data is available on tape, and therefore we can afford more flexibility in accessing the data. In this section we show that every well-formed SRV specification can be monitored efficiently offline, in contrast to online monitoring where we required that the dependency graph not have any positive-weight cycles. The reason why we can efficiently monitor in an offline manner all specifications is that we can perform both forward and backward passes over the trace. We will show that every well-formed specification can be decomposed into sub-specifications such that each sub-specification needs to be checked only once and can be done so efficiently by either traversing the trace in a forward or in a backward direction. In this manner, all values of the output streams of a sub-specification can written to tape and are accessible for subsequent traversals.

We first define the notions of *reverse efficiently monitorable* and its corresponding syntactic condition, *past bounded*, as the duals of efficiently monitorable and future bounded. A reverse monitoring algorithm RevMonitor can be easily obtained by initializing j to N (line 4) decreasing j (line 10), pruning j on the dual of the back-reference distance in line 31 and performing substitutions when the offset becomes negative (so Finalize is not necessary for reverse monitoring). This is essentially the same algorithm as Monitor but performing the index transformation $j' = N - (j + 1)$.

Definition 16 (Reverse Efficiently Monitorable). *A* Lola *specification is reverse efficiently monitorable if its worst-case memory requirement when applying* RevMonitor *is independent of the length of the trace.*

Definition 17 (Past Bounded). *A well-formed* Lola *specification is past bounded if its dependency graph has no negative-weight cycles.*

Lemma 5. *Every past-bounded specification is reverse efficiently monitorable.*

Proof. The dual of the argument for Corollary 1. □

We construct now an offline algorithm that can check a well-formed Lola specification in a sequence of forward and reverse passes over the tapes, such that the number of passes is linear in the size of the specification and each pass is trace-length indepedent.

Let φ be a well-formed specification with dependency graph D. From the definition of well-formedness it follows that D has no zero-weight cycles, so each MSCC consists of only negative-weight or only positive-weight cycles. Let $G_M : \langle \{V_p, V_n\}, E_M \rangle$ be the graph induced by the MSCCs of D defined as follows. For each positive-weight MSCC in D there is a vertex in V_p and for each negative-weight MSCC in D there is a vertex in V_n. For each edge between two MSCCs there is an edge in E_M connecting the corresponding vertices. Clearly, G_M is a DAG.

Now we assign each MSCC a stage that will determine the order of computing the output for each MSCC following the topological order of G_M. Positive

MSCCs will be assigned even numbers and negative MSCCs will be assigned odd numbers. Every MSCC will be assigned the lowest stage possible that is higher than that of all its descendants with opposite polarity. In other words, the stage of an MSCC v is at least the number of alternations in a path in G_M from v. Formally, let the opposite descendants be defined as follows:

$$op(v) = \{v' \mid (v, v') \in E_M^* \text{ and } (v \in V_p \text{ and } v' \in V_n, \text{ or } v \in V_n \text{ and } v' \in V_p)\}$$

Then,

$$stage(v) = \begin{cases} 0 & \text{if } op(v) \text{ is empty and } v \in V_n \\ 1 & \text{if } op(v) \text{ is empty and } v \in V_p \\ 1 + max\{stage(v') \mid v' \in op(v)\} & \text{otherwise} \end{cases}$$

which can be computed following a topological order on G_M. Each vertex v in G_M can be viewed as representing a sub-specification φ_v whose defining equations refer only to stream variables in sub-specifications with equal or lower stage processing order. Based on this processing order we construct the following algorithm.

Algorithm 3 Offline Trace Processing

```
1: procedure OfflineMon
2:     for i = 0 to max(stage(v)) with increment 2 do
3:         for all v with stage(v) = i do
4:             Monitor(φ_v)                          ▷ Forward pass
5:         end for
6:         for all v with stage(v) = i + 1 do
7:             RevMonitor(φ_v)                       ▷ Backward pass
8:         end for
9:     end for
10: end procedure
```

Theorem 6. *Given a well-formed specification, a trace can be monitored in time linear in the size of the specification and the length of the trace, with memory requirements linear in the size of the specification and independent of the length of the trace.*

Proof. Follows directly from Lemmas 1 and 5 and Algorithm 3. □

Example 12. Figure 1 shows the dependency graph of a Lola specification and its decomposition into MSCCs, along with its induced graph G_M annotated with the processing order of the vertices. MSCCs G_1 and G_4 are positive, while G_2 and G_5 are negative. G_3 is a single node MSCC with no edges, which can be chosen to be either positive or negative. The passes are: G_5 is first monitored forward because it is efficiently monitorable. Then, G_4 is monitored backwards. After that, G_3 and G_2 are monitored forward. Finally, G_1 is monitored backwards.

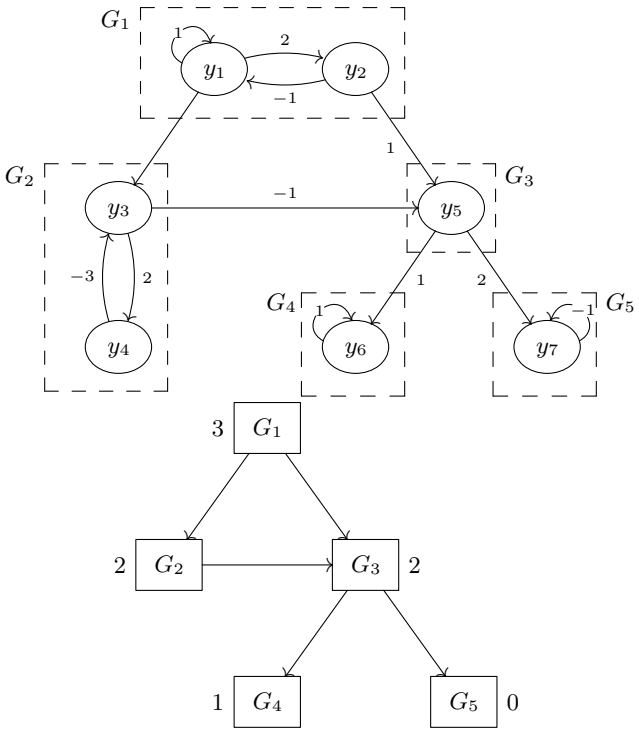


Fig. 1. A dependency graph G and its MSCC induced graph G_M.

5 Conclusions

We have revisited Stream Runtime Verification, a formalism for runtime verification based on expressing the functional relation between input streams and output streams, and we have presented in detail evaluation strategies for online and offline monitoring.

SRV allows both runtime verification of temporal specifications and collection of statistical measures that estimate coverage and specify complex temporal patterns. The Lola specification language is sufficiently expressive to specify properties of interest to applications like large scale testing, and engineers find the language easy to use. Even specifications with more than 200 variables could be constructed and understood relatively easily by engineers. Even though the language allows ill-defined specifications, SRV provides a syntactic condition that is easy to check and that guarantees well-definedness, using the notion of a dependency graph. Dependency graphs are also used to check whether a specification is efficiently monitorable online, that is, in space independent of the trace length. In practical applications most specifications of interest are in fact efficiently monitorable or can be rewritten into an efficiently monitorable fashion. We revisited the online algorithm for Lola specifications, and presented

an algorithm for offline monitoring whose memory requirements are independent of the trace length for any well-formed specification.

The design of the Lola specification language was governed by ease of use by engineers. In runtime verification, unlike in static verification, one is free to choose Turing-complete specification languages. As a result researchers have explored the entire spectrum from temporal logics to programming languages. The advantage of a programming language in comparison with a temporal logic is that a declarative programming language is more familiar to engineers and large specifications are easier to write and understand. The disadvantage is that the semantics is usually tied to the evaluation strategy (typically in an informal implicit manner) and the complexity is hard to determine, while the semantics of a logic is independent of the evaluation strategy and upper bounds for its complexity are known. In practice, the choice is motivated by the intended use. Stream Runtime Verification is usually employed to facilitate the task of writing large specifications for engineers, so the natural choice was a programming-language. SRV retains, however, most of the advantages of a logic: the semantics is independent of the evaluation strategy and the efficiently monitorable specifications provide a clear bound on complexity. See [10] where we study decision procedures and complexities of decision problems for Boolean Stream Runtime Verification. For example, comparing Lola with specification languages at the other end of the spectrum, such as Eagle [3], Lola usually allows simpler specifications, as illustrated in Figure 2.

Eagle encoding of the operator $\Box_p F$ with intended meaning that F holds with at least probability p over a given sequence [3]:

$$\begin{aligned}
&\underline{\min}\ A(\mathit{Form}\ F, \mathit{float}\ p, \mathit{int}\ f, \mathit{int}\ t) = \\
&\left(\bigcirc \mathit{Empty}() \wedge \left(\begin{array}{ll} (F & \wedge (1 - \frac{f}{t}) \geq p) \\ & \vee \\ (\neg F & \wedge (1 - \frac{f+1}{t}) \geq p) \end{array}\right)\right) \vee \\
&\left(\neg \mathit{Empty}() \wedge \left(\begin{array}{ll} (F & \rightarrow \bigcirc A(F, p, f, t+1)) \\ & \wedge \\ (\neg F & \rightarrow \bigcirc A(F, p, f+1, t+1)) \end{array}\right)\right)
\end{aligned}$$

Corresponding Lola specification:

$$\begin{aligned}
\mathit{countF} &= F[1,0] + \texttt{ite}(F, 1, 0) \\
\mathit{total} &= \mathit{total}[1,0] + 1 \\
\mathit{BoxFp} &= (F/\mathit{total}) \geq p
\end{aligned}$$

Fig. 2. Comparison between the Lola and Eagle specification language

References

1. Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.
2. Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *Proc of the 18th Int'l Symp. on Formal Methods (FM'12)*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.
3. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proc. of VMCAI'04*, LNCS 2937, pages 44–57. Springer, 2004.
4. Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*. Springer, 2018.
5. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In *Proc. of RV'13*, volume 8174 of *LNCS*, pages 59–75. Springer, 2013.
6. Andreas Bauer, Martin Leucker, and Chrisitan Schallhart. Runtime verification for LTL and TLTL. *ACM T. Softw. Eng. Meth.*, 20(4):14, 2011.
7. Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *Proc. of the 26th Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*, volume 4337 of *LNCS*, pages 260–272. Springer, 2006.
8. Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the uglybut how ugly is ugly? In *Proc. of RV'07*, volume 4839 of *LNCS*, pages 126–138. Springer, 2007.
9. Gérard Berry. *Proof, language, and interaction: essays in honour of Robin Milner*, chapter The foundations of Esterel, pages 425–454. MIT Press, 2000.
10. Laura Bozelli and César Sánchez. Foundations of Boolean stream runtime verification. In *In Proc. RV'14*, volume 8734 of *LNCS*, pages 64–79. Springer, 2014.
11. Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. TeSSLa: Temporal stream-based specification language. In *Proc. of the 21st. Brazilian Symp. on Formal Methods (SBMF'18)*, LNCS. Springer, 2018.
12. Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime monitoring of synchronous systems. In *Proc. of TIME'05*, pages 166–174. IEEE CS Press, 2005.
13. Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. of CAV'03*, volume 2725 of *LNCS 2725*, pages 27–39. Springer, 2003.
14. Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In *Proc. of the 16th Int'l Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*, pages 152–168. Springer, 2016.
15. Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. Real-time stream-based monitoring. *CoRR*, abs/1711.03829, 2017.
16. Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny B. Sipma. Collecting statistics over runtime executions. *ENTCS*, 70(4):36–54, 2002.
17. Thierry Gautier, Paul Le Guernic, and Löic Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In *Proc. of FPCA'87*, LNCS 274, pages 257–277. Springer, 1987.

18. Alwyn E. Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical report, NASA Langley Research Center, 2010.
19. Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. In *Proc. of RV'18*, LNCS. Springer, 2018.
20. Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. of IEEE*, 79(9):1305–1320, 1991.
21. Klaus Havelund and Allen Goldberg. Verify your runs. In *Proc. of VSTTE'05*, LNCS 4171, pages 374–383. Springer, 2005.
22. Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Proc. of TACAS'02*, LNCS 2280, pages 342–356. Springer, 2002.
23. Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. TeSSLa: Runtime verification of non-synchronized real-time streams. In *Proc. of the 33rd Symposium on Applied Computing (SAC'18)*. ACM, 2018.
24. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Logic Algebr. Progr.*, 78(5):293–303, 2009.
25. Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Proc. of RV'10*, LNCS 6418. Springer, 2010.
26. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *Proc. of FM'06*, LNCS 4085, pages 573–586. Springer, 2006.
27. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Autom. Softw. Eng.*, 12(2):151–197, 2005.
28. Koushik Sen and Grigore Roşu. Generating optimal monitors for extended regular expressions. *ENTCS*, 89(2):226–245, 2003.