LOLA: Runtime Monitoring of Synchronous Systems

Ben D'Angelo * Sriram Sankaranarayanan * César Sánchez * Will Robinson *
Bernd Finkbeiner † Henny B. Sipma * Sandeep Mehrotra ‡ Zohar Manna *

* Computer Science Department, Stanford University, Stanford, CA 94305
{bdangelo,srirams,cesar,sipma,manna}@theory.stanford.edu

† Department of Computer Science, Saarland University
finkbeiner@cs.uni-sb.de

‡ Synopsys, Inc.

Abstract— **We present a specification language and algorithms for the online and offline monitoring of synchronous systems including circuits and embedded systems. Such monitoring is useful not only for testing, but also under actual deployment. The specification language is simple and expressive; it can describe both correctness/failure assertions along with interesting statistical measures that are useful for system profiling and coverage analysis. The algorithm for online monitoring of queries in this language follows a partial evaluation strategy: it incrementally constructs output streams from input streams, while maintaining a store of partially evaluated expressions for forward references. We identify a class of specifications, characterized syntactically, for which the algorithm's memory requirement is independent of the length of the input streams. Being able to bound memory requirements is especially important in online monitoring of large input streams. We extend the concepts used in the online algorithm to construct an efficient offline monitoring algorithm for large traces.**

We have implemented our algorithm and applied it to two industrial systems, the PCI bus protocol and a memory controller. The results demonstrate that our algorithms are practical and that our specification language is sufficiently expressive to handle specifications of interest to industry.

I. INTRODUCTION

Monitoring synchronous programs for safety and liveness properties is an important aspect of ensuring their proper runtime behavior. An offline monitor analyzes traces of a system post-simulation to spot violations of the specification. Offline monitoring is critical for testing large systems before deployment. An online monitor processes the system trace while it is being generated. Online monitoring is used to detect violations of the specification when the system is in operation so that they can be handled before they translate into observable and cascading failures, and to adaptively optimize system performance.

Runtime monitoring has received growing attention in recent years [1], [2], [3]. While static verification intends to show that every (infinite) run of a system satisfies the specification, runtime monitoring is concerned only with a single (finite) trace. Runtime monitoring can be viewed as an extension of testing with more powerful specification languages.

The offline monitoring problem is known to be easy for purely past or purely future properties. It is well known that for past properties, the online monitoring problem can be solved efficiently using constant space and linear time in the trace size. For future properties, on the other hand, the space requirement generally depends on the length of the trace, which suggests that online monitoring may quickly become intractable in practical applications with traces exceeding 10^6 simulation steps.

In this paper, we present a specification language, intended for industrial use. The language can express properties involving both the past and the future. It is a functional stream computation language like LUSTRE [4] and ESTEREL [5], with features that are relevant to our problem at hand. It is parsimonious in its number of operators (expressions are constructed from three basic operators), but the resulting expressiveness surpasses temporal logics and many other existing formalisms

This research was supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, by ARO grant DAAD19-01-1-0723, by NAVY/ONR contract N00014-03-1-0939, by the Siebel Graduate Fellowship, and by the BMBF grant 01 IS C38 B as part of the Verisoft project.

including finite-state automata.

We provide a syntactic characterization of *efficiently monitorable* specifications, for which the space requirement of the online monitoring algorithm is independent of the size of the trace, and linear in the specification size. An analysis of some industrial specifications provided by Synopsys, Inc. showed that a large majority of these specifications lie in this efficiently monitorable class. For the offline monitoring problem, we demonstrate an efficient monitoring strategy in the presence of mixed past/future properties.

We have implemented our algorithm and specification language in a system called LOLA. LOLA accepts a specification in the form of a set of stream expressions, and is then run on a set of input streams. Two types of specifications are supported: properties that specify correct behavior, and properties that specify *statistical measures* that allow profiling the system that produces the input streams. An execution of LOLA computes arithmetic and logical expressions over the *finite* input and intermediate streams to produce an output consisting of error reports and the desired statistical information.

A. Related Work

Much of the initial work on runtime monitoring (cf. [6], [7], [8]) was based on temporal logic [9]. In [10], non-deterministic automata are built from LTL to check violations of formulas over finite traces and the complexity of these problems is studied. LTL based specifications have already been pursued in tools such as the Temporal Rover [7] and Java PathExplorer [11]. One limitation of this approach is that the logic must be adapted to handle truncated traces. The approach taken in [12] considers extensions of LTL for the case of truncated paths with different interpretations (weak and strong) of the next operator at the end of the trace. The choice of handling success/failure on a finite trace frequently depends on the situation being modeled.

Another important difference between runtime verification and static verification is that liveness properties can never be violated on a finite trace. An appealing solution is to extend the specification language to compute *quantitative measures* based on the trace. Temporal properties can be specified in LOLA, but one of the main goals is to go beyond property checking to the collection of numerical statistics. For example, instead of checking the property "there are only finitely many retransmissions of each package," which is vacuously true over finite traces, we desire to evaluate queries like "what is the average number of retransmissions." Our first approach to combine the property proving with data collection appeared in [13]. Following this trend, runtime verifiers can be used not only for bug-finding, but also for profiling, coverage, vacuity and numerous other analyses.

LOLA models runtime verification as a *stream computation*. The definition of LOLA output streams in terms of other streams resembles synchronous programming languages (notably LUSTRE [4], ESTEREL [5], Signal [14]), but there is a significant difference: these languages are designed primarily for the construction of synchronous systems. Therefore, output values for a time instant are computed directly from values at the same and *previous* instants. This assumption makes perfect sense if we desire that the systems we specify be *executable*, and therefore be *causal*. However, runtime specifications are *descriptive* in nature. They include future formulas whose evaluation may have to be delayed until future values arrive. This requires stronger expressiveness in the language and the corresponding evaluation strategies.

Other efforts in run-time verification include [15], which studies the efficient generation of monitors from specifications written as extended regular expressions, and [16], which studies rewriting techniques for the efficient evaluation of LTL formulas on finite execution traces, both online and offline. In [8], an efficient method for the online evaluation of *past* LTL properties is presented. This method exploits that past LTL can be recursively defined using only values in the previous state of the computation. Our *efficiently monitorable* specifications generalize this idea, and apply it uniformly to both verification and data collection.

The system that most closely resembles LOLA is Eagle [17]. Eagle allows the description of monitors based on greatest and least fixed points of recursive definitions. Many logical formalisms used to describe properties, including past and future LTL formulas, can be translated to Eagle specifications. These are then compiled into a set of rules that implements the monitor. 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.

II. LOLA OVERVIEW

In this section we describe the specification language. The monitoring algorithms will be presented in Section III.

A. Specification Language: Syntax

A LOLA specification describes the computation of output streams from a given set of input streams. A *stream* σ of type T is a *finite* sequence of values from T. We let $\sigma(i)$, $i \geq 0$ denote the value of the stream at time step i.

Definition 1 (LOLA specification) A LOLA specification is a set of equations over typed *stream variables*, of the form

$$\begin{array}{lcl} s_1 & = & e_1(t_1, \ldots, t_m, s_1, \ldots, s_n) \\ \vdots & & \vdots \\ s_n & = & e_n(t_1, \ldots, t_m, s_1, \ldots, s_n), \end{array}$$

where $s_1, \ldots, s_n$ are called the *dependent variables* and $t_1, \ldots, t_m$ are called the *independent variables*, and $e_1, \ldots, e_n$ are *stream expressions* over $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$. Independent variables refer to input streams and dependent variables refer to output streams*. A LOLA specification can also declare certain output boolean variables as *triggers*. Triggers generate notifications at instants when their corresponding values become $true$. Triggers are specified in LOLA as

$$\textbf{trigger } \varphi$$

where φ is a boolan expression over streams.

A stream expression is constructed as follows:

- If c is a constant of type T, then c is an *atomic stream expression* of type T;
- If s is a stream variable of type T, then s is an *atomic stream expression* of type T;
- Let $f : T_1 \times T_2 \times \cdots \times T_k \mapsto T$ be a k-ary operator. 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;
- If b is a boolean stream expression and e_1, e_2 are stream expressions of type T, then $\texttt{ite}(b, e_1, e_2)$ is a stream expression of type T; note that `ite` abbreviates *if-then-else*.
- If e is a stream expression of type T, c is a constant of type T, and i is an integer, then $e[i, c]$ is a stream expression of type T. Informally, $e[i, c]$ refers to the value of the expression e offset i positions from the current position. The constant c indicates the *default* value to be provided, in case an offset of i takes us past the end or before the beginning of the stream.

*In our implementation we partition the dependent variables into output variables and intermediate variables to distinguish streams that are of interest to the user and those that are used only to facilitate the computation of other streams. However, for the description of the semantics and the algorithm this distinction is not important, and hence we will ignore it in this paper.

Example 1 Let t_1, t_2 be stream variables of type boolean and t_3 be a stream variable of type integer. The following is an example of a LOLA specification with t_1, t_2 and t_3 as independent variables:

$$\begin{array}{lcl} s_1 & = & \mathbf{true} \\ s_2 & = & t_3 \\ s_3 & = & t_1 \vee (t_3 \leq 1) \\ s_4 & = & ((t_3)^2 + 7)\ mod\ 15 \\ s_5 & = & \texttt{ite}(s_3, s_4, s_4 + 1) \\ s_6 & = & \texttt{ite}(t_1, t_3 \leq s_4, \neg s_3) \\ s_7 & = & t_1[+1, \mathbf{false}] \\ s_8 & = & t_1[-1, \mathbf{true}] \\ s_9 & = & s_9[-1, 0] + (t_3\ mod\ 2) \\ s_{10} & = & t_2\ \vee\ (t_1\ \wedge\ s_{10}[1, \mathbf{true}]) \end{array}$$

Stream variable s_1 denotes a stream whose value is $\mathbf{true}$ at all positions, while s_2 denotes a stream whose values are the same at all positions as those in t_3. The values of the streams corresponding to $s_3, \ldots, s_6$ are obtained by evaluating their defining expressions place-wise at each position. The stream corresponding to s_7 is obtained by taking at each position i the value of the stream corresponding to t_1 at position $i + 1$, except at the last position, which assumes the default value **false**. Similarly for the stream for s_8, whose values are equal to the values of the stream for t_1 shifted by one position, except that the value at the first position is the default value $\mathbf{true}$. The stream specified by s_9 counts the number of odd entries in the stream assigned to t_3 by accumulating $(t_3\ mod\ 2)$. Finally, s_{10} denotes the stream that gives at each position the value of the temporal formula $t_1 \mathbf{Until}\ t_2$ with the stipulation that unresolved eventualities be regarded as satisfied at the end of the trace.

B. Specification Language: Semantics

The semantics of LOLA specifications is defined in terms of *evaluation models*, which describe the relation between input streams and output streams.

Definition 2 (Evaluation Models) Let φ be a LOLA specification over independent variables $t_1, \ldots, t_m$ with types $T_1, \ldots, T_m$, and dependent variables $s_1, \ldots, s_n$ with types $T_{m+1}, \ldots, T_{m+n}$. Let $\tau_1, \ldots, \tau_m$ be streams of length $N{+}1$, with τ_i of type T_i. The tuple $\langle \sigma_1, \ldots, \sigma_n \rangle$ of streams of length $N + 1$ with appropriate types is called an *evaluation model*, if for each equation in φ

$$s_i = e_i(t_1, \ldots, t_m, s_1, \ldots, s_n),$$

$\langle \sigma_1, \ldots, \sigma_n \rangle$ satisfies the following *associated equations*:

$$\sigma_i(j) = val(e_i)(j) \qquad \text{for } 0 \leq j \leq N$$

where $val(e)(j)$ is defined as follows. For the base cases:

$$\begin{aligned} val(c)(j) &= c \ . \\ val(t_i)(j) &= \tau_i(j) \ . \\ val(s_i)(j) &= \sigma_i(j) \ . \end{aligned}$$

For the inductive cases:

$$\begin{aligned} & val(f(e_1,\ldots,e_k)(j) = \\ & \quad f(val(e_1)(j),\ldots,val(e_k)(j)) \ . \\ & val(\mathtt{ite}(b,e_1,e_2))(j) = \\ & \quad \mathit{if}\, val(b)(j) \ \mathit{then}\ val(e_1)(j)\ \mathit{else}\ val(e_2)(j) \ . \\ & val(e[k,c])(j) = \\ & \quad \begin{cases} val(e)(j+k) & \mathit{if}\ 0 \le j+k \le N, \\ c & \mathit{otherwise} \ . \end{cases} \end{aligned}$$

The set of all equations associated with φ is denoted by φ_σ.

Example 2 Consider the LOLA specification

$$\varphi: \ s \ = \ t_1[1,0] + \mathtt{ite}(t_2[-1,\mathbf{true}], t_3, t_4 + t_5).$$

The associated equations φ_σ are

$$\sigma(j) = \begin{cases} \tau_1(j+1) + \mathtt{ite}\begin{pmatrix} \tau_2(j-1), \\ \tau_3(j), \\ \tau_4(j)+\tau_5(j) \end{pmatrix} & j \in [1,N), \\ \mathtt{ite}\begin{pmatrix} \tau_2(N-1), \\ \tau_3(N), \\ \tau_4(N)+\tau_5(N) \end{pmatrix} & j = N, \\ \tau_1(1) + \tau_3(0) & j = 0. \end{cases}$$

A LOLA specification is *well-defined* if for any set of appropriately typed input streams, all of the same length, it has exactly one evaluation model.

Example 3 Consider the LOLA specification

$$\varphi_1: \ s_1 = (t_1 \le 10).$$

For the stream $\tau_1 : 0,\ldots,100$, the associated equations are $\sigma_1(j) = (\tau_1(j) \le 10)$. The only evaluation model of φ_1 is the stream $\sigma_1(i) = \mathit{true}$ iff $i \le 10$. In fact, this LOLA specification is well-defined, since it defines a unique output for each possible input. However, the specification

$$\varphi_2: \ s_2 \ = \ s_2 \wedge (t_1 \le 10)$$

is not well-defined, because there are many streams σ_2 that satisfy $\varphi_{2,\sigma}$ for some input stream. Similarly, the specification

$$\varphi_3: \ s_3 = \neg s_3$$

is not well-defined, but for this specification the reason is that it has no evaluation models.

To avoid ill-defined specifications we define a syntactic restriction on LOLA specifications guaranteeing that any well-formed LOLA expression is also well-defined.

Definition 3 (Dependency Graph) Let φ be a LOLA specification. A *dependency graph* for φ is a weighted and directed multi-graph $G = \langle V, E\rangle$, with vertex set $V = \{s_1,\ldots,s_n,t_1,\ldots,t_m\}$. An edge $e : \langle s_i, s_k, w\rangle$ labeled with a weight w is in E iff the equation for $\sigma_i(j)$ in φ_σ contains $\sigma_k(j+w)$ as a subexpression of the RHS, for some j (or $e : \langle s_i, t_k, w\rangle$ for subexpression $\tau_k(j+w)$). Intuitively, the edge records the fact that s_i at a particular position depends on the value of s_k, offset by w positions. Note that there can be multiple edges between s_i and s_k with different weights on each edge. Vertices labeled by t_i do not have outgoing edges.

Example 4 Consider the LOLA specification over independent integer variables t_1, t_2:

$$\begin{aligned} s_1 &= s_2[1,0] + \mathtt{ite}\begin{pmatrix} s_2[-1,7] \le t_1[1,0], \\ s_2[-1,0], \\ s_2 \end{pmatrix}. \\ s_2 &= (s_1 + t_2[-2,1]). \end{aligned}$$

Its dependency graph, shown in Figure 1, has three edges from s_1 to s_2, with weights 1, 0, -1, and one zero weighted edge from s_2 back to s_1. There is one edge from s_1 to t_1, and one from s_2 to t_2.

A *walk* of a graph is a sequence $v_1,\ldots,v_{k+1}$ of vertices, for $k \ge 1$, and edges $e_1,\ldots,e_k$, such that $e_i : \langle v_i, v_{i+1}, w_i\rangle$. The walk is closed iff $v_1 = v_{k+1}$. The total weight of a walk is the sum of weights of its edges.

Definition 4 (Well-Formed Specifications) A LOLA specification is *well-formed* if there is no closed-walk with total weight zero in its dependency graph.

Theorem 1 Every well-formed LOLA specification is well-defined.

All proofs will be available in an extended version of this document. The following alternative characterization of well-formedness is useful for algorithmic purposes and for the offline monitoring algorithm.

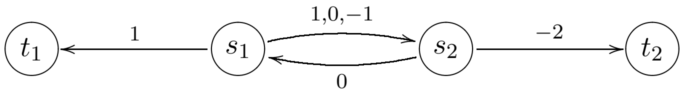


Fig. 1: Dependency graph for the specification of Example 4.

Theorem 2 A LOLA specification is well-formed iff no strongly connected component in G has both a positive and a negative weighted cycle.

The converse of Theorem 1 is not true: not every well-defined LOLA specification need be well-formed. For instance, the specification $s = s \wedge \neg s$ is well-defined, but not well-formed.

C. Statistics and Context-free Properties

We shall now demonstrate the use of our specification language for computing statistical properties over trace data. Numerical properties over traces are essential as (1) components of correctness properties that involve counts, maxima or minima over trace data, and (2) estimating performance and coverage metrics in the form of averages.

LOLA can be used to compute *incremental statistics*, i.e., measures that are defined using an update function $f_\alpha(v, u)$ where u represents the measure thus far, and v represents the new incoming data. Given a sequence of values $v_1, \ldots, v_n$, with a special default value d, the statistic over the data is defined in the *reverse sense* as

$$v = f_\alpha(v_1, f_\alpha(v_2, \ldots, f_\alpha(v_n, d)))$$

or in the *forward sense* as

$$v = f_\alpha(v_n, f_\alpha(v_{n-1}, \ldots, f_\alpha(v_1, d)))$$

Examples of such statistical measures include *count* with $f_{count}(v, u) = u+1$, *sum* with $f_{sum}(v, u) = v+u$, *max* with $f_{max}(u, v) = max(u, v)$, among many others; the statistical *average* can be incrementally defined as a pair consisting of the sum and the count.

Given an update function f_α and a data-stream v, the following LOLA queries compute the statistic in the forward and reverse senses respectively:

$$\begin{array}{rcl} stat_f &=& f_\alpha\left(stat_f[-1, d], v\right), \\ stat_r &=& f_\alpha\left(stat_r[1, d], v\right). \end{array}$$

For most common incremental statistical measures, either of these LOLA queries compute the same result. The choice of a monitoring strategy can dictate the use of one over another as will be evident in the subsequent section.

The use of numeric data also increases the expressiveness of the language; it enables the expression of context-free properties. Commonly encountered *context-free* properties include properties such as "*every request has a matching grant*." In programs, we may use such properties to verify that every lock acquired has been released, or that every memory cell allocated is eventually freed exactly once.

Example 5 Consider the property: *"the number of a's must always be no less than the number of b's."* This property can be expressed in LOLA as

$$\begin{array}{l} s = s[-1, 0] + \mathtt{ite}((a \ \wedge \ \neg b), 1, 0) \\ \quad\quad + \mathtt{ite}((b \ \wedge \ \neg a), -1, 0) \\ \mathbf{trigger}(s \leq 0) \end{array}$$

Integer streams in a LOLA specification enable the expression of context-free properties by being used as counters to model stacks. For instance, a two alphabet stack with alphabet symbols 0 and 1 can be modelled by a counter. Each pop is implemented by dividing the counter by 2, thereby eliminating the least significant bit. Each push is modelled by a multiplication by 2 followed by addition, thereby setting the least significant bit. Thus, with one (unbounded) counter, a LOLA specification can express context-free properties.

It can be shown that LOLA specifications with only boolean streams cannot express context-free properties.

III. Monitoring Algorithm

In this section, we first describe the setting for the monitoring problem considered in the paper. We then describe our monitoring algorithm using partial evaluation of the equational semantics.

A. Monitoring Setup

We distinguish two situations for monitoring — *online* and *offline* monitoring. With online monitoring, system behaviors are observed as the system is run under a test/real-life setting. In a simulation setting, we can assume that the monitor is working in tandem with the simulator, with the monitor processing a few trace positions while the simulator waits, and then the monitor waiting while the simulation proceeds to produce the next few positions. On the other hand, offline monitoring assumes that the system has been run to completion, and the trace data was dumped to a storage device. This leads to the following restriction for online monitoring: the traces are available a few points at a time starting from time 0 onwards, and need to be processed online to make way for more incoming data. In particular, random access to the traces is not available.

B. Online Monitoring Algorithm

In online monitoring we assume that the trace is available one position at a time, starting from time 0. The length of the trace is assumed to be unknown and large.

Let $t_1, \ldots, t_m$ be independent (input) stream variables, and $s_1, \ldots, s_n$ be dependent (output) stream variables. Let $j \geq 0$ be the current position where the latest trace data is available from all the input streams.

Evaluation Algorithm: The evaluation algorithm maintains two stores of equations:

- *Resolved* equations R of the form $\sigma_i(j) = c$, or $\tau_i(j) = c$, for constant c.
- *Unresolved* equations U of the form $\sigma_i(j) = e_i$ for all other stream expressions e_i.

Initially both stores are empty. At the arrival of input stream data for a particular position j, $0 \leq j \leq N$, that is, when $\tau_1(j), \ldots, \tau_m(j)$ become available, the following steps are carried out:

1) The equations $\tau_1(j) = c_1, \ldots, \tau_m(j) = c_m$ are added to R,
2) The associated equations for $\sigma_1(j), \ldots, \sigma_n(j)$ are added to U,
3) The equations in U are simplified as much as possible; if an equation becomes of the form $\sigma_i(j) = c$, it is removed from U and added to R. If c is $true$ and the corresponding output variable s_i is marked as a trigger, then a violation is reported.
4) For each stream t_i(also s_i), there is a non-negative constant k_i such that $\tau_i(j - k_i)$, if present in R can be safely removed. The constant $k_i \geq 0$ is defined as
$$k_i = \max \left\{ k \;\middle|\; \begin{array}{l} k \text{ is non-negative and} \\ t_i[-k, d] \text{ is a subexpression.} \end{array} \right\}.$$
Intuitively, for any position j, $j + k_i$ is the latest value in the future whose computation requires the value of $\tau_i(j)$.

Example 6 To illustrate the last point, consider the specification,

$$s = s[-3, 0] + t.$$

Let τ be the input stream. The value of k_i for s is 3 and for t is zero. This indicates that for any input stream τ, the value $\tau(j)$ can be removed from R at position j itself. Similarly any $\sigma(j) \in R$ may be removed from R at (or after) position $j + 3$.

Equations in U are simplified using the following rules:

1) Partial evaluation rules for function applications such as,
$$\mathit{true} \wedge e \;\rightarrow\; e, \;\; 0 + x \rightarrow x \;\; \cdots$$
2) Rewrite rules for if-then,
$$\mathtt{ite}(\mathit{true}, e_1, e_2) \;\rightarrow\; e_1 \;\; \cdots$$
3) Substitution of resolved positions from R. If $\sigma_i(j) = c \in R$, then every occurrence of $\sigma_i(j)$ in U is substituted by c and possibly simplified further.

We illustrate the operation of the algorithm on a simple example.

Example 7 Let t_1, t_2 be two input boolean stream variables. Consider the specification

$$\varphi : \; s \; = \; t_2 \vee (t_1 \wedge s[1, \mathbf{false}]),$$

which computes t_1 **Until** t_2. The associated equations for φ are:

$$\sigma(j) = \begin{cases} \tau_2(j) \vee (\tau_1(j) \wedge \sigma(j+1)) & j + 1 \leq N \\ \tau_2(j) & \mathit{otherwise}. \end{cases}$$

Let the input streams, τ_1 and τ_2 be given by

τ_1	*false*	*false*	*true*	*true*	*true*	*true*	*true*
τ_2	*true*	*false*	*false*	*false*	*false*	*false*	*false*

At position 0, we encounter $\langle \mathit{false}, \mathit{true} \rangle$. The equation for $\sigma(0)$ is

$$\begin{array}{rcl} \sigma(0) & = & \tau_2(0) \vee (\tau_1(0) \wedge \sigma(1)) \\ & \rightarrow & \mathit{true} \vee (\mathit{false} \wedge \sigma(1)) \\ & \rightarrow & \mathit{true} \end{array}$$

and thus $\sigma(0) = \mathit{true}$ is added to the resolved store R. At position 1, we encounter $\langle \mathit{false}, \mathit{false} \rangle$ and thus we can set $\sigma(1) = \mathit{false}$, which is also added to R. From $j = 2$ until $j = 5$, we encounter $\langle \mathit{true}, \mathit{false} \rangle$. At each of these positions the equations $\sigma(j) = \sigma(j + 1)$ are added to U. The equation store U now has the equations

$$\sigma(2) = \sigma(3), \; \sigma(3) = \sigma(4), \ldots, \; \sigma(5) = \sigma(6).$$

At position 6, we encounter $\langle true, false\rangle$ with the added information that the trace has ended. We set $\sigma(6) = false$ and add it to R. This lets us resolve the equations in U and set all the positions from 2 to 6 to $false$.

Note that the equation associated with $\sigma_i(j)$ on the LHS is added only after the current position reaches j, even if the term $\sigma_i(j)$ appears on the RHS of some equation before position j is reached.

The algorithm above works in time and space that is linear in the length of the trace and the size of the specification. Since the memory usage can be as large as the length of the trace in the worst-case, the method may not work for long simulations and large traces.

Example 8 Consider the following LOLA specification:

$$\begin{array}{rcl} ended & = & \mathbf{false}[1, \mathbf{true}] \\ s & = & \mathtt{ite}(ended, t, s[1, \mathbf{true}]) \end{array}$$

in which the output stream σ takes the same value everywhere that the input stream τ takes at the end of the trace. The partial evaluation algorithm maintains the unresolved $\sigma(0), \ldots, \sigma(N)$. Such specifications cannot be monitored efficiently. Furthermore, if the variable s appears in other expressions, the evaluation of the corresponding streams need to be delayed until σ can be resolved.

In the next section we characterize an *efficiently monitorable* set of LOLA specifications based on the properties of their dependency graphs. The partial evaluation algorithm will be shown to work efficiently for such specifications.

C. Efficiently Monitorable Specifications

We present a class of specifications that are efficiently monitorable. These specifications are guaranteed to limit the number of unresolved equations in the memory to a pre-determined constant that depends only on the size of the specification and *not* on the size of the trace.

Definition 5 (Efficiently Monitorable Specifications) A LOLA specification is *efficiently monitorable* (EM) if its worst case memory requirement under our online monitoring algorithm is constant in the size of the trace.

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

$$\begin{array}{l} reqgrant \ = \ \mathtt{ite}(request, evgrant, \mathbf{true}) \\ evgrant \ = \ grant \ \vee \ evgrant[1, \mathbf{false}] \\ \mathbf{trigger}\ (\neg\ reqgrant) \end{array}$$

The specification encodes the temporal assertion $\Box(request \rightarrow \Diamond(grant))$. Another way that produces the same result is

$$waitgrant \ = \ \left(\neg grant \wedge \left(\begin{array}{l} request \ \vee \\ waitgrant[-1, \mathbf{false}] \end{array} \right) \right)$$
$$\mathbf{trigger}\ ended \ \wedge \ waitgrant$$

The stream $waitgrant$ records if the monitor is currently waiting for a grant. The monitor waits for a grant whenever it encounters a $request$ and stops waiting if there is a $grant$. If the trace ends while the monitor is still waiting, it triggers an error. The latter formulation is efficiently monitorable, while the former is not. For instance, at every time instance, $waitgrant(i)$ is instantly resolved given its previous value, and those of the input streams. Thus, the simple partial evaluation algorithm monitors the latter with very little, constant, buffering.

The following theorem characterizes efficiently monitorable LOLA specifications.

Theorem 3 If the dependency graph of a LOLA query has no positive cycles then it is efficiently monitorable.

The converse of the theorem above does not hold in general. However, in the absence of an alternative syntactic characterization of EM specification, we shall henceforth use the term EM specification to denote queries whose dependency graphs do not contain positive cycles.

Given graph G, that does not have any positive weight cycles, we construct a graph G^+, obtained by removing all negative weight edges from G. Furthermore, among all the edges in G between two nodes s_i and s_j, we choose to add only that edge to G^+ which has the maximum positive weight. The graph G^+ has no self loops or multiple edges, and hence is a weighted directed acyclic graph (DAG). For each node $s_i \in G^+$, we define Δ_i as follows:

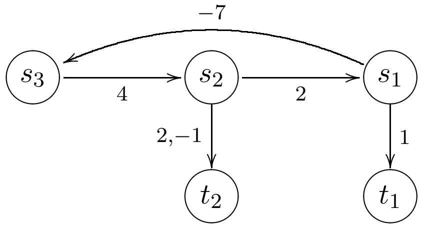
$$\Delta_i = \begin{cases} 0, & \textit{if there is no outgoing edge from } s_i, \\ \max\left\{ \Delta_j + w(e_j) \;\middle|\; \begin{array}{l} e_j : s_i \xrightarrow{w(e_j)} s_j \\ \textit{is an edge in } G^+ \end{array} \right\}, & ow. \end{cases}$$

Example 10 Consider the following LOLA specification:

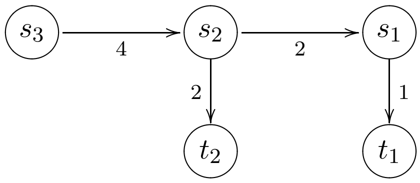
$$\begin{array}{rcl} s_1 & = & t_1[1, \mathbf{false}] \ \wedge \ s_3[-7, \mathbf{false}] \\ s_2 & = & \mathtt{ite}(s_1[2, \mathbf{true}], t_2[2, 0], t_2[-1, 2]) \\ s_3 & = & (s_2[4, \mathbf{true}] \leq 5) \end{array}$$

The dependency graph G is shown in Figure 2. The values of the Δ function are as follows:

$$\Delta(t_1) = \Delta(t_2) = 0, \ \Delta(s_1) = 1, \ \Delta(s_2) = 3, \ \Delta(s_3) = 7.$$



(a) Dependency graph G.



(b) Derived graph G^+.

Fig. 2: The dependency graph G for Example 10 and its derived graph G^+.

The significance of the Δ function is clear through the following theorem.

Theorem 4 The partial evaluation algorithm resolves any trace position $\sigma_i(j)$ before time $j + \Delta_i$.

The memory requirement is therefore *constant* in N for an efficient specification. This number of unresolved positions in U is upper-bounded by $O(\Delta_1 + \cdots + \Delta_n)$. For instance, computing the Δ values for the queries in Example 9, we find that $\Delta(waitgrant) = 0$. This shows that the value of $waitgrant$ resolves immediately, given its previous value and the inputs. Our experimental results in the subsequent section show that requiring specifications to be efficiently monitorable is not unreasonable in practice. Furthermore, streams involved in positive cycles can be discarded or even rewritten (as shown in Example 9) for the purposes of online monitoring.

The framework developed generalizes naturally to an offline monitoring algorithm. Please refer to the full version of this paper available online.

IV. Applications

There are numerous applications of this formalism. In this section, we describe two such applications obtained directly from the industry. Synopsys, Inc. provided some circuit simulation dumps, along with specifications written in the industry standard *System Verilog Assertions* (SVA)[18]. We were able to hand-translate the SVA queries directly into LOLA specifications, a process that is potentially mechanizable.

Our OCAML-based implementation of LOLA reads a trace file and the specification file. It implements the online monitoring algorithm described in Section III with some direct optimizations. We have incorporated facilities for displaying dependency graphs of specifications.

The following two case studies were considered:

a) Memory Controller: A Verilog model for a memory controller was simulated yielding 13 input streams. The corresponding SVA assertions were hand-translated into a LOLA specification. The specification had 21 intermediate streams and 15 output streams, all of which were declared triggers. Properties enforced included mutual exclusion of signals, correct transfers of address and data, and timing specifications (e.g. signal stability for 3 or 4 cycles). The specifications were not EM: the dependency graph had three positive-sum cycles, each encoding a temporal until operator. Figure 3 shows the performance of LOLA on these traces.

b) PCI: We hand translated SVA assertions describing the PCI 2.2 specifications for the master. A circuit implementing the master was simulated for varying times to produce a set of traces to plot the performance. The specification had 15 input streams, 161 output streams and 87 trigger streams. Our initial implementation contained three positive weight cycles. We were able to remove these by rewriting the queries carefully. Running times can also be found in Figure 3. Bugs were deliberately introduced into the circuit in order to evaluate the effectiveness of runtime verification. LOLA reports numerous useful trigger violations for the longest trace.

V. Conclusions

We have presented LOLA, a formalism for runtime verification based on a functional language over finite streams equipped with a partial evaluation-based strategy for online evaluation. Our formalism combines runtime verification of boolean temporal specifications with statistical measures to estimate coverage and specify complex temporal patterns. By evaluating our system on industrial strength specifications, we have demonstrated that LOLA can express relevant properties. Using dependency graphs, we have characterized efficiently monitorable queries that can be monitored online efficiently in terms of space. Based on our case-studies so far, the restriction to efficiently monitorable specifications seems

	Controller example		PCI example	
# simulation steps	# clock pos. edges	time (sec)	# clock pos. edges	time
5000	250	0.18	834	4.62
10000	500	0.35	1667	8.87
20000	1000	0.71	3334	19.04
50000	2500	1.78	8334	29.47
100000	5000	3.47	16667	52.53
200000	10000	6.83	33334	99.17
500000	25000	17.02	83334	236.96
1000000	50000	33.70	166667	467.98

Fig. 3: Running times for both examples. All timings were measured on an Intel Xeon Processor running Linux 2.4 with 2Gb RAM.

practical.

In the future, we intend to study automatic techniques for rewriting non-EM specifications into efficiently monitorable ones where possible, and in further collaboration with industry study the applicability of these techniques for larger case studies. We expect that for such use some syntactic sugar needs to be added to LOLA to facilitate specification of common constructs. Also the error reporting needs to be improved by synthesizing explanations for each violation. Extensions to handle synchronous systems with many clocks, asynchronous systems, and distributed systems are also under consideration.

REFERENCES

[1] K. Havelund and G. Roşu, Eds., *Runtime Verification 2001 (RV'01)*, ser. ENTCS, vol. 55. Elsevier, 2001.

[2] ——, *Runtime Verification 2002 (RV'02)*, ser. ENTCS, vol. 70, no. 4. Elsevier, 2001.

[3] O. Sokolsky and M. Viswanathan, Eds., *Runtime Verification 2002 (RV'03)*, ser. ENTCS, vol. 89, no. 2. Elsevier, 2003.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data-flow programming language LUSTRE," *Proc. of IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[5] G. Berry, *Proof, language, and interaction: essays in honour of Robin Milner*. MIT Press, 2000, ch. The foundations of Esterel, pp. 425–454.

[6] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "Runtime Assurance Based on Formal Specifications," in *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[7] D. Drusinsky, "The temporal rover and the ATG rover," in *SPIN Model Cheking and Software Verification*, 2000, pp. 323–330.

[8] K. Havelund and G. Roşu, "Synthesizing monitors for safety properties," in *Proc. of TACAS'02*. Springer, 2002, pp. 342–356.

[9] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. New York: Springer, 1995.

[10] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," *Formal Methods in System Design*, vol. 19, no. 3, pp. 291–314, 2001.

[11] K. Havelund and G. Roşu, "An overview of the runtime verification tool java pathexplorer," *Formal Methods for Systems Design*, vol. 24, no. 2, pp. 189–215, 2004.

[12] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout, "Reasoning with temporal logic on truncated paths," in *Proc. of CAV'03*, ser. LNCS, vol. 2725. Springer, 2003, pp. 27–39.

[13] B. Finkbeiner, S. Sankaranarayanan, and H. B. Sipma, "Collecting statistics over runtime executions," in [2].

[14] T. Gautier, P. Le Guernic, and L. Besnard, "SIGNAL: A declarative language for synchronous programming of real-time systems," in *Proc. Conference on Functional Programming Languages and Computer Architecture*. Springer, 1987, pp. 257–277.

[15] K. Sen and G. Roşu, "Generating optimal monitors for extended regular expressions," in [3].

[16] G. Roşu and K. Havelund, "Rewriting-based techniques for runtime verification," Journal of Automated Software Engineering (to appear).

[17] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *Proc. of 5th International Conference VMCAI'04*, ser. LNCS, vol. 2937. Springer, 2004, pp. 44–57.

[18] "System verilog assertion homepage," 2003, [Online] Available: http://www.eda.org/sv-ac.