Decentralized Stream Runtime Verification*

Luis Miguel Danielsson[1,2] and César Sánchez[1]

[1] IMDEA Software Institute, Spain
[2] Universidad Politécnica de Madrid (UPM), Spain
{luismiguel.danielsson,cesar.sanchez}@imdea.org

Abstract. We study the problem of decentralized monitoring of stream runtime verification specifications. Decentralized monitoring uses distributed monitors that communicate via a synchronous network, a communication setting common in many cyber-physical systems like automotive CPSs. Previous approaches to decentralized monitoring were restricted to logics like LTL logics that provide Boolean verdicts. We solve here the decentralized monitoring problem for the more general setting of stream runtime verification. Additionally, our solution handles network topologies while previous decentralize monitoring works assumed that every pair of nodes can communicate directly. We also introduce a novel property on specifications, called decentralized efficient monitorability, that guarantees that the online monitoring can be performed with bounded resources. Finally, we report the results of an empirical evaluation of an implementation and compare the expressive power and efficiency against state-of-the-art decentralized monitoring tools like Themis.

1 Introduction

We study the problem of decentralized runtime verification of stream runtime verification (SRV) specifications. Runtime verification (RV) is a dynamic technique for software quality assurance that consists of generating a monitor from a formal specification, that then inspects a single trace of execution of the system under analysis. One of the problems that RV must handle is to generate monitors from a specification. Early approaches for specification languages were based on temporal logics [6, 11, 18], regular expressions [25], timed regular expressions [2], rules [3], or rewriting [23]. Stream runtime verification, pioneered by Lola [10], defines monitors by declaring the dependencies between output streams of results and input streams of observations. SRV is a richer formalism that goes beyond Boolean verdicts, like in logical techniques, to allow specifying the collection of statistics and the generation richer (non-Boolean) verdicts. Examples include counting events, specifying robustness or generating models or

* This work was funded in part by the Madrid Regional Government under project "S2018/TCS-4339 (BLOQUES-CM)", by EU H2020 project 731535 "Elastest" and by Spanish National Project "BOSCO (PGC2018-102210-B-100)".

quantitative verdicts. See [10,14,17] for examples illustrating the expressivity of SRV languages.

Another important aspect of RV is the operational execution of monitors: how to collect information and how to monitor incrementally. In this paper we consider using a network of distributed monitors connected via a synchronous network, together with periodic sampling of inputs. This problem is known as *decentralized monitoring*. Our goal is to generate local monitors at each node that collaborate to monitor the specification, distributing the computational load while minimizing the network bandwidth and the latency of the computation of verdicts. Apart from more efficient evaluation, decentralized monitoring can provide fault-tolerance as the process can partially evaluate a specification with the part of the network that does not fail.

Our Solution. In this paper we provide a solution to the decentralized monitoring problem for Lola specifications for arbitrary network topologies and placement of the local monitors. We assume a connected network topology where nodes can only communicate directly with their neighbors. In general, messages between two nodes require several hops, and all nodes have initially deployed a local routing table that contains the next hop depending on the final destination. We follow the synchronous distributed model of computation, where computation (input readings from the system, message routing and local monitor computations) proceeds in rounds. We also assume a reliable system: nodes do not crash, and messages are not lost or duplicated. These assumptions are realistic, for example in automotive CPSs like solutions based on a synchronous BUS, like CAN networks [19] and Autosar [1]. In our solution, different parts of the specification (modeled as streams), including input readings, will be deployed in different nodes as a local monitor. Local monitors will communicate when necessary to resolve the streams assigned to them, trying to minimize the communication overhead.

A degenerated case of this setting is a centralized solution: nodes with mapped observations send their sensed values to a fixed central node that is responsible of computing the whole specification. The SRV language that we consider is Lola [10] including future dependencies. We will identify those specifications that can be monitored decentralized with finite memory, independently of the trace length.

Motivating Example. Our motivation is to build rich monitors for decentralized synchronous networks, used in automotive CPSs [20] for example using the Autosar standard over the CAN network. This example is inspired by the Electronic Stability Program (ESP) and models the under steering to the left scenario in an attempt to avoid an obstacle. The ESP must detect the sudden turn of the steering wheel and the deviation produced with the actual movement of the car (yaw). When this deviation is perceived and

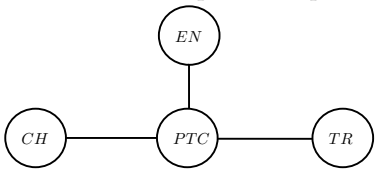

Fig. 1. Autosar simplified topology

maintained over a period of time, the ESP must act on the brakes, the torque distribution and the engine in order to produce the desired movement without any lose of control over the vehicle. The topology is shown on the right. The monitor in node *CH* (chassis) detects the under-steering and whether the ESP must be activated. The monitor in *EN* (engine) checks that the throttle is ready to react to the evasive maneuver. Our intention is to define the following streams:

- `ESP_on`: represents whether there is under-steering or the wheel is slipping,
- `req_thr`: the requested throttle,
- `good_thro`: whether the throttle is correct

We achieve this by using input values, like `yaw` (the direction of the wheels), the desired `steering`, `drive_wheel_slip` (whether the drive wheels are slipping), and the `throttle`.

```
@Chassis{ input num yaw, steering, drive_wheel_slip
  define num dev = steering - yaw
  define bool under_steering_l = dev > 0 and dev > 0.2
  output bool ESP_on = under_steering_left or drive_wheel_slip
}
@Engine{ input num throttle
  output num req_thr = if ESP_on then req_thr[-1|0]-dev else 0
  output bool good_thro = req_thr[-1|0]/throttle <= 0.1
}
```

Related work. The work in [4] shows how monitoring Metric Temporal Logic specifications of distributed systems (including failures and message reordering) where the nodes communicate in a tree fashion and the root emits the final verdict. Sen et al. [26] introduces PT-DTL, a variant of LTL logic for monitoring distributed systems. The work in [16] uses slices to support node crashes and message errors when monitoring distributed message passing systems with a global clock. Bauer et al. [5] introduce a first-order temporal logic and trace-length independent spawning automaton. Bauer et al. [7] shows a decentralized solution to monitor LTL_3 in synchronous systems using formula rewriting. This is improved in [12, 13] using an Execution History Encoding (EHE). All these approaches consider only Boolean verdicts. SRV can generate verdicts from arbitrary data domains, but all previous SRV efforts, from Lola [10], Lola2.0 [14], Copilot [21, 22] and extensions to timed event streams, like TeSSLa [8], RT-Lola [15] or Striver [17] assume a centralized monitoring setting.

Contributions and structure. The main contribution of this paper is a solution, described in Section 3, to the decentralized stream runtime verification problem. A second contribution, included in Section 4, is the identification of a fragment of specifications, called decentralized efficiently monitorable, that ensure that monitoring can be performed with bounded memory (independently of the length of the input trace). A third contribution, detailed in Section 5, is a prototype implementation and an empirical evaluation. Section 2 contains the preliminaries and Section 6 concludes.

2 Preliminaries. Stream Runtime Verification

We recall now SRV briefly (see [10] and the tutorial [24]). The fundamental idea of SRV, pioneered by Lola [10] is to cleanly separate the temporal dependencies from the individual operations to be performed at each step, which leads to generalization of monitoring algorithms for logics to the computation of richer values. A Lola specification declares the relation between output streams and input streams, including both future and past temporal dependencies. The streams are typed using arbitrary multi-sorted first-order theories. A type has a collection of symbols used to construct expressions, together with an interpretation of these symbols to evaluate ground expressions.

Lola Syntax. Given a set Z of (typed) stream variables the set of *stream expressions* consists of (1) variables from Z, (2) offsets $v[k, d]$ where v is a stream variable of type D, k is a natural number and d a value from D, and (3) function applications $f(t_1, \ldots, t_n)$ using constructors f from the theories to previously defined terms. Stream variables represent streams (sequences of values). The intended meaning of expression $v[-1, \mathit{false}]$ is the value of stream v in the previous position of the trace (or *false* if there is no such previous position, that is, at the beginning). We assume that all theories have a constructor `if` $\cdot$ `then` $\cdot$ `else` $\cdot$ that given an expression of type Bool and two expressions of type D constructs a term of type D. We use $\mathit{Term}_D(Z)$ for the set of stream expressions of type D constructed from variables from Z (and drop Z if clear from the context). Given a term t, $\mathit{sub}(t)$ represents the set of sub-terms of t.

Definition 1 (Specification). *A Lola specification $\varphi(I, O)$ consists of a set $I = \{r_1, \ldots, r_m\}$ of input stream variables, a set $O = \{s_1, \ldots, s_n\}$ of output stream variables, and a set of defining equations, $s_i = e_i(r_1, \ldots, r_m, s_1, \ldots, s_n)$ one per output variable $s_i \in O$, from $\mathit{Term}_D(I \cup O)$ where D is the type of s_i.*

A specification describes the relation between input streams and output streams. We will use r, r_i,... to refer to input stream variables; s, s_i, ... to refer to output stream variables; and u, v for an arbitrary input or output stream variable. Given $\varphi(I, O)$ we use $\mathit{appears}(u)$ for the set of output streams that use u, that is $\{s_i \mid u[-k, d] \in \mathit{sub}(e_i)$ or $u \in \mathit{sub}(e_i)\}$. Also, $\mathit{ground}(t)$ indicates whether expression t is a ground (contains no variables or offsets) and can be evaluated into a value.

Example 1. The property "*sum the previous values in input stream y, but if the reset stream is true, reset the count*", can be expressed as follows, where stream variable `root` uses the accumulator `acc` and the input `reset` to compute the desired sum:

```
input  bool reset, int i
define int acc  = i + root[-1|0]
output int root = if reset then 0 else acc
```

Lola semantics. At runtime, input stream variables are associated with input streams (sequence of values of the appropriate type and of the same length M). The intended meaning of a Lola specification is to associate output streams to output stream variables (of the same length M) that satisfy the equations in the specification. Formally, this semantics are defined denotationally. Given input streams σ_I (one sequence per input stream variable) and given an output candidate σ_O (one sequence per output stream) the semantics describe when the pair (σ_I, σ_O) matches the specification, which we write $(\sigma_I, \sigma_O) \vDash \varphi$. We use σ_r for the stream in σ_I corresponding to input variable r and $\sigma_r(k)$ for the value at position k (with $0 \leq k \leq M$).

A *valuation* is a pair $\sigma : (\sigma_I, \sigma_O)$. Given a valuation the *evaluation* $[\![t]\!]_\sigma$ of a term t is a sequence of length M of values of the type of t defined as follows:

- If t is a stream variable u, then $[\![u]\!]_\sigma(j) = \sigma_u(j)$.
- If t is $f(t_1, \ldots, t_k)$ then $[\![f(t_1, \ldots, t_k)]\!]_\sigma(j) = f([\![t_1]\!]_\sigma(j), \ldots, [\![t_k]\!]_\sigma(j))$
- If t is $v[i, c]$ then $[\![v[i, c]]\!]_\sigma(j) = [\![v]\!]_\sigma(j+i)$ if $0 \leq j+i < M$, and c otherwise.

Definition 2 (Evaluation Model). *A valuation (σ_I, σ_O) satisfies a Lola specification φ whenever for every output variable s_i, $[\![s_i]\!]_{(\sigma_I, \sigma_O)} = [\![e_i]\!]_{(\sigma_I, \sigma_O)}$. In this case we say that σ is an evaluation model of φ and write $(\sigma_I, \sigma_O) \vDash \varphi$.*

This semantics capture when a candidate valuation is an evaluation model, but the intention of a Lola specification is to compute the unique output streams given input streams.

A *dependency graph* D_φ of a specification $\varphi(I \cup O)$ is a weighted multi-graph (V, E) whose vertices are the stream variables $V = I \cup O$, and E contains a directed weighted edge $u \xrightarrow{w} v$ whenever $v[w, d]$ is a subterm in the defining equation of u. If a dependency graph D_φ contains no cycles with 0 weight then the specification is called well-formed, and it guarantees that for every σ_I there is a unique σ_O such that $(\sigma_I, \sigma_O) \vDash \varphi$. This is because the value of a stream at a given position cannot depend on itself.

Given a stream variable u and position $i \geq 0$ the *instant stream variable* (or simply instant variable) $u[i]$ is a fresh variable of the same type as u.

Definition 3 (Evaluation Graph). *Given $\varphi(I, O)$ and a trace length M the evaluation graph $G_{\varphi,M}$ has as vertices the set of instant variables $\{u[k]\}$ for $u \in I \cup O$ and $0 \leq k < M$, and has edges $u[k] \to v[k']$ if the dependency graph contains an edge $u \xrightarrow{j} v$ and $k + j = k'$*

For example, if the defining equations of u contains $v[-1, d]$ then $u[16]$ points to $v[15]$ in all evaluation graphs with $M \geq 16$. In well-formed specifications there are no cycles in any evaluation graph, which enables to reason by induction on evaluation graphs.

Lola Online Monitoring. The Lola online monitoring algorithm [10,24] maintains two storages:

- R: for instant variables that have been resolved (that is, pairs $(u[k], c)$ that denote that $u[k]$ is known to have value c);

– U: for instant variables $u[k]$ whose value is not determined yet (that is, whose instantiated equation still contains variables).

At instant k the equation e_i for s_i gets instantiated as follows: every variable u in e_i is converted into $u[k]$ and every offset $u[j, d]$ is turned into $u[k + j]$ (or into d if $k + j$ falls out of bounds). After instantiating all equations, the monitor substitutes instant variables by their value if these values are resolved (in R). If the resulting equation is not ground then it remains in U. Eventually, all values will be discovered and every term will be resolved and moved from U to R.

3 Decentralized Stream Runtime Verification

In this section we describe our solution to the decentralized SRV problem. Given a well-formed Lola specification, the decentralized online algorithm that we present here will incrementally compute a value for each output instant variable, reading values from the input stream variables at every clock instant. The starting point of the solution is a map that associates each variable in the specification to a network node. The node associated to an input variable corresponds to the location where readings of new values are performed, and the node associated to an output variable s is the node responsible to incrementally compute the stream for s. Each node will run a local monitor, that will collaborate with other monitors by exchanging messages to perform the global monitoring task.

The main correctness criteria is that the output produced by our network of cooperating monitors corresponds to the denotational semantics. However, the decentralized algorithm may compute some output values at different time instants than the centralized version, due to the different location of the inputs and the delays caused by the communication.

3.1 Problem Description

The description of the decentralized SRV monitoring problem consists of a specification, a network topology and a stream assignment.

Network. A network topology $\mathcal{T} : \langle N, \rightarrow \rangle$ is given by a set of nodes N connected by directed edges $\rightarrow \subseteq N \times N$ that represent communication links between the nodes. We assume that the graph is connected. A route between two nodes n and m is list of nodes $[n_0, \ldots, n_k]$ such that consecutive nodes are neighbors (i.e. $n_i \rightarrow n_{i+1}$), no node is repeated, and $n = n_0$ and $m = n_k$. We statically fix routes between every two nodes with the following properties: (1) if two nodes n, m are neighbors then they communicate directly (that is $[n, m]$ is the route from n to m); (2) if $[n_0, \ldots, n_i, \ldots n_k]$ is a route, then the route from n_i to n_k is the sub-list $[n_i \ldots n_k]$. These properties can be enforced easily in a connected graph, and they imply that routing tables can be encoded locally in every node by just encoding at every node the next hop for every destination.

We use $next_n(m)$ for the next hop in the routing table of n for messages with destination m, and $dist(n, m)$ for the number of hops between n and m. This is

precisely the number of routing operations that are needed for a message from n to arrive to m. Consider the topology in Fig. 1. A message inserted at time 17 in *CH* with destination *EN* will arrive to *PTC* at time 18, which will be routed, arriving at *EN* at time 19.

We assume reliable unicast communication (no message loss or duplication) over a synchronous network, from which we build a synchronous distributed system where computation proceeds as a sequence of cycles. In this computational model, all nodes in the network execute in every cycle—in parallel and to completion— the following actions: (1) read input messages, (2) perform a terminating local computation, (3) generate output messages. We describe below our decentralized monitoring solution as a synchronous distributed system. In our solution we use two types of messages:

- **Requests** messages: $(\mathbf{req}, s[k], n_s, n_d)$ where $s[k]$ is an instant variable, n_s is the source node and n_d is the destination node of the message.
- **Response** messages: $(\mathbf{resp}, s[k], c, n_s, n_d)$ where $s[k]$ is an instant variable, c is a value, n_s is the source node and n_d is the destination node.

Let $msg = (\mathbf{resp}, s[k], c, n_s, n_d)$, then $msg.src = n_s$, $msg.dst = n_d$, $msg.type = \mathbf{resp}$, $msg.stream = s[k]$ and $msg.val = c$ (the analogous definitions apply for a request message except that $msg.val$ is not applicable). The intention of request messages is that n_s requests the value of $s[k]$ from n_d, which is the node in charge of stream s. Response messages are used to inform of the actual values read or computed.

Stream Assignment and Communication Strategies. Given a specification $\varphi(I, O)$ and a network topology $\mathcal{T} : \langle N, \rightarrow \rangle$ a *stream assignment* is a map $\mu : I \cup O \rightarrow N$ that assigns a network node to each stream variable. The node $\mu(r)$ for an input stream variable r is the location in the network where r is sensed in every clock tick. At runtime, at every instant k new input values for variables mapped to different nodes are read simultaneously. The node $\mu(s)$ for an output stream variable s is the location whose local monitor is responsible for resolving the values of s.

Additionally, each stream variable v can be assigned one of the following two *communication strategies* to denote whether an instant value $v[k]$ is automatically communicated to all potentially interested nodes, or whether its value is obtained on request only. Let v and u be two stream variables such that v appears in the equation of u and let $n_v = \mu(v)$ and $n_u = \mu(u)$.

- **Eager communication**: the node n_v informs n_u of every value $v[k] = c$ that it resolves by sending a message $(\mathbf{resp}, v[k], c, n_v, n_u)$.
- **Lazy communication**: node n_u requests n_v the value of $v[k]$ (in case n_u needs it to resolve $u[k']$ for some k') by sending a message $(\mathbf{req}, v[k], n_u, n_v)$. When n_u receives this message and resolves $v[k]$ to a value c, n_u will respond with $(\mathbf{resp}, v[k], c, n_v, n_u)$.

Each stream variable can be independently declared as eager or lazy. We use two predicates $eager(u)$ and $lazy(u)$ (which is defined as $\neg eager(u)$) to indicate the communication strategy of stream variable u. Note that the lazy strategy involves two messages and eager only one, but eager sends every instant variable

resolved, while lazy will only sends those that are requested. In case the values are almost always needed, eager is preferable while if values are less frequently required lazy is preferred. We are finally ready to define the decentralized SRV problem.

Definition 4. *A decentralized SRV problem* $\langle \varphi, \mathcal{T}, \mu, eager \rangle$ *is characterized by a specification* φ*, a topology* $\mathcal{T}$*, a stream assignment* μ *and a communication strategy for every stream variable.*

3.2 Decentralized Stream Runtime Verification

Our solution consists of a collection of local monitors, one for each network node n. A local monitor $\langle Q_n, U_n, R_n, P_n, W_n \rangle$ for n consists of an input queue Q_n and four storages:

- **Resolved** storage R_n, where n stores resolved instant variables $(v[k], c)$.
- **Unresolved** storage U_n, where n stores unresolved equations $v[k] = e$ where e is not a value, but an expression that contains other instant variables.
- **Pending** requests P_n, where n records instant variables that have been requested from n by other monitors but that n has not resolved yet.
- **Waiting** for responses W_n, where n records instant variables that n has requested from other nodes but has received no response yet.

When n receives a response from remote nodes, the information is added to R_n, so future local requests for the same value can be resolved immediately. The storage W_n is used to prevent n from requesting the same value twice while waiting for the first request to be responded. An entry in W_n is removed when the value is received, since the value will be subsequently fetched directly from R_n and not requested through the network. The storage P_n is used to record that a value that n is responsible for has been requested, but n does not know the answer yet. When n computes the answer, then n sends the corresponding response message and removes the entry from P_n.

Informally, in each cycle, the local monitor for n processes the incoming messages from its input queue Q_n. Then n reads the values for input streams assigned to it and also instantiates for the current instant the output stream variables that n is responsible for. After that, the equations obtained are simplified using the knowledge acquired so far by n. Finally, new response and request messages are generated and inserted in the queues of the corresponding neighbors.

More concretely, every node n will execute the procedure Monitor shown in Algorithm 1, which invokes Step in every clock tick until the input terminates. The procedure Finalize is used to resolve the pending values at the end of the trace to their default. The procedure Step executes the following steps:

1. **Process Messages**: Lines 11-20 deal with the processing of incoming messages. First, Lines 13-14 route messages with a different destination. Lines 16-17 annotate requests in P, which will be later resolved and responded. Lines 19-20 handle response arrivals, adding them to R and removing them from W.

2. **Read New Inputs and Outputs:** Line 21 reads new inputs for current time k, and line 22 instantiates the equation of every output stream that n is responsible for.
3. **Evaluate:** Line 23 evaluates the unresolved equations using EVALUATE.
4. **Send Responses:** Lines 24-27 send messages for all eager variables. Lines 28-31 deal with pending lazy variables. If a pending instant variable is now resolved, the response message is sent and the entry is removed from P_n.
5. **Send new Requests:** Lines 32-35 send new request messages for all lazy instant streams that are now needed, to the corresponding responsible nodes.
6. **Prune:** Line 37 prunes the set R from information that is no longer needed.

The pruning algorithm appears in Algorithm 2 and it is described in Section 4. We now show that our solution is correct by proving that the output computed is the same as in the denotational semantics, and that every output is eventually computed.

Theorem 1. *All of the following hold for every instant variable $u[k]$:*
(1) *If $lazy(u)$ then all request messages for $u[k]$ are eventually responded.*
(2) *If $eager(u)$ then a response message for $u[k]$ is eventually sent.*
(3) *The value of $u[k]$ is eventually resolved.*
(4) *The value of $u[k]$ is c if and only if $(u[k], c) \in R$ at some instant.*

The proof proceeds by induction in the evaluation graph, showing simultaneously in the induction step (1)-(4) as these depend on each other (in the previous inductive steps). Theorem 1 implies that every value of every defined stream at every point is eventually resolved by our network of cooperating monitors. Therefore, given input streams σ_I, the algorithm computes (by (4)) the unique output streams σ_i one for each s_i. The element $\sigma_i(k)$ is the value resolved for $s_i[k]$ by the local monitor for $\mu(s_i)$. The following theorem captures that Algorithm 1 computes the right values (according to the denotational semantics) and Theorem 1 that all values are eventually computed.

Theorem 2. *Let φ be a specification, $S = \langle \varphi, \mathcal{T}, \mu \rangle$ be a decentralized SRV problem, and σ_I an input. Then $(\sigma_I, out(\sigma_I)) \vDash \varphi$.*

3.3 Simplifiers

The evaluation of expressions in Algorithm 1 assumes that all instant variables in an expression e are known (i.e., e is ground), so the interpreted functions in the data theory can evaluate e. Sometimes, expressions can be partially evaluated (or even the value fully determined) knowing only some of the instant variables. A *simplifier* is a function $f : Term_D \to Term_D$ such that
- for every term t of type D, $Vars(f(t)) \subseteq Vars(t)$
- for every substitution ρ of $Vars(t)$, $[\![t \lhd \rho]\!]_{(\sigma_I, \sigma_O)} = [\![f(t) \lhd \rho]\!]_{(\sigma_I, \sigma_O)}$

For example, the following are sound simplifications

$\mathit{if\ true\ then}\ s[0]\ \mathit{else}\ t[1] \mapsto s[0]$	$0 + s[7] \mapsto s[7]$	$\mathit{true} \vee s[0] \mapsto \mathit{true}$
$\mathit{if\ false\ then}\ s[0]\ \mathit{else}\ t[1] \mapsto t[1]$	$1 \cdot t[23] \mapsto t[23]$	$\mathit{false} \vee s[0] \mapsto s[0]$

Simplifiers can dramatically affect the performance in terms of the instant at which an instant variable is resolved and the number of messages exchanged.

It is easy to see that for every term t obtained by instantiating a defining equation and for every simplifier f, $[\![t]\!]_{\sigma_I,\sigma_O} = [\![f(t)]\!]_{(\sigma_I,\sigma_O)}$, because the values of the variables in t and in $f(t)$ are filled with the same values (from σ_I and σ_O). The following also holds for every φ and valuation (σ_I, σ_O).

Lemma 1. *Let e be an instant term and let $\rho = \{u[k] \mapsto c\}$ be the substitution such that $c = [\![u[k]]\!]_{(\sigma_I,\sigma_O)}$. Then, $[\![e]\!]_{(\sigma_I,\sigma_O)} = [\![e \triangleleft \rho]\!]_{(\sigma_I,\sigma_O)}$.*

Lemma 1 holds immediately because the substitution ρ is just the partial application of one of the values of the variables that may appear in e. Now, consider arbitrary simplifiers *simp* used in line 43 to simplify expressions. Let U_n be the unresolved storage for node n and let $u[k]$ be an instant variable with $\mu(u) = n$. By Algorithm 1 the sequence of terms $(u[k], t_0), (u[k], t_1), \ldots (u[k], t_k)$ that U_n will store are such that $t_{i+1} = \mathit{simp}(t_i)$ or $t_{i+1} = t_1 \triangleleft \rho$ where $\rho = \{v_i[k_i] \leftarrow c_i\}$ corresponds to the substitution of values of instant variables that are discovered at the given time step. By Lemma 1, it follows that $[\![t_i]\!]_{(\sigma_I,\sigma_O)} = [\![t_{i+1}]\!]_{(\sigma_I,\sigma_O)}$ which in particular when $t_k = c$ implies that the value computed is $[\![u[k]]\!]_{(\sigma_I,\sigma_O)} = c$. The following theorem follows.

Theorem 3. *The decentralized algorithm using simplifiers terminates and computes the unique output for every well-formed specification φ.*

In fact, it is easy to show that the algorithm using simplifiers obtains the value of every instant variable no later than the algorithm that uses no simplifier. This is because in the worst case every instant variable is resolved when all its depending variables are known, and all response messages are sent at the moment they are resolved.

4 Decentralized Efficient Monitorability

In this section we identify a fragment of specifications, called *decentralized efficiently monitorable*, for which the local monitors only need bounded memory to compute every output value. To guarantee that a given storage in a local monitor for node n is bounded, one must provide a bound on both: (1) when a value $(u[k], c)$ in R_n can be removed; and (2) when it is guaranteed that an unresolved value from U_n is resolved. Note that if $s[k]$ is resolved in bounded time then all occurrences of $s[k]$ in W_n and P_n are also removed in bounded time, because it only takes a bounded amount of time for response messages to arrive.

Pruning the Resolved Storage. We show now that the memory necessary in the resolve storage R_n can be bounded (for all specifications). If a stream s is $\mathit{eager}(s)$ then once $s[k]$ is resolved it is sent to the potentially interested remote nodes. However, the value of $s[k]$ has to remain in R_n (and in R_m for remote nodes that receive it) until it is no longer needed. For streams s that are $\mathit{lazy}(s)$, the value must remain in R_n until it is guaranteed that the value will not be requested any more. This information is captured by the notion of back reference.

Algorithm 1 Local monitoring algorithm at node n with $\langle Q_n, U_n, R_n, P_n, W_n \rangle$

```
1: procedure MONITOR
2:     U_n, R_n, P_n, W_n ← ∅
3:     k ← 0
4:     while not END do
5:         STEP(k)
6:         k ← k + 1
7:     M ← k                                              ▷ Trace length M
8:     FINALIZE(M)

9: procedure STEP(k)
10:    R_old ← R_n
11:    for all msg ∈ Q do                                 ▷ Process incoming messages
12:        Q_n ← Q_n \ msg
13:        if msg.dst ≠ n then
14:            route(msg)
15:        else
16:            if msg.type = req then
17:                P_n ← P_n ∪ msg
18:            else
19:                R_n ← R_n ∪ {(msg.stream, msg.val)}
20:                W ← W \ {msg.stream}
21:    R_n ← R_n ∪ {r[k], new(r, k) | r ∈ inputs(n)}      ▷ Read inputs
22:    U_n ← U_n ∪ {s[k], instantiate(e_s, k) | s ∈ outputs(n)}   ▷ Instantiate outputs
23:    EVALUATE(U_n, R_n)
24:    for all (r[k'], c) ∈ R_n \ R_old do                ▷ New knowledge
25:        if eager(r) ∧ μ(r) = n then                    ▷ Eager new knowledge
26:            for all n_d ∈ μ(appears(r)) such that n ≠ n_d do
27:                send(resp, r[k'], c, n, n_d)
28:    for all msg ∈ P_n do                               ▷ Pending lazy new knowledge
29:        if (msg.stream, c) ∈ R_n then
30:            send(resp, msg.stream, c, n, msg.src)
31:            P_n ← P_n \ {msg}
32:    for all (_, e) ∈ U do
33:        for all u[k'] ∈ sub(e) do
34:            if lazy(u) ∧ u[k'] ∉ W_n ∧ μ(u) ≠ n then   ▷ Send needed new requests
35:                send(req, u[k'], n, μ(u))
36:                W_n ← W_n ∪ {u[k']}
37:    PRUNE(R_n)

38: procedure EVALUATE(U_n, R_n)
39:    done ← false
40:    while not done do
41:        done ← true
42:        for all (s[k], e) ∈ U_n do
43:            e' ← SUBST(e, R_n)
44:            if ground(e') then
45:                done ← false
46:                U_n ← U_n \ {(s[k], e)}
47:                R_n ← R_n ∪ {(s[k], e')}
48:            else
49:                U_n ← U_n \ {(s[k], e)} ∪ {(s[k], e')}
```

Algorithm 2 Pruning R_n at node n at instant k

```
1: procedure PRUNE
2:     for all (u[j], c) ∈ R_n do
3:         if k ≥ j + Δ(u) then                ▷ If u[j] will not be needed
4:             R_n ← R_n \ {(u[j], c)}          ▷ Remove
```

Definition 5. *Let φ be a Lola specification with dependency graph D_φ. The back reference of a stream s is*

$$\Delta(s) \stackrel{def}{=} \begin{cases} \max(0, \{-k \mid r \xrightarrow{k} s\}) & \text{if } eager(s) \\ \max(0, \{-k + dist(r,s) \mid r \xrightarrow{k} s\}) & \text{if } lazy(s) \end{cases}$$

Note that for lazy streams the request is guaranteed to be received after $dist(\mu(r), \mu(s))$ steps of the instantiation of the correspondent instance of r. Therefore, a node responsible for s will have received all requests for $u[k]$ at $k + \Delta(u)$. Similarly, a fetch for $u[k]$ in R_n locally at n is guaranteed to be done no later than $k + \Delta(u)$. Therefore, the following results holds.

Lemma 2. *A value $(u[k], c) \in R_n$ will not be fetched or requested after $k+\Delta(u)$.*

This implies that at every node n, all values of $u[k]$ can be removed at instant $k+\Delta(s)$, which allows to implement the algorithm for pruning shown in Algorithm 2. Therefore, the maximum size of R_n needed is bounded linearly by the maximum $\Delta(s)$ times the number of streams.

Time to resolve. In centralized SRV monitoring [10, 24] a specification is efficiently monitorable whenever all cycles in D_φ have negative weight. This guarantees that the online algorithm can be performed in a *trace length independent* way. However, this is not true for decentralized monitoring as illustrated in the following example.

Example 2. Consider the following specification deployed in monitors 1 and 2 with $dist(1,2) = dist(2,1) = 2$:

```
@1{output num a eval = b[-1|0]}
@2{output num b eval = a[-1|0]}
```

It is easy to see that $a[0]$ and $b[0]$ will be resolved at time 0, $a[1]$ and $b[1]$ at time 2, and $a[n]$ and $b[n]$ at time $2n$. Then, U_1 and U_2 will grow to contain a number of equations that depends on the length the trace. □

We introduce the notion of *decentralized efficiently monitorable*, that guarantees an upper bound on the number of steps that it takes to resolve an equation in U_n. Note that Algorithm 1 removes an equation from U_n and moves it into R_n once it is resolved. In turn, this also gives a bound on the duration of the elements in P_n and W_n. It follows that for decentralized efficiently monitorable specifications, the monitoring process requires only a constant amount of memory (on the size of the specification) independently of the length of the trace.

Definition 6 (Decentralized Efficiently monitoriable). *A specification φ is decentralized efficiently monitoriable whenever it is efficiently monitorable and no cycle in D_φ visits two streams r and s assigned to different nodes $\mu(r) \neq \mu(s)$.*

Note that since Lola is very expressive many decision problems for Lola specs (well-formedness, equivalence, etc) are undecidable. However, well-definedness (which guarantee that the monitoring algorithm always computes a verdict), efficient monitorability and decentralized efficient monitorability are syntactic properties which are very easy to check.

We now define the notion of look-ahead of a stream s, that bounds for decentralized efficiently monitorable specifications the maximum between the moment at which $s[k]$ is inserted in U_n and $s[k]$ is resolved into a value. Note that a decentralized efficiently monitorable specification can be decomposed into a DAG of sets of stream variables such that each set is mapped to a single node (because cycles in the graph must belong to a single node). We use $S(s)$ for the set of streams that are grouped with s. In order to define the look-ahead distance $\nabla(s)$ we use an auxiliary definition: $\nabla_{rem}(s)$. This provides an upper-bound on the time to receive from a remote node the value of an instant variable $r[k']$ that $s[k]$ directly depends on.

- If r is eager, this value depends on $\nabla(r)$ to guarantee that $r[k']$ is known at $k' + \nabla(r)$ and the time $dist(r, s)$ to communicate this value.
- If r is lazy, the instant at which the network node of r sends the value of $r[k']$ is the later instant between $k' + \nabla(r)$ and the reception of the request, that is $k + dist(s, r)$. After receiving the request, the response takes $dist(r, s)$ to arrive to s.

Once ∇_{rem} has been determined for all edges in the dependency graph that leave a component $S(s)$, $\nabla(s)$ can be determined by the weight of the maximum simple path in $S(s)$ adding also the additional time to resolve the remote dependencies. Note that the definition of $\nabla(s)$ is identical to the look-ahead in a centralized specification with $S(s)$ as streams that considers directly accessible streams r at remote nodes as input streams. Formally:

Definition 7 (Look-ahead). *The remote look-ahead distance $\nabla_{rem}(s)$ of a stream s is $\nabla_{rem}(s) \stackrel{def}{=} \max(0, \{delay(r \xrightarrow{w} s) \mid \mu(r) \neq \mu(s)\}$, where*

$$delay(r \xrightarrow{w} s) \stackrel{def}{=} \begin{cases} dist(r, s) + w + \nabla(r) & \text{if eager}(r) \\ dist(r, s) + \max(w + \nabla(r), d(s, r)) & \text{if lazy}(r) \end{cases}$$

The look-ahead distance is $\nabla(s) \stackrel{def}{=} \max(0, \{w + \nabla_{rem}(r) \mid s \xrightarrow{w}{}^{*} r \text{ with } S(s) = S(r)\})$

The definition is well-defined because the graph is a DAG of components, each of which is mapped to single network node. Intuitively, the remote look-ahead $\nabla_{rem}(s)$ captures how long it takes to receive information from $\mu(r)$ that is relevant to compute $s[k]$. Note that if the specification is centralized, then there is a single component, $\nabla_{rem}(s)$ is 0 and the look-ahead distance coincides with the look-ahead for centralized Lola evaluation [24].

Lemma 3. *Every unresolved $s[k] = e$ in U_n is resolved at most at $k + \nabla(s)$.*

Lemmas 2 and 3 imply that decentralized efficiently monitorable specifications can be monitored with bounded resources. The bound depends only linearly on the size of the specification and the diameter of the network.

5 Empirical Evaluation

We have implemented our solution in a prototype tool dLola, written in the Go programming language (available at http://github.com/imdea-software/dLola). We describe now (1) an empirical comparison of dLola versus Themis [13]—a state-of-the-art tool for decentralized runtime verification of LTL specifications—and (2) the effect on dLola of the network placement on richer specifications (not supported by Themis).

Themis comparison. Themis can only monitor Boolean specifications while dLola can monitor arbitrary values from richer domains. Also, Themis can only handle a clique topology while dLola supports arbitrary connected networks. In this comparison, we restrict to specifications and topologies that Themis can handle, and we translate directly LTL formulas to Lola specifications. We evaluate both tools against 213,196 synthesized input tests in a network with 5 nodes. The results from Themis where obtained from the database provided openly at https://gitlab.inria.fr/monitoring/themis. Our tool reached a final verdict on all cases, which coincided with Themis on all experiments for which Themis had a verdict in the database (85% of our input cases). Fig. 2 report metrics collected using these experiments. We compared our centralized setting (with decentralized observation) with the Themis' Orchestration algorithm and our decentralized setting with Themis' Choreography algorithm. Fig. 2(a) shows the number of messages exchanged to compute the final verdict. In the best case

	min			avg			max		
	dLola		Themis	dLola		Themis	dLola		Themis
	Lazy	Eager		Lazy	Eager		Lazy	Eager	
decentr	6.00	12.00	0.00	564.19	332.50	6751.12	4201.00	2101.00	66000.00
centr	1.00	9.00	0.00	98.33	140.88	7085.40	1001.00	801.00	48400.00

(a) Number of messages exchanged

decentr	139.50	279.00	0.00	13186.87	7792.24	60743.17	97862.00	49074.50	594000.00
centr	24.50	204.50	0.00	2208.38	3171.91	83833.05	82759.45	22462.00	576950.00

(b) Payload size (in bits)

decentr	2.00	1.00	0.00	23.72	20.49	84.46	115.00	110.00	4070.00
centr	0.00	0.00	0.00	17.61	16.59	6.52	101.00	100.00	110.00

(c) Time delay (in cycles)

Fig. 2. Comparison dLola vs Themis

	Ring			Ringshort			Line			Clique			Star		
	best	even	worst	best	even	worst	best	even	worst	best	even	worst	best	even	worst
4	301	1301	2901	301	1301	1301	301	1501	2401	301	901	1101	301	1401	2101
5	301	1301	3903	301	1301	2103	301	1501	3401	301	901	1101	301	1501	2101
7	301	1301	5701	301	1301	3001	301	1801	5401	301	901	1101	301	2401	3901
9	301	1301	5901	301	1301	4301	301	1301	7401	301	901	1101	301	2301	3901
10	301	1301	6501	301	1301	4501	301	1301	8401	301	901	1101	301	2701	5701

Fig. 3. Number of messages exchanged by topology and placement (for $4, \ldots, 10$ nodes)

a lazy strategy requires less messages than an eager strategy because many remote values are not required. In the worst case the eager strategy consumes less messages than the lazy, because the request messages are not sent.

In comparison with Themis, dLola requires less messages on average and in the worst case, but more messages in the minimum case. Fig. 2(b) shows the size of the message payload used for the computation of verdicts. Again, dLola uses smaller payloads except in the minimum case. Fig. 2(c) contains the maximum delay, which shows that dLola incurs in a higher maximum delay for the centralized cases, but significantly lower when decentralized.

Topologies. Intuitively speaking, the performance depends on the placement of streams, as more locality reduces the latency and the number of messages required. We selected five representative topologies (ring, ringshort, linear, clique, star) and for each topology selected three different placements: (1) maximizing manually the locality, (2) assigning output streams evenly, and (3) minimizing manually the locality. For all experiments we use the following specification, where we make a chain of four output streams depend on an input and on the previous stream. Fig. 3 illustrates how the placement of subformulas affect the overall efficiency of the monitors, which confirms that placement is crucial for efficiency and suggests that in most cases, values can be resolved with a number of messages independently of the topology and size of the network by careful placement. This is relevant since the topology may be fixed by the system design, while the placement is part of the monitoring solution.

6 Conclusions and Future Work

We have studied the problem of decentralized stream runtime verification, that starts from a specification, a topology and a placement of the input streams. Our solution consists of a placement of output streams and an online local monitoring algorithm that runs on every node. We have captured specifications that guarantee that the monitoring can be performed with constant memory independently of the length of the trace. We report on an empirical evaluation of our prototype tool dLola. Our empirical evaluation shows that placement is crucial for performance and suggest that in most cases careful placement can lead to constant costs and delays. As future work we plan to extend our solution to timed asynchronous distributed systems [9], to monitor under failures and uncertainties and to support reading at different nodes (alternatively or simultaneously).

References

1. Autosar. https://www.autosar.org/.
2. Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.
3. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proc. of the 5th Int'l Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
4. David Basin, Felix Klaedtke, and Eugen Zalinescu. Failure-aware runtime verification of distributed systems. In *Proc. of the 35th IARCS Annual Conf on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'15)*, volume 45 of *LIPIcs*, pages 590–603. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.
5. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2013.
6. Andreas Bauer, Martin Leucker, and Chrisitan Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14, 2011.
7. Andreas Klaus Bauer and Yliès Falcone. Decentralised LTL monitoring. In *Proc. of the 18th Int'l Symp. on Formal Methods (FM'12)*, volume 7436 of *LNCS*, pages 85–100. Springer, 2012.
8. Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. TeSSLa: Temporal stream-based specification language. In *Proc. of the 21th Brazilian Symp. on Formal Methods (SBMF'18)*, volume 11254 of *LNCS*, pages 144–162. Springer, 2018.
9. Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.
10. 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 the 12th Int'l Symp. of Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE CS Press, 2005.
11. Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. of the 15th Int'l Conf. on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 27–39. Springer, 2003.
12. Antoine El-Hokayem and Yliès Falcone. Monitoring decentralized specifications. In *Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA'17)*, pages 125–135. ACM, 2017.
13. Antoine El-Hokayem and Yliès Falcone. THEMIS: A Tool for Decentralized Monitoring Algorithms. In *Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA'17)*, pages 125–135. ACM, July 2017.
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, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Torfah Hazem. StreamLAB: Stream-based monitoring of cyber-physical systems. In *Proc. of the 31st Int'l Conf. on Computer-Aided Verification (CAV'19)*, volume 11561 of *LNCS*, pages 421–431. Springer, 2019.
16. Adrian Francalanza, Jorge A. Pérez, and César Sánchez. Runtime verification for decentralised and distributed systems. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 176–210. Springer, 2018.
17. Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. In *Proc. of the 18th Int'l Conf. on Runtime Verification (RV'18)*, volume 11237 of *LNCS*, pages 282–298. Springer, 2018.
18. Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Proc. of the 8th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 342–356. Springer-Verlag, 2002.
19. ISO Central Secretary. Road vehicles interchange of digital information controller area network (CAN) for high speed communication. Standard ISO 11898, International Standards Organisation, 1993.
20. E. K. Liebemann, K. Meder, J. Schuh, and G. Nenninger. Safety and performance enhancement: the Bosch electronic stability control(ESP). In *SAE*, pages 421–428, 2004.
21. Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Proc. of the 1st Int'l Conf. on Runtime Verification (RV'10)*, volume 6418 of *LNCS*, pages 345–359. Springer, 2010.
22. Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: monitoring embedded systems. *Innovations in Systems and Software Engineering*, 9(4):235–255, Dec 2013.
23. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.
24. César Sánchez. Online and offline stream runtime verification of synchronous systems. In *Proc. of the 18th Int'l Conf. on Runtime Verification (RV'18)*, volume 11237 of *LNCS*, pages 138–163. Springer, 2018.
25. Koushik Sen and Grigore Roşu. Generating optimal monitors for extended regular expressions. In Oleg Sokolsky and Mahesh Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.
26. Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proc. of the 26th Int'l Conf. on Software Engineering (ICSE'04)*, pages 418–427. IEEE CS Press, 2004.