Master in Software and Systems

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros Informáticos

MASTER THESIS

Decentralised Stream Runtime Verification

Author: Luis Miguel Danielsson Villegas
Supervisor: César Sánchez
Co-Supervisor: Clara Benac Earle

MADRID, JULY, 2018

Abstract

We study the problem of decentralized monitoring of stream runtime verification (SRV) specifications. Decentralized monitoring consists of organizing a monitoring activity to be performed by distributed components that communicated using a synchronous network, a setting common in some cyber-physical systems like automotive CPSs. Previous approaches to decentralized monitoring were restricted to LTL and similar logics whose monitors compute Boolean verdicts. We present here an algorithm that solves the decentralized monitoring problem for the more general setting of stream runtime verification. Additionally, our algorithm handles network topologies were previous work assumed a network in which all nodes can communicate directly.

Our algorithm is able to reach verdicts efficiently by exploiting partial evaluation strategies, expression simplifiers and advanced communication strategies. Finally, we present 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.

Resumen

Se estudia el problema de monitorización descentralizada de especificaciones de stream runtime verification (SRV). La monitorización descentralizada consiste en organizar una actividad de monitorización para que sea realizada por componentes distribuidos que se comunican utilizando una red síncrona, un escenario común en algunos sistemas ciber-físicos(CPS) como en automoción. Alternativas previas de monitorización descentralizada se restringían a LTL y lógicas similares cuyos monitores computan veredictos booleanos. Se presenta un algoritmo que solventa el problema de la monitorización descentralizada para el caso más general de stream runtime verification. Adicionalmente, el algoritmo gestiona topologías de red mientras que trabajos previos asumían una red en la que todos los nodos se podían comunicar directamente. Este algoritmo es capaz de obtener veredictos de forma eficiente al hacer uso de estrategias de evaluación parcial, simplificadores de expresiones y estrategias avanzadas de comunicación. Finalmente, presentamos los resultados de una evaluación empírica de una implementación y comparamos la expresividad y la eficiencia con la herramienta de monitorización descentralizada de vanguardia Themis.

INDEX

1 Introduction **1**
- 1.1 Related work . . . 3
- 1.2 Our Solution . . . 3
- 1.3 Uses . . . 4

2 Preliminaries **7**
- 2.1 Runtime Verification . . . 7
- 2.2 Stream Runtime Verification . . . 8
- 2.3 LTL . . . 9
- 2.4 Decentralised Systems . . . 9
- 2.5 Lola:SRV of Centralized Synchronous Systems . . . 10
- 2.6 Assumptions . . . 14

3 Decentralized Monitors **17**
- 3.1 Goals of the solution . . . 18
- 3.2 Distributing monitors . . . 19
- 3.3 Data Domains . . . 22
- 3.4 Communication Strategies . . . 23
- 3.5 Computation duration . . . 24
- 3.6 Local Algorithm: Operational Semantics . . . 24
- 3.7 Simplification . . . 28

4 Empirical Evaluation **37**

5 Conclusions and Future work **45**

Bibliography **47**

A Additional Information **51**
- A.1 More Examples . . . 51

Chapter 1

Introduction

Checking both safety and liveness properties on reactive concurrent systems is particularly hard, since classical formal verification techniques are not suitable for non-terminating programs as it is the case for reactive systems. Moreover, concurrency present a great challenge when trying to decide whether a certain property holds in the system due to the interleavings.

Concurrent reactive systems include a wide variety of systems such as cyber-physical systems (CPS) or web servers among others which are in nature concurrent and reactive. Often these systems are really big, for example, CPS systems include software in automotive or for aerospacial industries where low-level hardware controllers and high-level tasks must be performed in symbiosis. In the case of cloud computing, the whole system should behave semantically as a single server but with the performance of a distributed system. In these scenarios is particularly interesting to be able to check properties as "every request has a matching response" or computing the "average time that a server takes to respond". However, the most common industry techniques: testing and peer reviews are not concerned about these kind of properties so there is virtually no tool used in industry to check properties on reactive concurrent systems.

Nowadays in industry the most common techniques for verifying software are testing and peer-reviews. Testing has improved greatly since the apparition of tools that analyze test coverage. But even advanced techniques such as concurrent coverage testing, which controls the scheduler to produce interleavings that may lead to concurrent errors, may not find every single bug in the code. The purpose of these techniques is to find bugs but they cannot tell whether a property holds in the system.

Static verification techniques like model checking [Queille and Sifakis, 1982] [Emerson and Clarke, 1980] could provide guarantees about the safety and liveness properties of the system but they suffer from the lack of proper models of the

physical world. This limitation is particularly relevant for reactive systems that interact not only with their physical environment but also can interact with humans, what is called human-in-the-loop systems. Modeling human behavior is not an easy option for mathematicians and computer scientists so it is better to find alternatives. Having improper models, although feasible, would make the results of the model checker useless, since the model would not reflect reality.

Moreover, the use of abstractions in the model often lead to imprecisions and therefore, false positives. In addition, model checking suffers from the state explosion problem. This means that model checking analyzes all possible states of a program so, for large programs -and reactive programs usually are- the state space is too big to be handled by the model checker and its time and space requirements may not be feasible in practice.

There is yet another inconvenient when using model checking. Nowadays software evolves so rapidly and considering that model checking is not a quick process, it may take longer to produce some result than an update of the software, thus yielding all the model checking work obsolete.

Runtime Verification (RV) deals with the problem of assessing whether a given property holds in a running system. In order to do that, the property must be expressed in some specification language. Early approaches for specification languages were based on temporal logics [Havelund and Roşu, 2002, Eisner et al., 2003, Bauer et al., 2011], regular expressions [Sen and Roşu, 2003], timed regular expressions [Asarin et al., 2002], rules [Barringer et al., 2004], or rewriting [Roşu and Havelund, 2005]. Stream runtime verification techniques, pioneered by Lola [D'Angelo et al., 2005], define monitors by declaring the dependencies between output streams of results and input streams of observations. This approach leads to a richer formalism that goes beyond Boolean verdicts, like in logical techniques, to allow the specification of the collection of statistics and can produce quantitative verdicts.

Other important aspects of a RV technique are how to collect information from the system under analysis and how to perform the monitoring process. The RV technique that we present here is intended to be used *online* (running alongside the system), *outline* (so the monitors are not intertwined with the code and the monitoring is *non-intrusive*). We also assume that the system under observation is distributed and that the monitors can use distributed computational nodes and that the communication network is synchronous. This problem is known as the *decentralized monitoring* problem. The goal is to generate local monitors that collaborate to perform the global monitoring, distributing the computational load of the monitoring process, trying to minimize the latency and the bandwidth. We provide solutions for arbitrary network topologies and placement of the local monitors.

In the following section we will describe some of those solutions and the main features they present.

1.1 Related work

In the following lines we describe some of the several approaches to Runtime Verification that have different assumptions about the computing paradigm as well as the solutions applied. The work in [Basin et al., 2015] 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. In [Sen et al., 2004b] a variant for linear temporal logic for distributed systems PT-DTL is presented, with a distributed monitoring algorithm. Distributed message passing environment with a global clock is handled in [Francalanza et al., 2018] which uses slices to support node crashes and message errors. The work in [Bauer and Falcone, 2011] studies how to monitor LTL_3 in a perfectly synchronous system, presenting a decentralized solution in which local monitors pass obligations to other monitors, simplifying the formula as it traverses the network. An advanced data-structure called Execution History Encoding (EHE) is introduced in [El-Hokayem and Falcone, 2017a, El-Hokayem and Falcone, 2017b]. The EHE ensures strong eventual consistency and is used both for centralized and decentralized monitoring. However, all these approaches consider only temporal logics with Boolean verdicts.

All previous SRV efforts, from Lola [D'Angelo et al., 2005], Lola2.0 [Faymonville et al., 2016], Copilot [Pike et al., 2010, Pike et al., 2013] and extensions to timed event streams, like TeSSLa [Convent et al.,], RTLola [Faymonville et al., 2017] or Striver [Gorostiaga and Sánchez,] follow a centralized monitoring approach, where all observations arrive to a single node where the monitoring activity takes place.

1.2 Our Solution

The solution we present in this thesis is a stream runtime verification framework specialized for decentralised synchronous environments. We can assume a global clock that is so slow (or computation so fast) that every node has enough time to receive messages, perform the necessary computations and put messages in the incoming queues of the neighboring nodes in just a cycle (a tick of the global clock). Therefore we assume a perfectly synchronous environment. This kind of assumption is perfectly real in some scenarios such as cyber-physical systems, for example in automotion where different components can communicate via a common BUS (see Chapter 1.3). We assume that the computation nodes are reliable, they do not crash and that message passing is also reliable, messages neither have delays nor they are lost nor duplicated. The system is decentralised in the sense that the observations are made in different nodes and there is no unique node with all the observations. Thus, we will have observations performed in multiple nodes, where we will deploy monitors (one per node) that will be responsible of computing some

part of a specification. Cooperation among nodes will result in resolving the whole specification and reaching a final verdict collectively.

Monitors should be able to use the actual topology of the nodes and communicate using the same network. This is the reason why we will make our best to keep communication at a minimum. A degenerated example of a decentralized solution is the following: having some nodes with observations, choose one of them and proxy all observations to that node that is also responsible of computing the whole specification. This is an example of decentralised observation with centralised computation. We are looking for solutions that exploit the capabilities of the nodes to improve collaboration and performance.

We would like a language expressive enough to describe Boolean assertions as well as richer domains as Integers, Maps or Sets. These richer domains enable the computation of statistics of the execution under analysis. Also, we would like a specification language that does not require causality among streams, allowing dependencies to future events (even though the execution of such specifications may not be efficient).

Intuitively, we want the decentralised setting to emit the same verdict that the centralised one would emit (correctness) and also minimize the network consumption (number of messages and their size), the computational load as well as the delay between the instant at which all the required data in the system is present and the time at which monitors emit the final verdict (delay).

1.3 Uses

In this sectopm we present a motivating example based on industry standards for automotive. Namely, the CAN bus and the AUTOSAR framework.

CAN Network The CAN network is implemented with a CAN bus which is a type of asynchronous BUS in the sense that computation nodes need not share the BUS clock internally. However, whenever a node wants to send a message, it is synchronized to the clock of the BUS. So, in the end a CAN BUS provide a synchronization mechanism that is shared across the network. Multiple computation nodes, usually ECUs, can be connected to the CAN BUS.

Versions In the standard ISO 11898-2 a high rate CAN bus is presented which can only have the topology of a BUS. Logically, this is equivalent as a clique topology because every node is directly connected to any other one. In the standard ISO 11898-3 a low-rate fault-tolerant CAN BUS is presented which admits either star or

multiple stars connected by a central BUS topologies.

AUTOSAR AUTOSAR is an industry standard framework for automotive applications and provides services such as scheduling or communication. The AUTOSAR framework is organized in several layers and can support different communication infrastructures such as CAN BUS, Lin or FlexRay. All these communication infrastructures are used in the automotive industry.

We have ideintified the following assumptions for our solution to apply that resemble a realistic scenario:

- Global clock (See Section 2.6)
- Multiple computation nodes (See Sections 3 and 2.6)
- Arbitrary topologies (See Section 2.6)
- Messages of arbitrary size (contain lists)
- Routing determined statically at initialization (See Section 2.6)
- Messages are reliable and arrive in a tick
- Nodes can send a bounded number of msgs in a tick

The first 3 properties are provided by the CAN BUS while the next two are provided by the AUTOSAR framework. A framework providing the remaining properties might be built upon these layers.

This thesis is structured as follows: in Chapter 2 all foundations needed to understand this thesis are presented; in Chapter 3 our solution is presented; in Chapter 1.3 an industry related motivating example is presented; in Chapter 4 the empirical setting where we compare our results with those of [El-Hokayem and Falcone, 2017b] is described; in Chapter 5 we state our conclussions.

This thesis proposes a solution to address the problem of decentralised monitoring using LOLA [D'Angelo et al., 2005] as a base for the monitoring algorithm, for a scenario that meets the assumptions stated in 3.1

Chapter 2

Preliminaries

In this chapter we present the underlying theory that we build our solution upon, and the assumptions about the computing paradigm in which our solution can solve the task at hand. We also state the goals we would like to achieve both as functional and non-functional properties.

2.1 Runtime Verification

Runtime Verification (RV) [D'Angelo et al., 2005] [Barringer et al., 2004] [Lee et al., 1999] [Dong et al., 2008] is an area of formal verification that tries to verify the behaviour of a program by observing one run. Thus, the goal is to observe the runtime of the program and be able to tell whether the run of the program satifies or violates a given property. Properties may be expressed in logics such as LTL (see Section 2.3) or may be expressed using a programming language or any other formal means. RV inherits from model checking the logics and methodologies of expressing properties but, as in RV we study a single run of the program, no model of the program is required. This has some consequences: on one hand, the outcome of the analysis will be precise but at the cost of losing completeness. This means that in RV there are no false positives or negatives but as the object of study is a single run, the outcome of the monitors is whether that single run complies or not to the specification. Therefore, in the case that the run complies we do not know if the system as a whole complies with the property or not. In a certain sense, RV checks for counterexamples that prove that a system does not adhere to a property, but cannot prove the compliance of the system to a property since it would need to observe every possible run, which may be unfeasible or even infinite.

Online monitoring Online monitoring [Sen et al., 2004a] [D'Angelo et al., 2005] is the technique for monitoring specifications while the monitored system is actually running.

In order to observe the runtime of the program, there are two options: inlining the monitoring code in the program resulting in tightly coupling or instrumenting the program to extract useful information and analysing it in a loosely coupled fashion.

This last option can be achieved either by simulation or by running in parallel. Simulation means using a scheduler that allows the system to run and produce some events and then letting the monitor(s) consume those events as inputs and so on, virtually keeping the system and the monitor synchronized. The parallel running way consists of running freely (without a scheduler) both the system and the monitors asynchronously so that as soon as the system produces any output it is ready for monitor(s) to be consumed. The issue with this second methodology is that monitors need to keep up with the pace marked by the system. The main advantage of online monitoring is the capability of reducing the time to detect a violation of a property, thus being able to take some corrective action earlier. Obviously, online monitoring has a negative impact on the system's efficiency as the monitor requires some resources and these are tipically shared between the monitor and the system.

Offline monitoring Offline monitoring [D'Angelo et al., 2005] is a different approach to runtime verification in which a trace of the running system is dumped to some type of storage (e.g. a database dump or a log) and then monitors use this data for post-mortem analysis. The advantage of this technique is the complete loose decoupling between the system and the monitors and it has no negative effect on the system's efficiency. However, using this approach renders the delay between the occurrence of a violation and its detection to be much higher.

2.2 Stream Runtime Verification

Stream Runtime Verification (SRV) [D'Angelo et al., 2005] is a branch of Runtime Verification based on observations from the monitored system that are received as an input trace of events, either offline or online. In SRV inputs are modeled as streams of values and outputs are modeled as streams of values computed from the input streams. Specifications express how to compute output streams from input streams. Thus, representing the different observations of the monitored program computation as a trace in time, SRV can effectively compute a specification over the traces to assess properties. In SRV, all states of the program are decomposed into variables, thus converting them in a set of stream values. E.g. let the state of the program be the variables x, y, then at each point in time (maybe each instruction

of the program), these variables have a different value of the corresponding type. We could express these values as as $x[t]$ read as "the value of variable x at the time point t". Therefore we can model the sequence of observations of the system as a set of stream variables $x[0], ..., x[n]; y[0], ..., y[n]$. These streams can be seen as input channels that receive the observations of a given variable.

2.3 LTL

LTL [Pnueli, 1977] is a modal temporal logic widely used to describe specifications in the runtime verification community. LTL is suitable to express specifications that require some kind of temporal relation among predicates. LTL allows for expressing future-only, past-only formulas or a combination of future and past. Thus, LTL provides the following future operators: $\Box$, $\Diamond$, $\bigcirc$, $\mathcal{U}$, $\mathcal{W}$. The informal semantics of each of them is the following: $\Box$ is used to express invariants, that is a property that holds in all future states; $\Diamond$ is used to express a property that must hold at some unknown point in the future; $\bigcirc$ is used to describe a property that must hold in the next state of the computation; $p\,\mathcal{U}\,q$ is used to express that some property p holds up to the point when other property q holds; $\mathcal{W}$ is very similar to the previous one but it doesn't require q to be satisfied, in which case p must hold forever.

Analogously, LTL provides past operators: $\boxminus$, $\diamondminus$, $\ominus$, $\circledcirc$, $\mathcal{S}$, $\mathcal{B}$. The informal semantics of these operators is: $\boxminus$ is used to express a property that holds on all previous states; $\diamondminus$ is used to express that a property must hold in some previous state; $\ominus$ is used to express a property that must hold in the previous state and there must be one such previous state; $\circledcirc$ is used to express a property that must hold at the previous state if it exists, otherwise it is true; $p\,\mathcal{S}\,q$ is used to express that after q holds then p must hold in all subsequent states.; $p\,\mathcal{B}\,q$ is very similar to the previous one but allows q never holding, in this case, making p mandatory to hold in all previous states.

2.4 Decentralised Systems

Decentralised systems are composed by computing nodes spatially distributed and connected using a computer network. These nodes contain a CPU, memory and a means of communication with other nodes. In decentralised systems the computation is organized in chunks that can be assigned to the nodes of the system. The coordinated work of the nodes results in the completion of the global computation (usually with performance benefits with respect to a single centralized solution).

2.5 Lola:SRV of Centralized Synchronous Systems

Lola [D'Angelo et al., 2005] is both a specification language and a monitoring algorithm. A specification in this language is described by defining a relation between typed output streams and typed input streams, thus defining typed output streams from typed input streams. This language allows expressing properties both for future and past, and both Boolean properties as well as the computation of metrics (profiling) of the system.

Lola can perform the monitoring in two different ways, offline or online. In online monitoring the monitors are running along with the monitored system, whereas in offline the system is executed until termination and a dump of traces is then analysed. In the online monitoring algorithm the trace must be analysed as it is received in order to be ready when the next event in the trace is available. Two stores are used, which are empty at the begining: Resolved and Unresolved. Stores are modelled as sets of pairs corresponding to the value of a stream (in the case of Resolved) and the symbolic expression of a stream (in the case of Unresolved). The first one stores equations of the form $\varphi[t] = c$ while the latter stores equations of the form $\varphi[t] = e$ where e is an unresolved expression. At each time, new values from the input streams are received and added to Resolved; then with this new information, those values are substituted in every expression where they appear, which are then simplified until a fixed point is reached. The simplification phase uses both substitution and partial evaluation. A specification is efficiently monitorable if its worst case memory consumption is constant under the online monitoring algorithm. It is known that if the dependency graph of a specification contains no positive cycles then it is efficiently monitorable.

Lola syntax A Lola specification [D'Angelo et al., 2005] is a set of equations over typed stream variables of the form,

$$\begin{aligned} s_1 &= e_1(t_1, ..., t_m, s_1, ..., s_n) \\ &\vdots \\ s_n &= e_n(t_1, ..., t_m, s_1, ..., s_n) \end{aligned}$$

where $s_1, \ldots, s_n$ are called the output variables and $t_1, \ldots, t_m$ are called the input variables, and $e_1, \ldots, e_n$ are stream expressions of the appropiate types. A Lola specification can also declare certain output boolean variables as triggers. Triggers make the monitor generate notifications at instants when their corresponding values become true. Triggers are specified in Lola as *trigger* φ.

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 $f : T_1 \times T_2 \times, ..., \times T_k \longrightarrow T$ be a k -ary operator. Let for $1 \leq i \leq k$, e_i be 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 $ite(b, e_1, e_2)$ 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.

Example 1. *Consider the property "calculate the sum of the input stream y. But if the reset stream is true, reset the counter". This property can be expressed in Lola in the following way:*

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

The stream `root` *gives the desired output by taking the accumulator* `acc` *unless a* `reset` *is received.* □

Lola semantics.

The semantics of Lola specification are defined denotationally, by capturing whether given an input σ_I and given an output candidate σ_O, the pair σ_I, σ_O matches the specification. More formally, an input is a sequence of values σ_x one for each input stream variable x (all of the same length N). The sequence of values σ_x are values from the type of x. Similarly, an output candidate is a sequence of values σ_y one for each output variable y, of length N, where all the values are from the type of y. A *valuation* σ is an input and an output candidate. Given a valuation we the *evaluation* $[\![t]\!]_\sigma$ of a term t is a sequence of values of length N of the type of t defined as follows:

- If t is c, then $[\![c]\!]_\sigma(j) = c$.
- If t is a stream variable x, then $[\![x]\!]_\sigma(j) = \sigma_x(j)$.

- If t is $f(v_1, \ldots, v_n)$ then $[\![f(v_1, \ldots, v_n)]\!]_\sigma(j) = f([\![v_1]\!]_\sigma(j), \ldots, [\![v_n]\!]_\sigma(j))$
- If t is $ite(b, e_1, e_2)$ then

$$[\![ite(b, e_1, e_2)]\!]_\sigma(j) = \begin{cases} [\![e_1]\!]_\sigma(j) & \text{if } [\![b]\!]_\sigma(j) \text{ is true} \\ [\![e_2]\!]_\sigma(j) & \text{otherwise} \end{cases}$$

- If t is $e[i, c]$ then

$$[\![e[i, c]]\!]_\sigma(j) = \begin{cases} [\![e]\!]_\sigma(j + i) & \text{if } 0 \leq j + i < N \\ c & \text{otherwise} \end{cases}$$

Now, a valuation σ satisfies a Lola specification φ whenever for every output variable s_n:

$$[\![s_n]\!]_\sigma = [\![e_n]\!]$$

In this case we say that σ is an evaluation model of φ.

Note that this semantics capture when a candidate is an evaluation model. The intention of a Lola specification is to compute the unique output streams for the output variables, given input streams for the input variables. Unfortunately, there are specifications (for example *bool* $s = not\ s$) that have no evaluation model. Similarly, there are specifications (for example *bool* $s = s$) that have multiple evaluation models. A specification is *well-defined* when for every input there is a unique output that forms an evaluation model. Well-definedness is a semantic condition, which is hard to check.

However, Lola [D'Angelo et al., 2005] introduces a syntactic condition, called *well-formedness* that guarantees well-definedness. A key concept is that of a dependency graph. A dependency graph is a multi-graph whose vertices are the stream variables and whose edges are the dependencies between streams with their corresponding time shifts, given by the offset expressions in the defining expressions.

Definition 1 (Timed stream)**.** *We define $st(\varphi) = \sigma_1, ..., \sigma_i$ as the set of subexpressions of φ.*
Thus, given $\pi : \{0..N\}$
We define $tst(\varphi) \stackrel{def}{=} st(\varphi) \times \pi : \{\sigma[t] \mid \sigma \in st(\varphi) \text{ and } t \in \pi\}$

Definition 2 (Evaluation Graph)**.** *Let $n \in \pi = [0..N]$, where N is the length of the trace, $k \in \mathbb{Z}$, $d_1, ..., d_{j+i}$ set of constants of the type of the corresponding stream (Bool, Int, ...), $i_1, ..., i_i$ inputs.*
Let $\sigma_1, ..., \sigma_j$ be streams of the form $\sigma_j[n] = f(\sigma_1[n - k_1|d_1], ..., \sigma_j[n - k_j|d_j], i_1[n - k_{j+1}|d_{j+1}], ..., i_i[n - k_{j+i}|d_{j+i}])$. Thus, as n is always a finite number and every stream depend on a finite number of other streams and inputs, we can build the evaluation graph. Starting form the dependency graph (Lola) one can build the evaluation graph

which would have an instance of each node of the dependency for each each instant n. Evaluation Graph is a pair (V, E)

$V = \{\sigma[n] | \textit{for every stream } \sigma \textit{ and instant } n\}$
$E = \{(\sigma[n], r[n+k]) | \textit{for every } r, \sigma, \textit{ such that } r[n+k|d] \in exp(\sigma[n]) \textit{ and } 0 \leq n+k \leq N\}$

In this graph every stream in an instant and input in an instant will be a node, and the edges will be dependencies. This way, a stream $\sigma[n]$ which depend on the substream $r[n+k]$ can be formally written as $\sigma[n] \rightarrow_E r[n+k]$

A specification is *well-formed* if in its dependency graph there is no closed-walk of weight zero, which would correspond to a cyclic dependency. Essentially, well-formed specifications prevent cyclic dependencies between a stream at a given position and itself at the same position, guaranteeing that there is a unique candidate for each position of every output stream.

Lemma 1. *Let φ be a spec, G_φ its dependency graph and (V,E) an Evaluation Graph for [0..N], then (V, E) contains no cycles. (V,E) is an acyclic directed graph (unweighted). Consequently, every node $\sigma[n]$ in the Evaluation Graph has a longest finite path.*

Proof. By contradiction. Suppose there is a cycle in the Evaluation Graph, then there must be a strongly connected component (SCC) in G_φ of weight 0. But we do not allow 0 weighted cycles for well-formed specifications in the dependency graph, as in [D'Angelo et al., 2005]. So this is a contradiction. □

Definition 3 (Stream rank)**.** *The length of the longest path of a node in the Evaluation Graph is what we shall call stream rank as $\rho(\sigma[n])$. Therefore ρ is a function defined as $\rho : V_{EvalGraph} \longrightarrow \mathbb{N}$*

$$\rho(\varphi) = \begin{cases} 0 & v \textit{ is a leaf} \\ max(\rho(\sigma) | \varphi \rightarrow_E \sigma) + 1 & otherwise \end{cases}$$

From this definition we can extract the following:
If $(\sigma[n] \rightarrow_E r[n+k])$ then $\rho(r[n]) < \rho(\sigma[n])$

Formulas whose dependency graph has no positive cycles (for example, past formulas) can be monitored in constant space and in constant time per event, independently of the trace length. For future operators the space required depends on the length of the trace which may not be feasible for large traces.

Example 2. *Lets consider the previous example deployed in an isolated monitor that receives as an input the streams i and $reset$. Both streams are received at each corresponding cycle, so $i[0]$ and $reset[0]$ will be received at $n = 0$ and stored in Resolved and so on for all n.*

Starting at $n = 0$, the expression will be instantiated with the current value of n, replacing n with its actual value in the expression, in this case $acc[0] = i[0] + root[0-1|0] = i[0] + root[-1|0]$. As $root[-1|0]$ is an invalid position in the stream root, then the default value for root will be used, that is, 0. So, the expression will simplify to $acc[0] = i[0] + 0 = i[0]$ which is in R and has a certain value c. This value will be substituted in the expression for $acc[0]$. As it is now resolved, it will be moved from Unresolved to Resolved. The instantiation of root is analogous: $root[0] = if\ reset[0]\ then\ 0\ else\ acc[0]$, lets consider that $reset[0] = False$ so $root[0] = acc[0]$ and as $acc[0]$ is already resolved, so $root[0] = c$ and it is moved to R.

Then, at $n = 1$ the equation is instantiated again $acc[1] = i[1] + root[1-1|0] = i[1] + root[0]$. This time, the required streams for the evaluation of $acc[1]$ are $i[1]$ and $root[0]$(which is a correct position of the stream root because it is inside the trace limits $0 <= n < N$) and will be searched in Resolved. If the value of the pair $(stream, time)$ is found in Resolved, then it is used to replace all references in every expression in Unresolved. In this case the value of $root[0]$ will be used to substitute in the appropiate place in the expression resulting in $acc[1] = i[1] + c = e$. As it is resolved it will be moved to Resolved. The instantiation of root is the following: $root[1] = if\ reset[1]\ then\ 0\ else\ acc[1]$. Lets consider this time that $reset[1] = True$, then the expression is resolved to $root[0] = 0$ and moved to R. If the values of the needed timed streams were not found in Resolved, then the expression could not be simplified any further and would be left in Unresolved awaiting for a new tick in which the needed streams are in R (maybe coming from other nodes) so the expression can be simplified. □

2.6 Assumptions

Global clock We assume that there are hardware clocks available to all computing nodes which can be used for synchronizing the nodes. For example, $tick = 1ms$. Another option is to use a synchronous BUS to communicate the nodes, then the BUS will act as a clock or at least as a clock synchronizer. Either case there is some shared device that acts like a global clock. Thus, in each tick every node will have plenty of time to perform a collection of data Input, compute the neccessary values and to perform data Output, in the form of message passing.

Monitors Local monitors will be implemented by processes running in the monitored system nodes which will be assigned the monitoring of a subformula and reporting its results. There is a special monitor, the root node for the abstract syntax tree (AST), which will have the additional task of emiting the verdict (or result) of the formula to the user (or the corresponding system). In this thesis we

assume that monitors do not crash.

Distributing the formula Given a formula φ, it will be deconstructed syntactically into an abstract syntax tree (AST). With this tree and the knowledge of the monitored system topology, the formula will be spread among the nodes. As in [Basin et al., 2015].

Types and expressions As we use SRV, both input and output streams are strong statically typed as in Lola. Available types are described in Section 3.3.

Routing Table In this thesis we consider static non changing topologies. In such a topology, modelled as a connected graph, a routing table can be build such that it contains the shortest path between every pair of nodes. Therefore, the routing table at any monitor is a $Map\ Node_{dst} \longrightarrow Node_{next}$. Thus, if the monitor requires sending a message to a certain destination, the routing table will tell what is the next hop in the shortest path between the two monitors. Such a table can be computed statically by applying Dijkstra's algorithm, for example.

Monitor interaction with the system All the interaction between monitors and the system is done through the trace of streams. This trace must be generated by either instrumenting the system to get specific valuable events or collecting the events that the system generates normally.

Synchronous model of computation The synchronous model of computation consists on a sequence of rounds, where each round has stages that are performed by each computation node. These rounds are global to all the nodes of the system and determine their progress and establish how nodes communicate with their neighbours. These synchronization can be achieved using a global clock that is shared across the nodes or a common synchronous BUS that allows nodes to syncrhonize periodically. Each round or cycle has stages in which each monitor will perform the corresponding task: Read inputs, Compute values and Send messages.

Chapter 3

Decentralized Monitors

In this section we describe the solution to the decentralized SRV problem. Given a well-formed Lola specification, the decentralized algorithm that we present in this paper will compute a value for each output stream variable at each position. The main *correctness* criteria is that this output must be the same as the output computed in a centralized solution (that is, the semantics of Lola presented in the previuos section). We allow the decentralized process to compute an output at a different time (for example, due to pending communications).

Decentralized observation We assume that each variable relevant to the monitor formula is observed at just one computing node of the monitored system. Resembling the decentralization of sensors in real systems in different specialized hardware devices. Monitors deployed on those same computing nodes have immediate and free (cost = 0) access to the values of the input streams that are visible from that node. These observations correspond to the variables of the monitored system, which are intrinsically decentralized among the nodes. Values of variables that are not local to a node need to be communicated if necessary.

Message passing Messages considered in this work will arrive always at their destination exclusively(i.e. unicast communication) and there will not be duplicates nor delays in the transmition. A message sent at time t will be available at destination at time $t' = t + 1$. The messages can be of either one of these three types:

- Request: $req\{\sigma_t, V_{src}, V_{dest}\}$
- Response: $res\{\sigma_t, c, V_{src}, V_{dest}\}$
- Trigger: $trigger\{\sigma_t, V_{src}, V_{dest}\}$ See 3.5

Example 3. *Lets continue with the same example, with the following placement: Monitor1 is assigned the stream root and receives the input stream reset while acc will be computed by Monitor2 which also receives the input stream i. As Monitor1 needs the value of acc, it will send the message* $req\{acc[n], Mon1, Mon2\}$ *which can be read as "Monitor1 requests the value of* $acc[n]$ *at time n to Monitor2". Then at time* $n+1$ *Monitor2 receives this message and responds with the message* $res\{acc[n] = c, Mon2, Mon1\}$ *which analogously can be read as "Monitor2 responds* $acc[n] = c$ *to Monitor1". At last, at time* $n+2$ *Monitor1 receives the message with the value of* $acc[n] = c$*, stores this value in R and proceeds to simplify the equation for* $root[n]$*.*

Meanwhile at $n+1$ *Monitor2 has the expression* $acc[n+1] = i[n+1]+root[n]$ *so it needs the value of* $root[n]$ *and will send the message* $req\{root[n], Mon2, Mon1\}$ *which arrives at Monitor 1 in* $n+2$*. At that time the value of* $root[n]$ *is already in R, so Monitor1 sends the message* $res\{root[n] = c, Mon1, Mon2\}$ *which in turn arrives at* $n+3$ *at Monitor2. This value is stored in R and used to resolve the value of* $acc[n+1]$*. This process is repeated for all n in the trace length.*

This is an example of Lazy communication(see Section 3.4) without any simplifier (see Section 11). □

3.1 Goals of the solution

The goals of this solution are the following:

Correctness Decentralized monitoring should yield the same verdicts as the centralized solution for any given formula and input, so only non-functional requirements can differ. Formally: let $c(\varphi, \pi)$ be the solution of the centralized Lola outcome given a specification φ and a trace π, and let $d(\varphi, \pi)$ be the decentralized Lola version given the same specification φ and trace π, then $c(\varphi, \pi) = d(\varphi, \pi)$. Independently of the topology of the network which the decentralized version will compute on, the final verdicts should be the same: $c(\varphi, \pi) = d(\varphi, \pi)$. This assertion implies that both algorithms terminate and yield the same result.

Performance Runtime verification is said to be a lightweight verification tool and as such its impact on the monitored system performance should be kept as low as possible, particularly for online monitoring. We consider different aspects to evaluate the performance criteria.

Number of messages The number of messages is defined as $\sum m_{ij}$ where i is the identifier of the node and j is the identifier of the message sent by the i-th node. These messages with monitoring purposes traversing the network should be minimized.

Bandwidth Decentralized monitoring should minimize the network bandwidth usage as much as possible. Formally, $\sum_i payload_i$ where i represents the number of messages and $payload_i$ represent the number of bits of payload that the message i carries. We will use standard sizes such as 4 bytes for an integer, 8 bytes for a double, one bit for a boolean and for data structures we will count the number of elements and multiply by the size of each individual element and so forth, resulting in measuring the size in the total number of bits used.

CPU Decentralized monitoring should minimize the CPU usage of a given node, so that the monitoring overhead is kept low at each computing node. We want the computation to be as evenly distributed among the monitors as possible in order not to overload some monitors while other are idle. Formally, let s_e be the number of simplifications until a certain expression in Unresolved is resolved, let $mon_m = sum_{expression} s_e$ be the number of simplifications that a single monitor perform in a cycle and let $allCycle = \sum_{monitor} mon_m$ is the total number of simplifications performed in a cycle in all monitors, then for all m . $mon_m \approx \frac{allCycle}{numMonitors}$.

Verdict delay Decentralized monitoring should minimize the delay between the time all variables for a formula are present in the system and the time a verdict or a result for the formula is returned to the user. Formally, let t be the time at which every needed input to the specification s are present in the system, and let t_1 such that $t_1 \geq t$ be the time in which s is resolved to a value, that is a final verdict. We would like to minimise $t_1 - t$. Ideally, we would like this delay to be zero, meaning that the system emits a verdict in the same tick it has all the needed values. This ideal is met in centralized Lola for efficiently monitorable specifications. But in decentralized solutions this is not achievable as observations may be performed in remote nodes. Additionally, decentralized monitoring algorithms may impose additional delays.

3.2 Distributing monitors

In this section we analyze the core of the solution presented in this work. We use a network of monitors that cooperate in order to resolve the output of the monitors created from a given formula on an input trace. Intuitively, we distribute the formula

in subformulas that are assigned to monitors, which in turn are deployed in the nodes of the topology. The input trace is observed in multiple nodes. At each cycle, each monitor reads the input and the incoming messages, then with their already resolved subformulas, the monitor tries to simplify the unresolved subformulas assigned to it. After the simplification stabilizes, the resolved subformulas obtained are sent to the interested nodes in accordance to the communication strategies (see Section 3.4). Repeating this process leads to the correct evaluation of the formula over the input as it is proved in the Section 3.7. Formally, we will have a set of monitors $M = m_1, ..., m_m$, a topology $\mathcal{T}$, a specification φ and an input trace of events I. We want our solution to be a function $L : M \times \mathcal{T} \times \varphi \times I \longrightarrow type$ whose result is that of computing the specification in the monitors of the topology over an input trace of events. In the case of Boolean specifications, the output type will be Boolean, and for any other specification type, the output type will be the corresponding one.

Local Monitor A Local Monitor is a process running in a computation node that communicates with neighbouring local monitors of the system through message passing. No shared memory is assumed in our system. A monitor M can be modelled as $M : \langle I, Q, E, U, R, Pen, O, \mathcal{T}, \Delta, Tracelen, EvalStreams, Triggers \rangle$. A monitor receives the set of Input variables at time n in I from the trace of observations that are in the scope of this Monitor, of the form $i[n] = c$ for $i \in I$ and will store them in R. A monitor receives messages in Q from its neighbouring nodes accordingly to the messages described in Section 3. A monitor stores the subexpressions that are assigned to it in E. These expressions are instantiated and placed in U for every valid tick of the clock. These expressions have the form $\sigma[n] = \gamma[n]$ simplified up to the best knowledge of this node at the present tick. $\gamma[n]$ is an expression of the type of the stream that is not a constant. As we will see, when $\gamma[n]$ becomes a constant c, the equation $\sigma[n] = c$ is resolved and moved to R. A monitor stores resolved equations in R, of the form $\sigma[n] = c$ where c is a constant of the data type. This corresponds to values that the monitor has been able to resolve. A monitor stores requested streams and triggers in Pen, in order to respond them when they get resolved.

Definition 4 (Local expressions)**.** *Let $\sigma[n]$ be a local expression that is (part of) a formula. Let n be a specific time, then $\sigma[n]$ is either on U_t or in R_t (but not in both).*

Definition 5 (Global as composition of local)**.** *Let μ be the global state of the decentralized version of LOLA and ν_i the state of the local monitor M_i, then $\mu = \prod \nu_i$*

A monitor has the set of messages to be sent in the tick in O.

We consider static non changing connected topologies. The information about the system topology as an adjacency matrix will be stored in $\mathcal{T}$. $\mathcal{T}$ is specific for each node and relates any destination node (V_d) to next-hop node (V_n).

Fact 1 (Route Table and distance)**.** *For every V, V_d, let $RT_V(V_d) = [U_1, ..., U_k]$ the list of monitors in the shortest path from V to V_d*
then U_1 is the next-hop monitor, $U_k = V_d$ and $k = len(l)$ the length of the path where $l = U_1 : RT_{U_1}(V_d)$

Lemma 2 (Every message arrives)**.** *Let $m = (req, \sigma, V_s, V_d)$ or $m' = (res, \sigma, c, V_s, V_d) \in Q_v$ for some node V and time n.*
then $M \in Q_{V_{d_{t'}}}$ for some $t' = t + k$ where h is the height of σ in φ and $k = len(l)$ and $l = RT_v(V_d)$

Proof. By Induction on k:
Base case:$k = 0$
If the distance between the nodes is zero the list of the nodes of the path is empty $RT_V(V_d) = []$. Then $V = V_d \wedge m \in Q_V \wedge t' = t$
Induction Hypothesis (IH) This lemma holds when $m \in Q_{V_{dt'}} \wedge t' = t + k$, $k = len(RT_v(V_d))$
Proof for $k + 1 = len(RT_v(V_d))$, $RT_V(V_d) = V_y : l$ *where* $l = [U_1, ..., U_k]$
then $m \in Q_{V_y}[t'] \; t' = t + 1$ By IH
$RT_{V_y}(V_d) = l$ By Fact1
then $m_2 \in Q_{V_d}[t"] \; t" = t' + k = t + k + 1$ By IH

□

The syntactical decomposition of the formula φ in subformulas σ and their assignment to nodes is stored in Δ. This assignment is made statically at the setup of the monitors and must ensure that for every σ it is deployed in some node.

Fact 2 (Deployment function)**.** *For every s subformula contained in φ there is a Node tasked of resolving it. This relation is provided by the function $\Delta : \Delta(\sigma_i) = V_j$ where $i, j \in \mathbb{N}$*

Fact 3 (Deployment independence)**.** *Using a different Δ function does not change the output of the formula φ but it could change its efficiency.*

Information about all the nodes that require the result of any given subformula is stored in Γ which is useful for the Eval mode of operation (see Section 3.4).

Example 4. *Lets consider Example 4 but using the Eval communication strategy (see Section 3.4). Once Monitor2 resolves the value of acc it will communicate it to those nodes which require it, namely Monitor1. Analogously, Monitor1 will communicate the values of root to Monitor1 as soon as they are resolved. This is possible because of $\Gamma = \{acc \rightarrow [Monitor1], root \rightarrow [Monitor2]\}$.* □

The node will know a priori the length of the input, in order to decide whether it must instantiate equations or use their default values in those cases where the times

shifts get out of the trace boundaries. The node needs to know which streams should be treated as Eval in order to use this communication strategy (see Section 3.4). The node will need to know which streams should be treated as triggers in order to allow the system to behave as described in Section 3.5.

3.3 Data Domains

We use many-sorted first order logic to describe data domains.

A signature $\Sigma : \langle \mathcal{S}, \mathcal{F} \rangle$ consist on a finite collection of sorts $\mathcal{S}$, and function symbols $\mathcal{F}$ (where each argument of a function has a sort, and the resulting term also has a sort).

A simple theory, *Booleans*, has only one sort[1], *Bool*, two constants `true` and `false`, binary functions $\wedge$ and $\vee$, unary function $\neg$, etc. A more sophisticated signature is *Naturals* that consists of two sorts (*Nat* and *Bool*), with constant symbols `0`, `1`, `2`... of sort *Nat*, binary symbols $+$, $*$, etc (of sort $Nat \times Nat \to Nat$) as well as predicates $<$, $\leq$, etc of sort $Nat \times Nat \to Bool$, with their usual interpretation. All theories have equality and are typically (e.g. *Naturals*, *Booleans*, *Queues*, *Stacks*, etc) equipped with a ternary symbol `if` $\cdot$ `then` $\cdot$ `else`$\cdot$. In the case of *Naturals*, the `if` $\cdot$ `then` $\cdot$ `else`$\cdot$ symbol has sort $Bool \times Nat \times Nat \to Nat$.

We use many-sorted first order logic to describe data domains. A many-sorted first-order signature is a tuple $\Sigma : \langle \mathcal{S}, \mathcal{F} \rangle$ where $\mathcal{S}$ is a finite collection of distinct sorts, and $\mathcal{F}$ is a collection of function symbols, each associated with a first-order *type*. For example, a functional symbol `f` that describes a function that takes values from S_1 and S_2 and returns a value in S_{res} has type $S_1 \times S_2 \to S_{\text{res}}$, where S_1, S_2 and S_{res} are sorts in $\mathcal{S}$. The arity of a function is the number of its arguments, each of the appropriate sort. In the example above, `f` has arity 2.

Our theories are interpreted, so each sort S is associated with a domain D_S (a concrete set of values), and each function symbol `f` is interpreted as a total computable function f, with the given arity and that produces values of the domain of the result given elements of the arguments' domains. We omit the sort S when it is clear from the context.

We will use *stream variables* with an associated sort, but from the point of view of the theories, these stream variables are atoms. As usual, given a set of sorted atoms A and a theory, terms are the smallest set containing A and closed under the use of function symbol in the theory as a constructor (respecting sorts).

[1]We use sort and type interchangeably in the rest of the paper.

3.4 Communication Strategies

In this section we explain the different strategies about how partial results are communicated among different nodes to obtain the final verdict and what are their advantages and disadvantages. Lets first establish when we consider a specification and a trace to meet what we call best and worst scenarios. Worst case is such that the whole expression must be processed for computing a verdict, that is, as if there were no simplifier at all and every subexpression deployed in every node must be computed in order to obtain the final verdict. Best case is that when for computing a verdict the node that produces the verdict (root node for the AST of the formula) has all the variables needed to resolve the whole equation and consequently, move it to R. In this case no other nodes are needed nor other messages containing partial results.

Eval evaluation As soon as an expression is resolved, its result is sent to those nodes that need it. In this evaluation strategy no request-type message will ever be sent. This strategy minimizes delay and messages sent in the worst case (compared to lazy) but consumes more bandwidth in the best case (compared to lazy). As a rule of thumb, Eval evaluation is suitable when the probability of needing to compute the formula is high.

Example 5. *Considering example 3 but with the Eval communication strategy, as soon as a Monitor2 resolves the value of* $acc[t]$ *the message* $res\{acc[t] = c, Monitor2, Monitor1\}$ *will be sent without the need of a request message. Analogously, as soon as Monitor1 resolves* $root[t]$ *the message* $res\{root[t] = c, Monitor1, Monitor2\}$ *will be sent.* □

Lazy evaluation In this strategy whenever a subexpression is needed a request-type message will be sent to the appropiate node in order to perform the actions needed to obtain the result of that subexpression and emit a response. Therefore, no expression result is ever transmitted over the network unless requested. This ensures that only strictly needed messages (considering simplification rules) travel over the network. On the one hand in the best case the root node emits the verdict with 0 delay and sending a total of 0 messages. On the other hand, in the worst case for any subexpression there will be two messages sent: request and response. That is, just the double than using Eval strategy.

Example 6. *Considering the example 3, when we considered that* $reset[t] = False$*, in this case if we have a simplifier the expression* $root[0] = if\ reset[0]\ then\ 0\ else\ acc[0]$ *can be resolved to* $root[0] = 0$*. Therefore there is no need to send a request message nor the corresponding response in order to get a value that is in fact not needed. This is an example of a best case scenario.* □

Hybrid evaluation Analyzing the subformulas of the specification one could find some subexpressions that will be needed no matter the input so the monitor tasked with that subexpression could treat them as Eval, and some other subexpressions that may or may not be needed depending on the input, so the monitor tasked with that subexpression could treat them as Lazy. This way, an intermediate solution could be achieved in which just the necessary messages are sent and the delay is kept at a minimum, obtaining the benefits of both strategies.

Example 7. *Considering our example the optimal way of configuring it is to treat the root stream as Eval and the acc stream as Lazy. This is due to the fact that acc will always need the value of root independetly of the inputs whereas the stream root only needs acc in some cases.* □

3.5 Computation duration

Monitors can compute in the following ways that differ in when to halt and emit the verdict.

Stream Request The monitors run until they resolve **all** requested streams in Pen in every node.

Example 8. *Consider a trace of 100 elements and the specification of our example 1, if we want the monitors to compute the specification at each point in time, then we will request* $root[0], ..., root[99]$*. Thus, the monitors will continue computing until they obtain all the values of* $root[t]$ *for every* t*. Depending on the trace and the specification itself, this situation may arise in a time before, at or after 99. This is how we could compute a specification at every point of a given trace.* □

Trigger The monitors run until they resolve **any** of the Trigger streams and emit it as the verdict.

Combination The two modes above can be combined in the following way: independently of whether there are remaining requested streams or other triggers in *Pen* in any monitor, as soon as one Trigger in the whole topology is resolved, the monitors will halt yielding the value of that Trigger as final verdict.

3.6 Local Algorithm: Operational Semantics

Every node V_x will execute Algorithm 1.

Incoming messages processing Lines 1-9 deal with incoming message processing. In this phase the forwarding of messages ocurrs. Requests and Triggers are stored in Pen. In addition, every response is stored in R.

Reading Input Line 10 all observations are received as inputs for the monitors, so they are added to the Resolved set of equations.

Instantiation of equations Line 11 describes how the subformulas assigned to the node are instantiated and added to U.

Simplification Line 12 shows how the equations in U are simplified with the knowledge in R. Those equations that get resolved will be moved from U to R. See section 3.7.

Send responses Lines 13-22 show how responses are sent for the newly resolved equations considering the communication strategies described in 3.4: send Lazy streams that were in Pen(received a request previously) send Eval streams to all those nodes that require it and send triggers.

Send and record requests Lines 23-26 show how requests are sent for those substreams that are treated as Lazy and are also in Pen. Also, remember already requested streams in order not to duplicate requests.

Theorem 1 (Convergence)**.** *Every request produces a response and every* $\sigma[n]$ *is eventually resolved and every time a response is sent then* $\sigma[n]$ *is resolved to c,* $(\sigma[n], c) \in R_v$
Formally:
Let $s[n]$ *be a timed stream and let* $V = \Delta(s)$ *and* V_{src} *be an arbitrary node and*

(1) *let* $req\{\sigma[n], V_{src}, V\}$ *be a msg in state* Σ_t
then, for some $t' > t$ *there is a c and a msg* $res\{\sigma[n], c, V, V_{src}\}$ *in state* $\Sigma_{t'}$

(2) *Also, for every t" for which* $\sigma[n] \in U_V$ *there is* $t''' > t$*" for which* $\sigma[n] \in R_V$
The proof will be done by structural induction, thus, a well-ordering is defined over ρ*.*

Algorithm 1 Local algorithm for monitor V_x at time n

1: **for all** $m \in Q_t$ **do** ▷ Process incoming messages
2: $\quad Q_t \leftarrow Q_t / m$
3: $\quad$ **if** $m.dst \neq V_x$ **then**
4: $\quad\quad route(m)$;
5: $\quad$ **else if** $m.dst = V_x$ **then**
6: $\quad\quad$ **if** $m.type = req \vee m.type = trigger$ **then**
7: $\quad\quad\quad Pen_t \leftarrow Pen_t + m$
8: $\quad\quad$ **else if** $m.type = res$ **then**
9: $\quad\quad\quad R_t \leftarrow R_t + (m.stream[n], m.res)$
10: $R_t \leftarrow R_t + (i[n],\ val(i[n]))$ for all $i[n] \in I_t$ ▷ Read input
11: $U_t \leftarrow U_t + (\sigma[n],\ \varepsilon(\sigma[n]))$ for all $\sigma\ .\ \Delta(\sigma) = V_x$ ▷ Generate equations
12: $U, R \cup newR \leftarrow Simplify\ (U,\ R)$ ▷ Substitute & Simplify
13: $newEval \leftarrow newR \cap Eval$
14: **for all** $r \in newEval$ **do** ▷ Send Eval responses
15: $\quad$ **for all** $V_j \in \Gamma(r)$ **do**
16: $\quad\quad route(res\{r.str, r.val, V_x, V_y\})$
17: **for all** $m \in Pen$ **do**
18: $\quad$ **if** $m.str \in R$ **then**
19: $\quad\quad$ **if** $m.type = Lazy$ **then** ▷ Lazy streams
20: $\quad\quad\quad Pen_t \leftarrow Pen_t\ /\ m; route(res\{r.str, r.val, V_x, m.src\})$
21: $\quad\quad$ **else if** $m.type = Trigger$ **then** ▷ Trigger streams
22: $\quad\quad\quad Pen_t \leftarrow Pen_t\ /\ m; trigger(res\{r.str, r.val, V_x, m.src\})$
23: $timedSubstreams \leftarrow substr(\varphi[n]).\varphi[n] \in Pen_t \cap Lazy \wedge \sigma_i \notin Requested$
24: **for all** $substream \in timedSubstreams$ **do** ▷ Request needed subexpressions
25: $\quad route(req\{substream, V_x, V_y\})$
26: $\quad Requested \leftarrow Requested + substream$ ▷ Remember requested streams

Proof. By induction on ρ based on previous lemmas and the lines of the algorithm.
Base case Let us show that (1) holds for inputs:
Let $req\{i_i[n-x_i],\ V_s,\ V\}\ \in\ Q_V[t']$ and $\Delta(i_i) = V,\ x_i\ \in\ \mathbb{Z}$By Lemma 2
$\rho(i_i[n]) = 0$ As it is an input, $(i_i[n-x_i], v_i)\ \in\ R_V[t"]\, t" = t - x_i$ By line 10
Then $res\{i_i[n-x_i],\ v_i,\ V,\ V_s\}$ will be sent by line 20
Then $res\{i_i[n-x_i],\ v_i,\ V,\ V_s\}\ \in\ Q_{V_s}[t''']$ By Lemma 2

Proof (2) holds for $\rho(\sigma_i[n]) = 0$:
$\sigma[n]\ =\ f\ (\sigma[n-y|d],\ i_1[n-x_1|d], ...,\ i_i[n-x_i|d],\ r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j])$
As $\rho(\sigma[n]) = 0$ then $\rho(r_1) = ... = \rho(r_j[n-k_j|d_j]) = 0$
But $\sigma[n]$ can only depend on subexpressions $r_1, ..., r_j$ such that
$r_1[n-k_1|d_1], ..,\ r_j[n-k_j|d_j] < \sigma[n]$ in the given well-founded order.
Thus, they are either inputs or streams whose all subexpressions are simplified to default values, to comply with the order.
Those that will be simplified are those that meet $t + k < 0\ or\ t + k > N$ Then
$r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j] = d_1, ..., d_j$ By line 12
Note that V need not send messages to know the default values $d_1, ..., d_j$
If an input it is resolved in V
$i_1[n-x_1], ...,\ i_i[n-x_i]\ \in\ R_v[t"]$ by line 10
If it is resolved to default values $\sigma[n]$ is resolved, so:
$\sigma[n]\ \in\ R_V[t"]$ By line 12
(2) holds.

Proof (1) holds based on (2) holds:
let $msg = req\{\sigma[n], V, V_s\},\ \Delta(\sigma) = V_s,\ msg\ \in\ Q_{V_s}[t'],\ t' <= t + d$ By Lemma 2
and $\rho(\sigma) = 0$
Then $\sigma[n]$ is resolved at some time in the future
$\sigma[n]\ \in R_{V_s}[t"]t" > t'$ By (2)
Then a response is sent back to $V\ res\{\sigma[n], v, V_s, V\}$ By line 20
which is received at $t'''\ res\{\sigma[n], v, V_s, V\} \in Q_V[t'''],\ t''' <= t" + d$ By Lemma 2
(1) and (2) hold for base case.

Induction Hypothesis (IH) Assuming the theorem holds for all $\rho(r[n]) < \rho(\sigma[n])$ for an arbitrary n
Proof of (2) assuming (1) holds for the IH
let $\sigma[n]\ =\ f\ (\sigma[n-y],\ i_1[n-x_1], ...,\ i_i[n-x_i],\ r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j])$
And considering
$x_{1..i}\ \in\ \mathbb{Z}$, $k_{1..j}\ \in\ \mathbb{Z}$, $y\ \in\ \mathbb{N}$ and $\rho(r_{1..j}) < \rho(\sigma[n])$
Then requests are sent to the corresponding nodes
$req\{r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j], V, V_{1..j}\}$ By line 25
Those requests produce responses that return
$res\{r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j], c_1, ..., c_j, V_{1..j}, V\}\ \in\ Q_V[t"]$ by (1)
So, both subexpressions and inputs are resolved waiting enough:
$(r_{1..j}, c_{1..j})\ \in\ R_V[t"]$

$(i_{1..i}, v_{1..i}) \in R_V[t'''],\ t''' = t + max_{i=1..i}(t - x_i)$
So, $\sigma[n]$ is resolved eventually in the future
$\sigma[n] \in R_V[t^{IV}],\ t^{IV} = max(t", t''') = max(t", t + max_{i=1..i}(t - x_i))$
This proves that(2) holds assuming (1).

Proof (1) assuming (2)
Let $msg = req\{\sigma[n], V, V_s\} \in Q_{V_s}$ By Lemma 2
let $\sigma[n] = f\ (\sigma[n-y],\ i_1[n-x_1], ...,\ i_i[n-x_i],\ r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j])$
then $\sigma[n]$ is eventually resolved
$(\sigma[n], c) \in R_{V_s}[t'], t' > t$ By (2)
then a response can be emitted
$res\{\sigma[n], c, V, V_s\}$ By line 20
at last, the response arrives at its destination
$res\{\sigma[n], c, V, V_s\} \in Q_V[t"], t" <= t' + d$ By Lemma 2
We have proved that (1) and (2) hold for the inductive case. □

Fact 4. *Convergence provides a value for every stream in a certain time t'.*

3.7 Simplification

In this section we discuss the use of simplifiers in our algorithm. We describe the properties that we desire for our simplifiers and provide formal proofs of their correctness. Simplifiers are important in practice because they provide performance enhancements. Simplifiers are based on the principle that sometimes an expression can be correctly evaluated without knowing all their parameters. The simplification algorithm is based in the simplification used in Lola [D'Angelo et al., 2005]. We are interested in showing that the decentralized algorithm that we propose (using simplifiers) is correct. In order to do this we will proceed as follows: we will show the correctness of the algorithm ignoring the presence of simplifiers, showing a proof by comparing with the output of Lola without simplifiers. Then we prove that the simplifiers preserve the equivalence under evaluation and prove that the centralized Lola without simplifiers is correct by comparing to Lola (which uses simplifiers). At the end we will show that Lola produces the same outputs that our proposed algorithm, that is decentralized and uses simplifiers. By transitivity we can also conclude that our algorithm is equivalent (produces the same outputs) as the decentralized algorithm without simplifiers.

Definition 6 (Strict Simplification)**.** *Let us define a special simplification function which will not simplify anything until it has all the parameters of the stream expression and then evaluates it.*
It meets both simplification properties, namely P1 and P2.
P1: is trivial
P2: the partial evaluation occurs only when $Y = X$ so $X \setminus Y is \varnothing$

Algorithm 2 Simplification algorithm

1: **for all** $v\{r[n] \leftarrow a_i\} \in R_V new$ **do**
2: $\quad \varepsilon_{new} \leftarrow simpl(\varepsilon, v)$
3: $\quad$ **if** $\varepsilon_{new}$is val **then**
4: $\quad\quad U' \leftarrow U - \{(\sigma[n], \varepsilon)\}$
5: $\quad\quad R' \leftarrow R + \{(\sigma[n], \varepsilon_{new})\}$
6: $\quad$ **else**
7: $\quad\quad U' \leftarrow (U - \{(\sigma[n], \varepsilon)\} + \{(\sigma[n], \varepsilon_{new})\})$

Then the partial application (with all the parameters) yields the same result as applying the function with all its parameters.

Theorem 2 (Correctness of decentralized and centralized with strict simplification)**.** *The centralized and decentralized algorithms using the Strict Simplification 6 yield the same outputs for any input. Let φ be a well defined specification and I an input of size N. Let O_{cst} be the outputs of the centralized strict and O_{dst} the outputs of the decentralized strict with the topology $\mathcal{T}$. Then $O_{cst} = O_{dst}$.*

Proof. Let E be the evaluation graph of φ, let I be the input and let ρ be the evaluation distance of E. We show by induction on $\rho(\sigma[n])$ that $O_{cst}(\sigma[n]) = O_{dst}(\sigma[n])$
Base case$\rho(\sigma[n]) = 0$
Let $\sigma[n] = f\ (\sigma[n-y|d],\ i_1[n-x_1|d], ...,\ i_i[n-x_i|d],\ r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j])$ be the stream expression
And this expression is the same for the centralized and decentralized algorithms. As $\rho(\sigma[n]) = 0$ then $\rho(r_1) = ... = \rho(r_j[n-k_j|d_j]) = 0$
But $\sigma[n]$ can only depend on subexpressions $r_1, ..., r_j$ such that
$r_1[n-k_1|d_1], ..,\ r_j[n-k_j|d_j] < \sigma[n]$ in the given well-founded order.
Thus, they are either inputs or streams whose all subexpressions are simplified to default values, to comply with the order.
Those that will be simplified are those that meet $t + k < 0$ *or* $t + k > N$ Then $r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j] = d_1, ..., d_j$ By line 12
Note that V need not send messages to know the default values $d_1, ..., d_j$

If an input:
if it will be resolved in a remote node, a request is sent by line 25
that request will arrive at its destination by Lemma 2
then it is processed and as it is an input at some time t' it will be resolved
then a response can be emitted with the value in R by line 20
then the response arrives at V with the pair $(i[n], c)$ by Lemma 2
so it is resolved in V
Alternatively, the input is received at some time in V.

Either case :
$i_1[n-x_1], ...,\ i_i[n-x_i] \in R_v[t"]$ with their respective values by line 10 or by

Convergence (Theorem 1)
So, by applying all its parameters, $\sigma[n]$ is resolved in the decentralized strict version: $(\sigma[n], c) \in R_{V_d}[t"]$ By line 12

On the other hand the centralized with strict algorithm will resolve $\sigma[n]$ at instant t' where $t' = max_i(t - x_i)$. At said moment both the default values $d_1, ..., d_j$ and the inputs $(i_1, c_1), ..., (i_i, c_i)$ are available to evaluate the expression of $\sigma[n]$. $(\sigma[n], c) \in R_{V_d}[t"]$ By line 12
Because applying the same expression (function) over the same arguments yield the same result.
IH :$\rho(r[n]) < \rho(\sigma[n])$ Assuming that for $\rho(r[n]) < \rho(\sigma[n])$ the value of $r[n]_{cst}$ is equal to $r[n]_{dst}$ for an arbitrary t
Prove $\sigma[n]_{cst} = \sigma[n]_{dst}$ assuming their subexpressions are resolved and equal for all r s.t. $\rho(r[n]) < \rho(\sigma[n])$
let $\sigma[n] = f\ (\sigma[n-y],\ i_1[n-x_1], ...,\ i_i[n-x_i],\ r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j])$
And considering $x_{1..i} \in \mathbb{Z}$, $k_{1..j} \in \mathbb{Z}$, $y \in \mathbb{N}$, $\rho(r_{1..j}) < \rho(\sigma[n])$
all params are resolved to the same values as in the centralized strict algorithm by IH
So, $\sigma[n]$ is resolved eventually in the future to a constant c by Convergence (Theorem 1)
$\sigma[n] \in R_V[t^{IV}],\ t^{IV} = max(t", t"') = max(t", t + max_{i=1..i}(t - x_i))$

On the other hand the centralized version will resolve it by evaluating the expression of $\sigma[n]$ applying all its parameters, both inputs and subexpressions. Thus, $\sigma[n] \in R_V$. As the inputs for the expression are equal to those of the decentralized version, and all the subexpressions with lesser ρ are resolved to the same values in both algorithms, we can conclude that the value of $\sigma[n]$ obtained by the centralized strict and decentralized strict algorithms are equal for every $\sigma[n]$. □

Definition 7 (Eval)**.** *$Eval :: Term(\varnothing) \longrightarrow Const$ is given/fixed in each theory. Simply bottom eval of each function of the corresponding datatype see Section 3.3.*

Definition 8 (Val substitution)**.** *υ Set of substitutions of the form $x \leftarrow c$ where c is a constant for some subset of the variables of the streams.*

Definition 9 (Function simpl does not introduce variables)**.** *simpl is a function from a term to another term which does not add variables to the term so that the new term has less or the same variables as the original term.*
$simpl : Term \longrightarrow Term$
$simpl(t) = t'$ where $Vars(t') \subseteq Vars(t)$

Definition 10 (Term equivalence)**.** *Let t and t' be two terms. We say that t and t' are "equivalent under evaluation", and we write $t \underset{Eval}{=} t'$ See definition 7. whenever $\forall \upsilon\ .\ Vars(t) \cup Vars(t') \in \upsilon\ .\ Eval(t\upsilon) = Eval(t'\upsilon)$*

Definition 11 (Admissible simplifiers)**.** *We capture "admissible" simplifiers as follows:*

1. *Simpl does not add variables (Definition 9)*

2. $t \underset{Eval}{=} simpl(t)$ *for every* t *(Definition 10)*

Fact 5 (Simplification preserves equivalence)**.**

$$\text{if } t \underset{Eval}{=} t' \text{ then } t \underset{Eval}{=} simpl(t') \tag{3.1}$$

Note 1. *It is easy to show that* $\underset{Eval}{=}$ *is reflexive, simmetric and transitive.*
Note that if $t \underset{Eval}{=} t'$ *then* $t \underset{Eval}{=} simpl(t')$
by reflexivity $(t \underset{Eval}{=} t)$
and transitivity $t \underset{Eval}{=} t'$ *and* $t' \underset{Eval}{=} simpl(t')$ *then* $t \underset{Eval}{=} simpl(t')$

We are interested in showing that the chain of substitution followed by simplification preserves the equivalence (Lemma 4 below). We need first an auxiliary lemma.

Lemma 3. *For every* t, υ_1 *and* υ_2
$t\upsilon_1\upsilon_2 \underset{Eval}{=} simpl(simpl(t\upsilon_1)\upsilon_2)$

Proof. We first show that

$$\text{if } t \underset{Eval}{=} t' \text{ then for every } \upsilon \text{ such that } t\upsilon \underset{Eval}{=} t'\upsilon \tag{3.2}$$

Let υ_2 be an arbitrary substitution from $Val(Vars(t\upsilon) \cup Vars(t'\upsilon))$.
That is a substitution of the remaining variables after substituting υ
then $(\upsilon\upsilon_2)$ is a substitution from $Val(Vars(t) \cup Vars(t'))$, all variables present in t or t'.
Since $t \underset{Eval}{=} t'$, then
$Eval(t\upsilon\upsilon_2) = Eval(t'\upsilon\upsilon_2)$, or in other words
$Eval((t\upsilon)\upsilon_2) = Eval((t'\upsilon)\upsilon_2)$ this implies $t\upsilon \underset{Eval}{=} t'\upsilon$. Hence, (3.2) holds.

Now, first $t\upsilon_1 \underset{Eval}{=} t\upsilon_1$ by reflexivity.
Then $t\upsilon_1 \underset{Eval}{=} simpl\ t\upsilon_1$ by definition of admissible simplifiers
It follows that $t\upsilon_1\upsilon_2 \underset{Eval}{=} (simpl t\upsilon_1)\upsilon_2$.
Then by Fact 3.1 $t\upsilon_1\upsilon_2 \underset{Eval}{=} simpl((simpl\ t\upsilon_1)\upsilon_2)$ as desired. □

Lemma 4 (Simplification distributive property)**.** *Let t be a term, let $\upsilon_0, ..., \upsilon_n$ be substitutions from $Val(Vars(t))$,*
Let $t_0, ..., t_n$ be defined as follows: $t_0 = t,\ t_{i+1} = simpl(t_i\upsilon_i)$
then
$t_n \underset{Eval}{=} simpl(t\upsilon_1...\upsilon_n)$
then $t_n \underset{Eval}{=} simpl(t\upsilon_0...\upsilon_{n-1})$

Proof. We show by induction on n that $t_n \underset{Eval}{=} t\upsilon_0, ..., \upsilon_{n-1}$.
IH: $t_{n-1} \underset{Eval}{=} t\upsilon_0, ..., \upsilon_{n-2}$
Now by "simpl preserves equivalence" (Lemma 3.1) and by "substitution to both members preserves equivalence" (Lemma 3.2) then $simpl(t_{n-1}\upsilon_{n-1}) \underset{Eval}{=} t\upsilon_0, ..., \upsilon_{n-2}\upsilon_{n-1}$
Now $t_n = simpl(t_{n-1}\upsilon_{n-1})$ and by reflexivity $t_n \underset{Eval}{=} simpl(t_{n-1}\upsilon_{n-1})$
Hence, $t_n \underset{Eval}{=} t\upsilon_0, ..., \upsilon_{n-1}$ follows. Now applying "simpl preserves equivalence" we obtain the desired result. □

Lemma 5. *Let $t[x_1, ..., x_n]$ be a term, and simpl a simplifier. Let υ be a partial substitution of X such that $simpl(t\upsilon) = c$. Then, for every $\upsilon_{rest}, simpl(t\upsilon\upsilon_{rest}) = c$*
In particular if $t\upsilon\upsilon_{rest}$ is ground, $Eval(t\upsilon\upsilon_{rest}) = c$

Proof. $t\upsilon \underset{Eval}{=} simpl(t\upsilon) \underset{Eval}{=} c$
Consequently, $t\upsilon\upsilon_{rest} \underset{Eval}{=} simpl(t\upsilon\upsilon_{rest}) \underset{Eval}{=} c\upsilon_{rest}$, and $c\upsilon_{rest} \underset{Eval}{=} c$
Finally, since $t\upsilon\upsilon_{rest}$ is ground $Eval(t\upsilon\upsilon_{rest}) = d$, which satisfies $d \underset{Eval}{=} c$ □

Theorem 3 (Theorem correctness of centralized with simplification and centralized strict)**.** *The centralized with evaluation and the centralized with simplification algorithms yield the same outputs for any input. Let φ be a well defined specification and I an input of size N. Let O_{cst} be the outputs of the centralized strict and O_{cs} the outputs of the centralized with simplification. Then $O_{cst} = O_{cs}$.*

Proof. Let E be the evaluation graph of φ and I and let ρ be evaluation distance of E. We show by induction on $\rho(\sigma[n])$ that for all $O_{ce}(\sigma[n]) = O_{cs}(\sigma[n])$
Base case$[\rho = 0]$ let $\sigma[n] = f\ (\sigma[n - y|d],\ i_1[n - x_1|d], ...,\ i_i[n - x_i|d],\ r_1[n - k_1|d_1], ...,\ r_j[n - k_j|d_j])$ be the stream expression
And this expression is the same for both algorithms.
As $\rho(\sigma[n]) = 0$ then $\rho(r_1) = ... = \rho(r_j[n - k_j|d_j]) = 0$
But $\sigma[n]$ can only depend on subexpressions $r_1, ..., r_j$ such that
$r_1[n - k_1|d_1], ..,\ r_j[n - k_j|d_j] < \sigma[n]$ in the given well-founded order.
Thus, they are either inputs or streams whose all subexpressions are simplified to default values, to comply with the order.
Those that will be simplified are those that meet $t + k < 0$ *or* $t + k > N$
Then $r_1[n - k_1|d_1], ...,\ r_j[n - k_j|d_j] = d_1, ..., d_j$ By line 12

Note that V need not send messages to know the default values $d_1, ..., d_j$
If an input it is received at some time in V.
$i_1[n - x_1], ..., \ i_i[n - x_i] \ \in \ R_v[t"]$
As both inputs and the default values are resolved, so by applying all its parameters to its expression, $\sigma[n]$ is resolved in the centralized with evaluation to a value c.
$(\sigma[n], c) \ \in \ R_{V_d}[t"]$ By line 12

But, at some time t''' such that $t''' \leq t"$ the centralized with simplification simplifies the expression of $\sigma[n]$ to its value d.
Then by lemma 5 we could substitute the remaining parameters v_{rest} in the expression on $\sigma[n]$ at t''' and evaluate the expression, obtaining the same output c, thus we can conclude that $c = d$.

IH Assuming centralized with evaluation and centralized with simplification yield the same outputs for $\rho(r[n]) < \rho(\sigma[n])$ for an arbitrary n
Prove $\sigma[n]_{cs} \ = \ \sigma[n]_{ce}$ assuming their subexpressions are resolved and equal for $\rho(r[n]) < \rho(\sigma[n]$
let $\sigma[n] \ = \ f \ (\sigma[n - y], \ i_1[n - x_1], ..., \ i_i[n - x_i], \ r_1[n - k_1|d_1], ..., \ r_j[n - k_j|d_j])$
And considering $x_{1..i} \ \in \ \mathbb{Z}$, $k_{1..j} \ \in \ \mathbb{Z}$, $y \ \in \ \mathbb{N}$ and $\rho(r_{1..j}) < \rho(\sigma[n])$
So, $\sigma[n]$ is resolved eventually in the future to a constant c
$\sigma[n] \ \in \ R_V[t^{IV}], \ t^{IV} = max(t", t''') = max(t", t + max_{i=1..i}(t - x_i))$
On the one hand the centralized version will resolve it by evaluating the expression of $\sigma[n]$ applying all its parameters, both inputs and subexpressions. As follows : $Eval(t, s1, ..., sn) = c$ where $s_1, ..., s_n$ are pairs of substitutions of the form $(stream, const)$ Thus, $\sigma[n] \ \in \ R_V$,

Nevertheless, the centralized with simplification algorithm may obtain the output of the expression in some time $t' \leq t$ by applying some of the parameters of the expression. As follows: $simpl(tv) = d$. Then by Lemma 5 we could substitute the remaining parameters v_{rest} in the expression on $\sigma[n]$ at t''' and evaluate the expression, obtaining the same output c, thus we can conclude that $c = d$.

□

Theorem 4 (Theorem correctness of decentralized with simplification and decentralized with strict)**.** *The decentralized strict and the decentralized with simplification algorithms yield the same outputs for any input. Let φ be a well defined specification and I an input of size N and let $\mathcal{T}$ be the topology for both algorithms. Let O_{dst} be the outputs of the decentralized with evaluation and O_{ds} the outputs of the decentralized with simplification. Then $O_{dst} = O_{ds}$.*

Proof. Let E be the evaluation graph of φ and I and let ρ be evaluation distance of E. We show by induction on $\rho(\sigma[n])$ that $O_{ds}(\sigma[n]) = O_{dst}(\sigma[n])$
Base case$[\rho = 0]$ let $\sigma[n] \ = \ f \ (\sigma[n - y|d], \ i_1[n - x_1|d], ..., \ i_i[n - x_i|d], \ r_1[n -$

$k_1|d_1], ...,\ r_j[n-k_j|d_j])$ be the stream expression
And this expression is the same for the decentralized with simplification and decentralized strict algorithms.
As $\rho(\sigma[n]) = 0$ then $\rho(r_1) = ... = \rho(r_j[n-k_j|d_j]) = 0$
But $\sigma[n]$ can only depend on subexpressions $r_1, ..., r_j$ such that
$r_1[n-k_1|d_1], ..,\ r_j[n-k_j|d_j] < \sigma[n]$ in the given well-founded order.
Thus, they are either inputs or streams whose all subexpressions are simplified to default values, to comply with the order.
Those that will be simplified are those that meet $t+k<0$ *or* $t+k>N$
Then $r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j] = d_1, ..., d_j$ By line 12
Note that V need not send messages to know the default values $d_1, ..., d_j$
If an input it is resolved in V
if it will be resolved in a remote node then a request is sent by line 25
that request will arrive at its destination by 2
then it is processed and as it is an input at some time t' it will be resolved
then a response can be emitted with the value in R by line 20
then the response arrives at V with the pair $(i[n], c)$
so it is resolved in V
Alternatively, the input is received at some time in V.
Either case :
$i_1[n-x_1], ...,\ i_i[n-x_i]\ \in\ R_v[t"]$ by line 10
If it is resolved to default values $\sigma[n]$ is resolved, so by applying all its parameters, $\sigma[n]$ is resolved in decentralized with evaluation as follows $Eval(t\upsilon) = c$ where υ are pairs of substitutions of the form $(stream, const)$ Then $(\sigma[n], c)\ \in\ R_{V_d}[t"]$ By line 12

Nevertheless, the decentralized with simplification algorithm may obtain the output of the expression in some time $t' \leq t$ by applying some of the parameters of the expression. As follows: $simpl(t\upsilon) = d$. Then by lemma 5 we could substitute the remaining parameters υ_{rest} in the expression on $\sigma[n]$ at t''' and evaluate the expression, obtaining the same output c, thus we can conclude that $c = d$.

IH :$\rho(r[n]) < \rho(\sigma[n])$ Prove $\sigma[n]_d = \sigma[n]_c$ assuming their subexpressions are resolved and equal for $\rho(r[n]) < \rho(\sigma[n]$
let $\sigma[n]\ =\ f\ (\sigma[n-y],\ i_1[n-x_1], ...,\ i_i[n-x_i],\ r_1[n-k_1|d_1], ...,\ r_j[n-k_j|d_j])$
And considering $x_{1..i}\ \in\ \mathbb{Z}, k_{1..j}\ \in\ \mathbb{Z}, y\ \in\ \mathbb{N}, \rho(r_{1..j}) < \rho(\sigma[n])$ $r_{1..j}$ will be resolved eventually to the same values that in the decentralized with simplification by IH
So, $\sigma[n]$ is resolved eventually in the future to a constant c by applying all its parameters.
$\sigma[n]\ \in\ R_V[t^{IV}],\ t^{IV} = max(t", t''')$

On the other hand the decentralized with simplification version will resolve it by evaluating the expression of $\sigma[n]$ applying some its parameters but will yield the same result by Lemma 5.
Thus, $\sigma[n]\ \in\ R_V$. Considering that the needed inputs for the expression are equal

as those of the decentralized version with evaluation, and all the needed subexpressions with lesser ρ are resolved to the same values in both algorithms by IH. Then $simpl(tv) = d$ follows. Then by lemma 5 we could substitute the remaining parameters v_{rest} in the expression on $\sigma[n]$ at t''' and evaluate the expression, obtaining the same output c, thus we can conclude that $c = d$. □

Lemma 6 (Oracle)**.** *Let there be an Oracle which injects the value of a stream in R_V for a given node V. The value injected by the oracle is the same as the value that the algorithm would obtain applying the expression of the stream to all its parameters. Just as in the centralized Lola.*

Proof. By contradiction.
Suppose there is a stream for which the value guessed by the Oracle differs from that obtained by the system.
Given that all its substreams are well calculated and that the Oracle guesses the correct value of the stream.
Then, the only possibility is that the application of the stream expression with all its parameters performed by the algorithm was incorrect. This is a contradiction. □

Chapter 4

Empirical Evaluation

In this section we describe an empirical evaluation of the techniques described in Section 3. We have implemented a prototype tool to verify the adequacy of these techniques.

We compared our results with those of the tool Themis [El-Hokayem and Falcone, 2017b] in those categories where we could find a matching counterpart. The Themis comparison experiments were performed in a machine with 16 processors Intel Xeon E5640 2.67GHz with 24 GB RAM while the rest of the experiments were performed in a desktop machine 4 processors Intel Core i5-2520 2.5GHz 4 GB RAM.

Themis comparison Themis is a tool that verifies LTL specifications by building the automaton and verifiying over it using the Execution History Encoding (EHE). So, as a first step we have built a compiler from LTL formulas to Lola formulas. Moreover, we can translate those formulas to any of the available logics that we have implemented (see Section 3.3). We compared our centralized setting (with decentralized observation) with the Themis' Orchestration algorithm and also compared our decentralized setting with Themis' Choreography algorithm.

We have synthetized 213196 tests with 5 nodes, with the same inputs and formulas to those of Themis, which were randomly generated, in order to compare with them. We have extracted all the Themis tool results from the database provided openly on Github.

Of the performed tests, 85% of them returned the same verdict as Themis provided while the remaining 15% did not have a value in the Themis database. This may be due to a timeout that was used in Themis, however with our tool we were able to reach a verdict. These experiments that did not have a counterpart in the database were also considered when computing the metrics of our system, thus affecting the values in the presented tables Table 4.1, Table 4.2, Table 4.3.

		min		avg		max	
		dLola	Themis	dLola	Themis	dLola	Themis
Lazy decentralized	normalized	1.50	0.00	55.61	71.61	944.25	600.00
	raw	6.00	0.00	564.19	6751.12	4201.00	66000.00
Lazy centralized	normalized	0.11	0.00	7.11	73.20	240.75	440.00
	raw	1.00	0.00	98.33	7085.40	1001.00	48400.00
Eval decentralized	normalized	3.00	0.00	30.61	71.58	500.25	600.00
	raw	12.00	0.00	332.50	6750.00	2101.00	66000.00
Eval centralized	normalized	2.25	0.00	10.81	72.34	200.25	440.00
	raw	9.00	0.00	140.88	6994.35	801.00	48400.00

Table 4.1: Themis comparison:#msgs

Results discussion Table 4.1 shows the number of messages used to compute the verdict of the specifications. We can observe how in the best case (those that minimize the #msgs), the Lazy strategy(3.4) consumes less messages than the Eval strategy(3.4), as expected. Also we can see how for the worst case the Eval consumes less messages than the Lazy, also as expected from the design of the two strategies. Please note that the specification that minimizes the Lazy strategy need not be the same as the one that minimizes the Eval strategy. Comparing with Themis we can observe that our tool uses less messages in every case except for the minimums.

Table 4.2 shows the payload (in bits) used for the computation of verdicts. We have measured the payload accounting for the size needed to enconde the data (1 bit for a boolean, 8 bits for a character, 32 bits for an integer). Comparing with Themis we can observe that our tool uses a lower payload in every case except for the minimums.

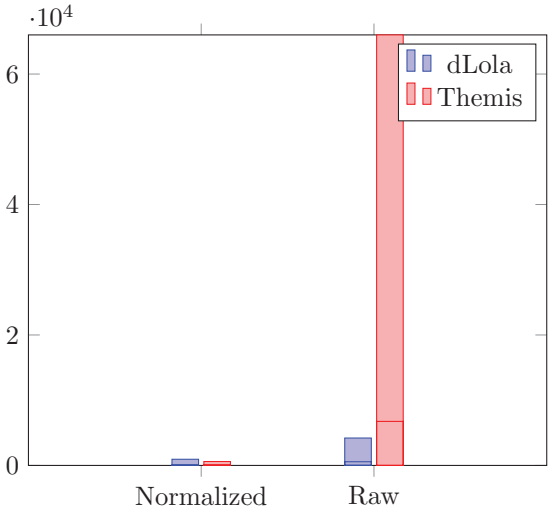
Table 4.3 shows the maximum difference per experiment between the time that all the inputs needed to compute a subformula were in the system and the time the subformula was resolved. Comparing with Themis we can observe that our tool incurrs in a higher maximum delay for the centralized cases, while for the decentralized cases the metric is significantly lower. In the figures below we can observe the plotting of the data of the tables. In each of them the minimum, average and maximum are represented, for all the categories studied. Please note that in most of them the minimum and sometimes the average cannot be seen.

		min		avg		max	
		dLola	Themis	dLola	Themis	dLola	Themis
Lazy decentralized	normalized	34.875	0	1297.99	642	22081.625	5400
	raw	139.5	0	13186.87	60743.17	97862	594000
Lazy centralized	normalized	2.51	0	160.09	876.35	5402.34	5245
	raw	24.5	0	2208.38	83833.05	22462	576950
Eval decentralized	normalized	69.75	0	715.02	642.11	11706.13	5400
	raw	279	0	7792.24	60735.12	49074.5	594000
Eval centralized	normalized	51.125	0	243.56	866.22	4506.13	5245
	raw	204.5	0	3171.91	82759.45	18024.5	576950

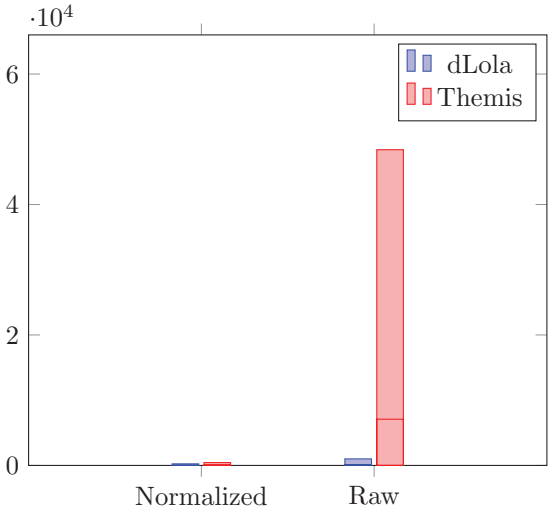
Table 4.2: Themis comparison: payload

		min		avg		max	
		dLola	Themis	dLola	Themis	dLola	Themis
Lazy decentralized	normalized	0.5	0	2.67	2.37	27.75	37
	raw	2	0	23.72	84.46	115	4070
Lazy centralized	normalized	0	0	1.32	0.81	25.25	1
	raw	0	0	17.61	6.52	101	110
Eval decentralized	normalized	0.25	0	1.93	2.38	26.5	46
	raw	1	0	20.49	85.064	110	5060
Eval centralized	normalized	0	0	1.11	0.81	25	1
	raw	0	0	16.59	6.41	100	110

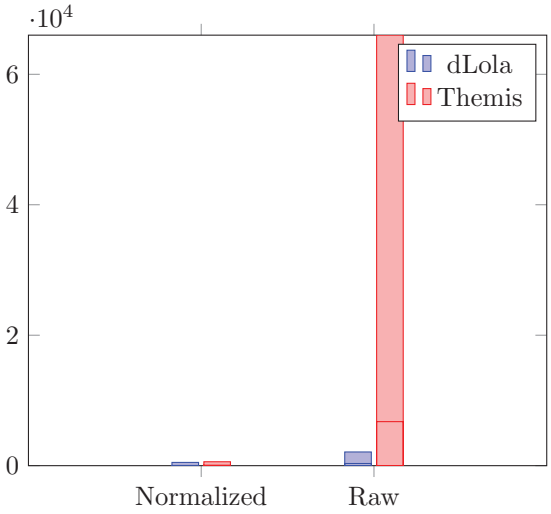
Table 4.3: Themis comparison: max delay



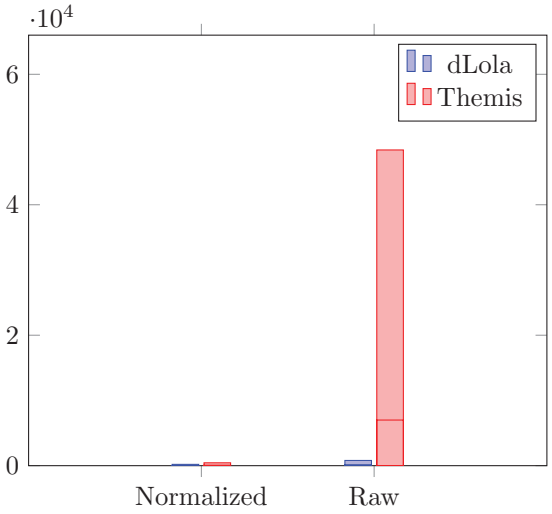
 # msgs Lazy Decentralized



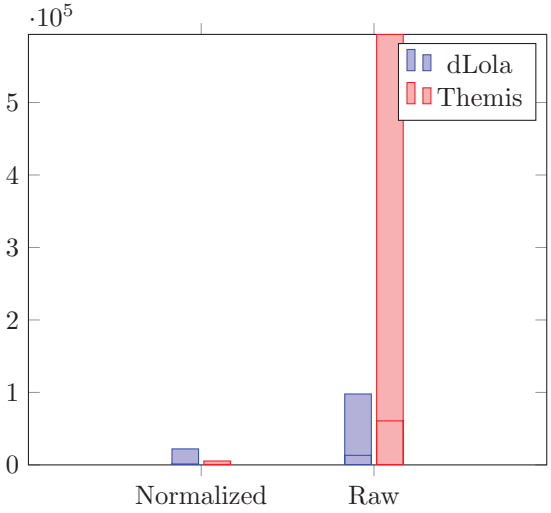
 # msgs Lazy Centralized



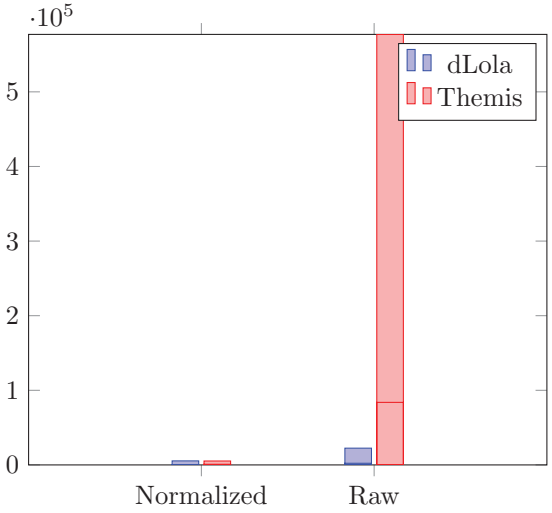
 # msgs Eval Decentralized



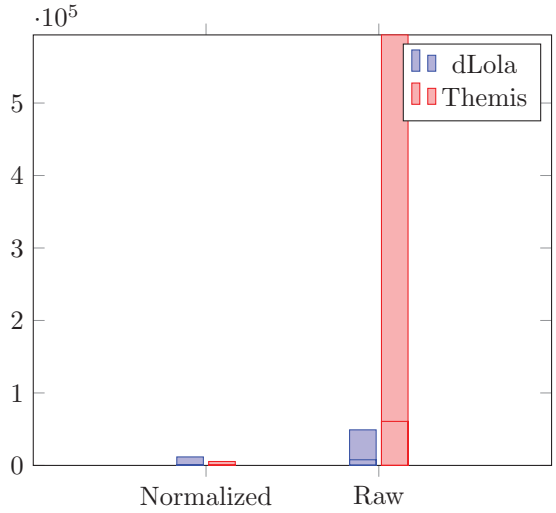
 # msgs Eval Centralized



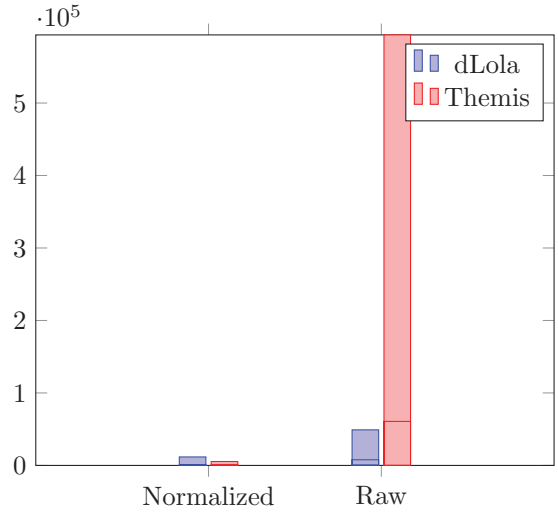
Payload Lazy Decentralized



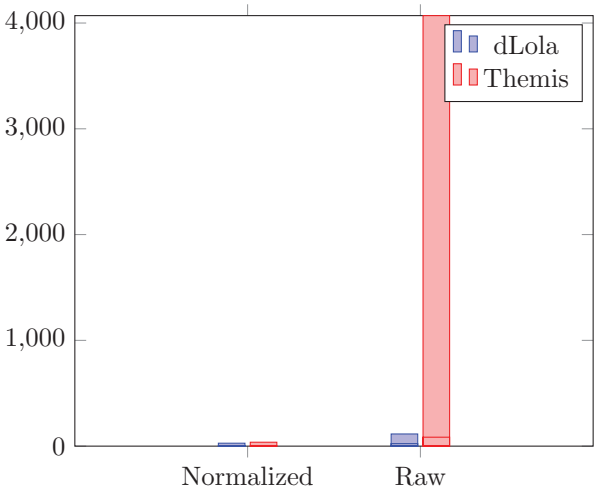
Payload Lazy Centralized



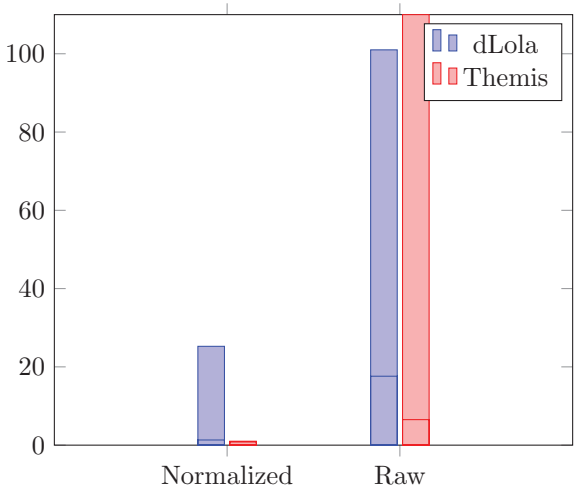
Payload Lazy Decentralized



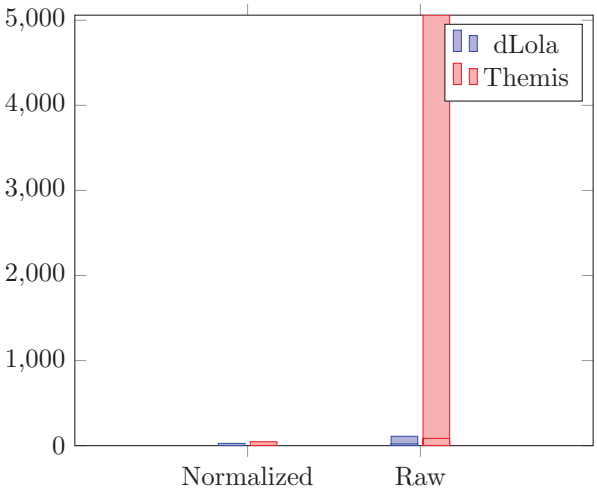
Payload Lazy Centralized



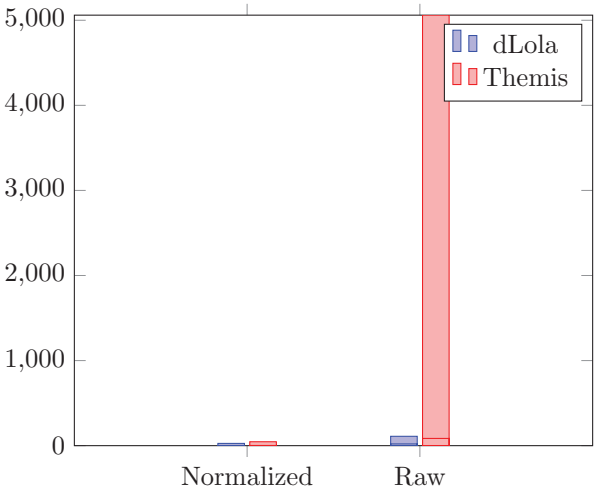
Max delay Lazy Decentralized



Max delay Lazy Centralized



Max delay Lazy Decentralized



Max delay Lazy Centralized

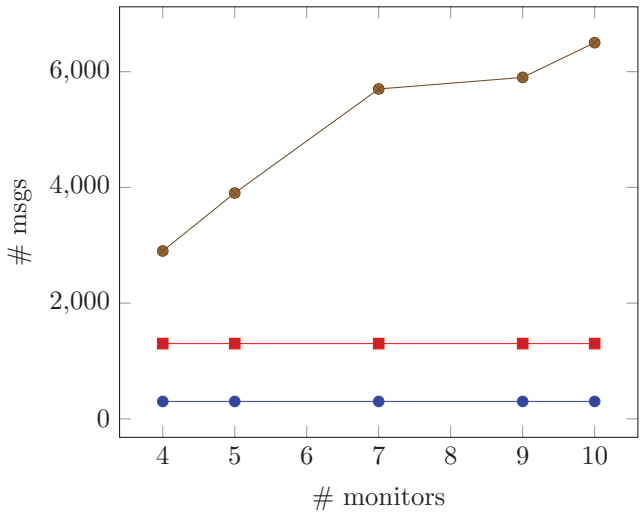
Topologies and arithmetics We had the intuition that the most important aspect for the performance of the monitors was the placement of subformulas of the specification among them. So, we designed several experiments to verify our hypothesis in which monitors will compute the same specification over the same random input of length 100. We also fixed the placement of the observations which are the inputs of the monitors to be the following: a in $Monitor0$, b in $Monitor1$, c in $Monitor2$ and d in $Monitor3$ (there are at least 4 monitors in all our experiments). Then for each of the selected topologies we crafted 3 different placements: one made by hand in which message passing is minimized, other where the subformulas are distributed evenly in a round-robin fashion among the nodes, and the third one searching to maximize the number of messages interchanged. The selected topologies are the following: ring, ringshort, line, clique and star. A ring topology is a circular topology in which all messages move clockwise (or counterclockwise). A ringshort topology is a circular topology where the communication among nodes is made through the shortest path among them. A line topology is a linear topology. A clique topology is a topology where every node is connected to the rest of nodes. A star topology is a topology with a central node and a number of arms which have the same number of nodes in them. So, in order for nodes of different arms to communicate, the messages need to pass through the center of the star. The specification is the following:

```
input int a, b, c, d
define int a' = a, b' = b, c' = c, d' = d
define int ab   = a' + b'
define int abc  = ab + c'
define int abcd = abc + d'
output int acc  = abcd + acc[+1|0]
```

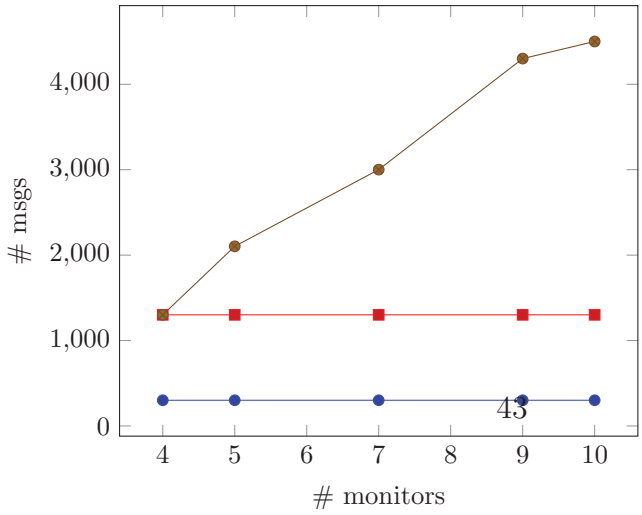
Table 4.4 shows how the placement of subformulas affect the overall efficiency of the monitors. This table confirms our hypothesis: the placement of subformulas is even more relevant to the efficiency than the existing topology, and proves that a formula can be resolved with a constant number of messages independently of the shape and the size of the network topology. This is important since it is possible that this topology may not be modified in every case, while the placement of subformulas is always under our control.

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

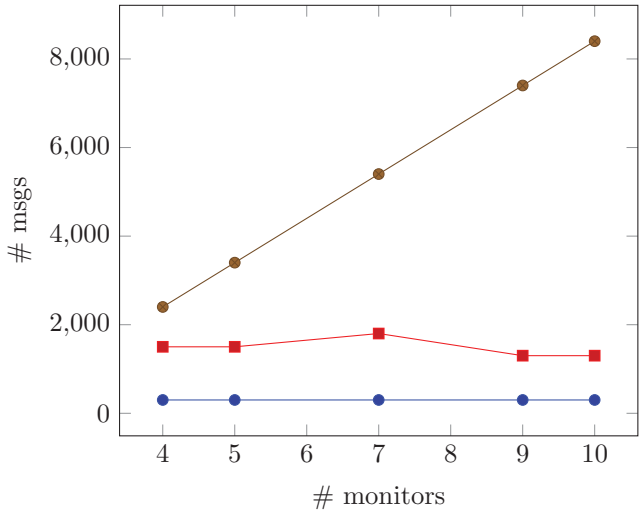
Table 4.4: Topology experiments: #msgs needed by topology and number of nodes



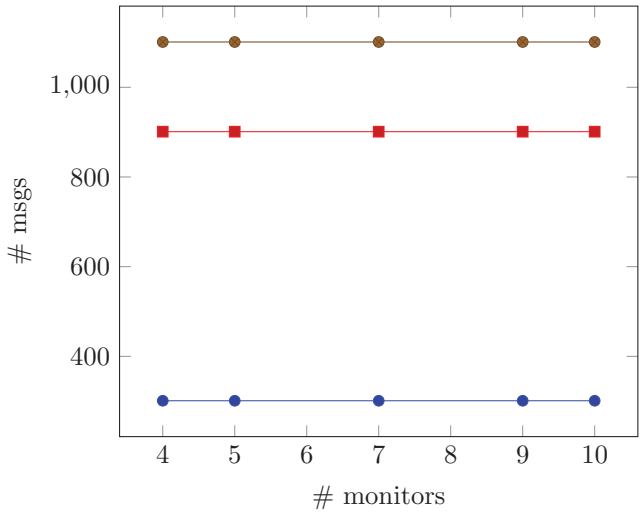
 # msgs Ring topology



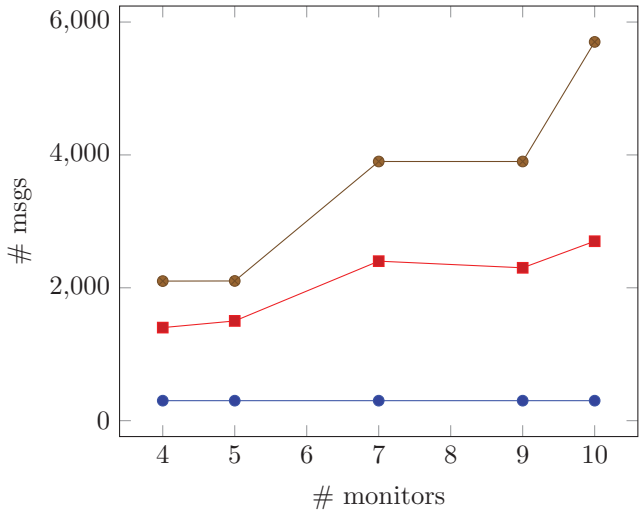
 # msgs Ringshort topology



 # msgs Line topology



 # msgs Clique topology



 # msgs Star topology

Chapter 5

Conclusions and Future work

Conclusions In this thesis we have studied the problem of how to monitor SRV specifications in a decentralized manner. We have considered placement of observations, placement of subformulas and different topologies along with different supported datatypes for the specifications. We have proved the correctness of the proposed algorithm and shown empirically that the approach taken in this paper is effective and efficient by comparing to the tool Themis. We have shown that the placement of subformulas into nodes is the most important factor affecting performance. Also we have shown empirically that a formula can be resolved with a constant number of messages independently of the shape and the size of the network topology.

Future work We plan to extend the system presented in this paper to the Gobally asynchoronous locally synchronous model of computation. Other interesting paths to explore would be fault tolerance, robustness or enforcement.

Bibliography

[Asarin et al., 2002] Asarin, E., Caspi, P., and Maler, O. (2002). Timed regular expressions. *J. ACM*, 49(2):172–206.

[Barringer et al., 2004] Barringer, H., Goldberg, A., Havelund, K., and Sen, K. (2004). Rule-based runtime verification. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57, Venice, Italy. Springer-Verlag.

[Basin et al., 2015] Basin, D., Klaedtke, F., and Zalinescu, E. (2015). Failure-aware Runtime Verification of Distributed Systems. In Harsha, P. and Ramalingam, G., editors, *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)*, volume 45 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 590–603, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Bauer and Falcone, 2011] Bauer, A. and Falcone, Y. (2011). Decentralised LTL monitoring. *CoRR*, abs/1111.5133.

[Bauer et al., 2011] Bauer, A., Leucker, M., and Schallhart, C. (2011). Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14.

[Convent et al.,] Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., and Thoma, D. TeSSLa: Temporal stream-based specification language. submitted.

[D'Angelo et al., 2005] D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H. B., Mehrotra, S., and Manna, Z. (2005). LOLA: Runtime monitoring of synchronous systems. In *Proceedings of the 12th International Symposium of Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE Computer Society Press.

[Dong et al., 2008] Dong, W., Leucker, M., and Schallhart, C. (2008). Impartial anticipation in runtime-verification. In *6th International Symposium on Automated Technology for Verification and Analysis (ATVA'08)*, volume 5311 of *LNCS*, pages 386–396. Springer-Verlag.

[Eisner et al., 2003] Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., and Campenhout, D. V. (2003). Reasoning with temporal logic on truncated paths. In *CAV03*, volume 2725 of *LNCS*, pages 27–39, Boulder, CO, USA. Springer.

[El-Hokayem and Falcone, 2017a] El-Hokayem, A. and Falcone, Y. (2017a). Monitoring decentralized specifications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 125–135, New York, NY, USA. ACM.

[El-Hokayem and Falcone, 2017b] El-Hokayem, A. and Falcone, Y. (2017b). THEMIS: A Tool for Decentralized Monitoring Algorithms. In *ISSTA 2017*, Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 125–135, Santa Barbara, United States.

[Emerson and Clarke, 1980] Emerson, E. A. and Clarke, E. M. (1980). Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming (ICALP'80)*, volume 85 of *LNCS*, pages 169–181. Springer-Verlag.

[Faymonville et al., 2016] Faymonville, P., Finkbeiner, B., Schirmer, S., and Torfah, H. (2016). 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.

[Faymonville et al., 2017] Faymonville, P., Finkbeiner, B., Schwenger, M., and Torfah, H. (2017). Real-time stream-based monitoring. *CoRR*, abs/1711.03829.

[Francalanza et al., 2018] Francalanza, A., Pérez, J. A., and Sánchez, C. (2018). Runtime verification for decentralised and distributed systems. In Bartocci, E. and Falcone, Y., editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 176–210. Springer.

[Gorostiaga and Sánchez,] Gorostiaga, F. and Sánchez, C. Striver: Stream runtime verification for real-time event-streams. submitted.

[Havelund and Roşu, 2002] Havelund, K. and Roşu, G. (2002). Synthesizing monitors for safety properties. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 342–356. Springer-Verlag.

[Lee et al., 1999] Lee, I., Kannan, S., Kim, M., Sokolsky, O., and Viswanathan, M. (1999). Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, volume 1, pages 279–287. CSREA Press.

[Pike et al., 2010] Pike, L., Goodloe, A., Morisset, R., and Niller, S. (2010). Copilot: A hard real-time runtime monitor. In Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., and Tillmann, N., editors, *Runtime Verification*, pages 345–359, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Pike et al., 2013] Pike, L., Wegmann, N., Niller, S., and Goodloe, A. (2013). Copilot: monitoring embedded systems. *Innovations in Systems and Software Engineering*, 9(4):235–255.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS'77),*, pages 46–67. IEEE Computer Society Press.

[Queille and Sifakis, 1982] Queille, J.-P. and Sifakis, J. (1982). Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer.

[Roşu and Havelund, 2005] Roşu, G. and Havelund, K. (2005). Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197.

[Sen and Roşu, 2003] Sen, K. and Roşu, G. (2003). Generating optimal monitors for extended regular expressions. In Sokolsky, O. and Viswanathan, M., editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier.

[Sen et al., 2004a] Sen, K., Roşu, G., and Agha, G. (2004a). Online efficient predictive safety analysis of multithreaded programs. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 123–138.

[Sen et al., 2004b] Sen, K., Vardhan, A., Agha, G., and Rosu, G. (2004b). Efficient decentralized monitoring of safety in distributed systems. In *Proceedings. 26th International Conference on Software Engineering*, pages 418–427.

Appendix A

Additional Information

A.1 More Examples

Example 9. *Consider the property "every time that the light switch is activated, then the lights are turned on in the next tick". The light-switch and the light status are two input streams of type Bool. Then we need to express that "if the light switch is on in the next tick the light will be on". Finally we want this behaviour to hold always. This property can be expressed in LOLA in the following way:*

$$activate[n] = light_{switch}[n] \Longrightarrow light_{on}[n+1]$$
$$always[n] = activate[n] \wedge always[n+1]$$

This is the end of the example. □

Example 10. *Let's explain how the expression resolution procedure works at a high level. Lets consider the following expression $e[n] = i[n-1|-2]+1$ to be the only expression in an isolated monitor that receives as an input the stream i. The input stream i is received at each corresponding cycle, so $i[0]$ will be received at $n = 0$ and stored in Resolved and so on for all n. This expression computes the successor of the input received in the previous cycle.*

Starting at $n = 0$, the expression will be instantiated with the current value of n, replacing n with its actual value in the expression, in our case $e[0] = i[0-1|-2]+1 = i[-1|-2]+1$. As $i[-1]$ is an invalid position in the stream i, then the default value for i will be used, that is, -2. So, the expression will simplify to $e[0] = -2+1 = -1$. As it is now resolved, it will be removed from Unresolved and inserted in Resolved.

Then, at $n = 1$ the equation is instantiated again $e[1] = i[1-1|-1]+1 = i[0]+1$. This time, the required streams for the evaluation of $e[1]$ are just $i[0]$ (which is a correct position

of the stream i because it is inside the trace limits $0 <= t < N$) and will be searched in Resolved. If the value of the pair $(stream, time)$ is found in Resolved, then it is used to replace all references in every expression in Unresolved. In this case the value of $i[0]$ will be used to substitute in the appropiate place in the expression resulting in $e[1] = 7 + 1 = 8$. As it is resolved it will be moved to Resolved. If the values of the needed timed streams were not found in Resolved, then the expression could not be simplified any further and would be left in Unresolved. Hopefully, the expression will be simplified it in the next tick.

Example 11. *At time t Monitor1 has the following equation in U, $\varphi_t = \sigma_t$, and it knows that σ_t will be computed by Monitor2. As Monitor1 needs the value of σ_t, it will send the following message $req\{\sigma_t, Mon1, Mon2\}$ which can be read as "Monitor1 requests the value of σ at time t to Monitor2". Then at time $t + 1$ Monitor2 receives the message and responds with the message $res\{\sigma_t = c, Mon2, Mon1\}$ which analogously can be read as "Monitor2 responds $\sigma_t = c$ to Monitor1". At last, at time $t + 2$ Monitor1 receives the message and can use the value of $\sigma_t = c$.*