Decentralised LTL monitoring

Andreas Bauer[1] and Yliès Falcone[2⋆]

[1] NICTA Software Systems Research Group ⋆⋆ and Australian National University
[2] Laboratoire d'Informatique de Grenoble, UJF Université Grenoble I, France

Abstract. Users wanting to monitor distributed or component-based systems often perceive them as monolithic systems which, seen from the outside, exhibit a uniform behaviour as opposed to many components displaying many local behaviours that together constitute the system's global behaviour. This level of abstraction is often reasonable, hiding implementation details from users who may want to specify the system's global behaviour in terms of an LTL formula. However, the problem that arises then is how such a specification can actually be monitored in a distributed system that has no central data collection point, where all the components' local behaviours are observable. In this case, the LTL specification needs to be decomposed into sub-formulae which, in turn, need to be distributed amongst the components' locally attached monitors, each of which sees only a distinct part of the global behaviour.
The main contribution of this paper is an algorithm for distributing and monitoring LTL formulae, such that satisfaction or violation of specifications can be detected by local monitors alone. We present an implementation and show that our algorithm introduces only a minimum delay in detecting satisfaction/violation of a specification. Moreover, our practical results show that the communication overhead introduced by the local monitors is generally lower than the number of messages that would need to be sent to a central data collection point.

1 Introduction

Much work has been done on monitoring systems w.r.t. formal specifications such as linear-time temporal logic (LTL [1]) formulae. For this purpose, a system is thought of more or less as a "black box", and some (automatically generated) monitor observes its outside visible behaviour in order to determine whether or not the runtime behaviour satisfies an LTL formula. Applications include monitoring programs written in Java or C (cf. [2, 3]) or abstract Web services (cf. [4]) to name just a few.

From a system designer's point of view, who defines the overall behaviour that a system has to adhere to, this "black box" view is perfectly reasonable. For example, most modern cars have the ability to issue a warning if a passenger (including the driver) is not wearing a seat belt after the vehicle has reached a certain speed. One could imagine using a monitor to help issue this warning based on the following LTL formalisation, which captures this abstract requirement:

$$\varphi = \mathbf{G}\big(speed\_low \vee ((pressure\_sensor\_1\_high \Rightarrow seat\_belt\_1\_on) \wedge \ldots \wedge (pressure\_sensor\_n\_high \Rightarrow seat\_belt\_n\_on))\big)$$

The formula φ asserts that, at all times, when the car has reached a certain speed, and the pressure sensor in a seat $i \in [1, n]$ detects that a person is sitting in it ($pressure\_sensor\_i\_high$), it has to be the case that the corresponding seat belt is fastened ($seat\_belt\_i\_on$). Moreover, one can build a monitor for φ, which receives the respective sensor values

⋆ This author was supported by an Inria Grant to visit NICTA Canberra where part of this work has been carried out.
⋆⋆ NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

and is able to assert whether or not these values constitute a violation—*but*, only if some central component exists in the car's network of components, which collects these sensor values and consecutively sends them to the monitor as input! In many real-world scenarios, such as the automotive one, this is an unrealistic assumption mainly for economic reasons, but also because the communication on a car's bus network has to be kept minimal. Therefore one cannot continuously send unnecessary sensor information on a bus that is shared by critical applications where low latency is paramount (cf. [5, 6]). In other words, in these scenarios, one has to monitor such a requirement not based on a single behavioural trace, assumed to be collected by some global sensor, but based on the many *partial* behavioural traces of the components which make up the actual system. We refer to this as *decentralised LTL monitoring* when the requirement is given in terms of an LTL formula.

The main constraint that decentralised LTL monitoring addresses is the lack of a global sensor and a central decision making point asserting whether the system's behaviour has violated or satisfied a specification. We already pointed out that, from a practical point of view, a central decision making point (i.e., global sensor) would require all the individual components to continuously send events over the network, and thereby negatively affecting response time for other potentially critical applications on the network. Moreover from a theoretical point of view, a central observer (resp. global sensor) basically resembles classical LTL monitoring, where the decentralised nature of the system under scrutiny does not play a role. Arguably, there exist many real-world component-based applications, where the monitoring of an LTL formula can be realised via global sensors or central decision making points, e.g., when network latency and criticality do not play an important role. However, here we want to focus on those cases where there exists no global trace, no central decision making point, and where the goal is to keep the communication, required for monitoring the LTL formula, minimal.

In the decentralised setting, we assume that the system under scrutiny consists of a set of components $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$, communicating on a synchronous bus acting as global clock. Each component emits events synchronously and has a local monitor attached to it. The set of all events is $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_n$, where Σ_i is the set of events visible to the monitor at component C_i. The global LTL formula, on the other hand, is specified over a set of propositions, AP, such that $\Sigma = 2^{AP}$. Moreover, we demand for all $i, j \leq n$ with $i \neq j$ that $\Sigma_i \cap \Sigma_j = \emptyset$ holds, i.e., events are local w.r.t. the components where they are monitored.

At first, the synchronous bus may seem an overly stringent constraint imposed by our setting. However, it is by no means unrealistic, since in many real-world systems, especially critical ones, communication occurs synchronously. For example, the FlexRay bus protocol, used for safety-critical systems in the automotive domain, allows synchronous communication (cf. [7, 5, 8]). What is more, experts predict "that the data volume on FlexRay buses will increase significantly in the future" [6, Sec. 2], promoting techniques to minimise the number of used communication slots. Hence, one could argue that synchronous distributed systems such as FlexRay, in fact, motivate the proposed decentralised monitoring approach. (Although, one should stress that the results in this paper do not directly target FlexRay or any other specific bus system.)

Let as before φ be an LTL formula formalising a requirement over the system's global behaviour. Then every local monitor, M_i, will at any time, t, monitor its own

LTL formula, φ_i^t, w.r.t. a partial behavioural trace, u_i. Let us use $u_i(m)$ to denote the $(m+1)$-th event in a trace u_i, and $\mathbf{u} = (u_1, u_2, \ldots, u_n)$ for the *global trace*, obtained by pair-wise parallel composition of the partial traces, each of which at time t is of length $t+1$ (i.e., $\mathbf{u} = u_1(0) \cup \ldots \cup u_n(0) \cdot u_1(1) \cup \ldots \cup u_n(1) \cdots u_1(t) \cup \ldots \cup u_n(t)$, a sequence of union sets). Note that from this point forward we will use $\mathbf{u}$ only when, in a given context, it is important to consider a global trace. However, when the particular type of trace (i.e., partial or global) is irrelevant, we will simply use u, u_i, etc. We also shall refer to partial traces as local traces due to their locality to a particular monitor in the system.

The decentralised monitoring algorithm evaluates the global trace $\mathbf{u}$ by considering the locally observed traces u_i, $i \in [1, n]$, in separation. In particular, it exhibits the following properties.

- If a local monitor yields $\varphi_i^t = \bot$ (resp. $\varphi_i^t = \top$) on some component C_i by observing u_i, it implies that $\mathbf{u}\Sigma^\omega \subseteq \Sigma^\omega \setminus \mathcal{L}(\varphi)$ (resp. $\mathbf{u}\Sigma^\omega \subseteq \mathcal{L}(\varphi)$) holds where $\mathcal{L}(\varphi)$ is the set of infinite sequences in Σ^ω described by φ. That is, a locally observed violation (resp. satisfaction) is, in fact, a global violation (resp. satisfaction). Or, in other words, $\mathbf{u}$ is a bad (resp. good) prefix for φ.
- If the monitored trace $\mathbf{u}$ is such that $\mathbf{u}\Sigma^\omega \subseteq \Sigma^\omega \setminus \mathcal{L}(\varphi)$ (resp. $\mathbf{u}\Sigma^\omega \subseteq \mathcal{L}(\varphi)$), one of the local monitors on some component C_i yields $\varphi_i^{t'} = \bot$ (resp. $\varphi_i^{t'} = \top$), $t' \geq t$, for an observation u_i', an extension of u_i, the local observation of $\mathbf{u}$ on C_i, because of some latency induced by decentralised monitoring, as we shall see.

However, in order to allow for the local detection of global violations (and satisfactions), monitors must be able to communicate, since their traces are only partial w.r.t. the global behaviour of the system. Therefore, our second objective is to monitor with *minimal communication overhead* (in comparison with a centralised solution where at any time, t, all n monitors send the observed events to a central decision making point).

Outline. Preliminaries are in Sec. 2. LTL monitoring via formula rewriting (progression), a central concept to our paper, is discussed in Sec. 3. In Sec. 4, we lift it to the decentralised setting. The semantics induced by decentralised LTL monitoring is outlined in Sec. 5, whereas Sec. 6 details on how the local monitors operate in this setting and gives a concrete algorithm. Experimental results are presented in Sec. 7. Section 8 concludes and gives pointers to related work. Formal proofs are available in an extended version of this paper, available as technical report [9].

2 Preliminaries

Each component of the system emits events at discrete time instances. An event σ is a set of *actions* denoted by some atomic propositions from the set AP, i.e., $\sigma \in 2^{AP}$. We denote 2^{AP} by Σ and call it the *alphabet* (of system events).

As our system operates under the *perfect synchrony hypothesis* (cf. [10]), we assume that its components communicate with each other in terms of sending and receiving messages (which, for the purpose of easier presentation, can also be encoded by actions) at *discrete* instances of time, which are represented using identifier $t \in \mathbb{N}^{\geq 0}$. Under this hypothesis, it is assumed that neither computation nor communication take time. In other words, at each time t, a component may receive up to $n-1$ messages and dispatch up to 1 message, which in the latter case will always be available at the respec-

tive recipient of the messages at time $t+1$. Note that these assumptions extend to the components' monitors, which operate and communicate on the same synchronous bus. The hypothesis of perfect synchrony essentially abstracts away implementation details of how long it takes for components or monitors to generate, send, or receive messages. As indicated in the introduction, this is a common hypothesis for certain types of systems, which can be designed and configured (e.g., by choosing an appropriate duration between time t and $t+1$) to not violate this hypothesis (cf. [10]).

We use a projection function Π_i to restrict atomic propositions or events to the local view of monitor M_i, which can only observe those of component C_i. For atomic propositions, $\Pi_i : 2^{AP} \to 2^{AP}$ and we denote $AP_i = \Pi_i(AP)$ for $i \in [1,n]$. For events, $\Pi_i : 2^{\Sigma} \to 2^{\Sigma}$ and we denote $\Sigma_i = \Pi_i(\Sigma)$ for $i \in [1,n]$. We also assume $\forall i,j \leq n.\ i \neq j \Rightarrow AP_i \cap AP_j = \emptyset$ and consequently $\forall i,j \leq n.\ i \neq j \Rightarrow \Sigma_i \cap \Sigma_j = \emptyset$.Seen over time, each component C_i produces a *trace* of events, also called its *behaviour*, which for t time steps is encoded as $u_i = u_i(0) \cdot u_i(1) \cdots u_i(t-1)$ with $\forall t' < t.\ u_i(t') \in \Sigma_i$. Finite traces over an alphabet Σ are elements of the set Σ^* and are typically encoded by $u, u', \ldots$, whereas infinite traces over Σ are elements of the set Σ^ω and are typically encoded by $w, w', \ldots$ The set of all traces is given by the set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. The set $\Sigma^* \setminus \{\epsilon\}$ is noted Σ^+. The finite or infinite sequence w^t is the *suffix* of the trace $w \in \Sigma^\infty$, starting at time t, i.e., $w^t = w(t) \cdot w(t+1) \cdots$. The system's global behaviour, $\mathbf{u} = (u_1, u_2, \ldots, u_n)$ can now be described as a sequence of pair-wise union of the local events in component's traces, each of which at time t is of length $t+1$ i.e., $\mathbf{u} = u(0) \cdots u(t)$.

Moreover since we use LTL to specify system behaviour, we also assume that the reader is familiar with the standard definition of LTL (cf. [1, 9]) and the usual syntactic "sugar". We refer to the syntactically correct set of LTL formulae over a finite set of atomic propositions, AP, by $\mathrm{LTL}(AP)$. When AP does not matter or is clear from the context, we also refer to this set simply by LTL. Finally, for some $\varphi \in \mathrm{LTL}(AP)$, $\mathcal{L}(\varphi) \subseteq \Sigma^\omega$ denotes the individual models of φ (i.e., set of traces). A set $L \subseteq \Sigma^\omega$ is also called a *language* (over Σ).

3 Monitoring LTL formulae by progression

Central to our monitoring algorithm is the notion of *good and bad prefixes* for an LTL formula or, to be more precise, for the language it describes:

Definition 1. *Let $L \subseteq \Sigma^\omega$ be a language. The set of all* good prefixes *(resp.* bad prefixes*) of L is given by $\mathrm{good}(L)$ (resp. $\mathrm{bad}(L)$) and defined as follows:*

$$\mathrm{good}(L) = \{u \in \Sigma^* \mid u \cdot \Sigma^\omega \subseteq L\}, \qquad \mathrm{bad}(L) = \{u \in \Sigma^* \mid u \cdot \Sigma^\omega \subseteq \Sigma^\omega \setminus L\}.$$

We will shorten $\mathrm{good}(\mathcal{L}(\varphi))$ (resp. $\mathrm{bad}(\mathcal{L}(\varphi))$) to $\mathrm{good}(\varphi)$ (resp. $\mathrm{bad}(\varphi)$).

Although there exist a myriad of different approaches to monitoring LTL formulae, based on various finite-trace semantics (cf. [11]), one valid way of looking at the monitoring problem for some formula $\varphi \in \mathrm{LTL}$ is the following: The monitoring problem of $\varphi \in \mathrm{LTL}$ is to devise an efficient monitoring algorithm which, in a stepwise manner, receives events from a system under scrutiny and states whether or not the trace observed so far constitutes a good or a bad prefix of $\mathcal{L}(\varphi)$. One monitoring approach along those lines is described in [12]. We review an alternative monitoring procedure based on for-

mula rewriting, which is also known as formula progression, or just *progression* in the domain of planning with temporally extended goals (cf. [13]).

Progression splits a formula into a formula expressing what needs to be satisfied by the current observation and a new formula (referred to as a *future goal* or *obligation*), which has to be satisfied by the trace in the future. As progression plays a crucial role in decentralised LTL monitoring, we recall its definition for the full set of LTL operators.

Definition 2. *Let $\varphi, \varphi_1, \varphi_2 \in \mathrm{LTL}$, and $\sigma \in \Sigma$ be an event. Then, the* progression function $P : LTL \times \Sigma \to LTL$ *is inductively defined as follows:*

$$
\begin{array}{ll}
P(p \in AP, \sigma) & = \top, \text{ if } p \in \sigma, \bot \text{ otherwise} \\
P(\varphi_1 \vee \varphi_2, \sigma) & = P(\varphi_1, \sigma) \vee P(\varphi_2, \sigma) \\
P(\varphi_1 \mathbf{U} \varphi_2, \sigma) & = P(\varphi_2, \sigma) \vee P(\varphi_1, \sigma) \wedge \varphi_1 \mathbf{U} \varphi_2 \\
P(\mathbf{G}\varphi, \sigma) & = P(\varphi, \sigma) \wedge \mathbf{G}(\varphi) \\
P(\mathbf{F}\varphi, \sigma) & = P(\varphi, \sigma) \vee \mathbf{F}(\varphi)
\end{array}
\qquad
\begin{array}{ll}
P(\top, \sigma) & = \top \\
P(\bot, \sigma) & = \bot \\
P(\neg\varphi, \sigma) & = \neg P(\varphi, \sigma) \\
P(\mathbf{X}\varphi, \sigma) & = \varphi
\end{array}
$$

Note that monitoring using rewriting with similar rules as above has been described, for example, in [14, 15], although not necessarily with the same finite-trace semantics in mind that we are discussing in this paper. Informally, the progression function "mimics" the LTL semantics on an event σ, as it is stated by the following lemmas.

Lemma 1. *Let φ be an* LTL *formula, σ an event and w an infinite trace, we have $\sigma \cdot w \models \varphi \Leftrightarrow w \models P(\varphi, \sigma)$.*

Lemma 2. *If $P(\varphi, \sigma) = \top$, then $\sigma \in \mathrm{good}(\varphi)$, if $P(\varphi, \sigma) = \bot$, then $\sigma \in \mathrm{bad}(\varphi)$.*

Moreover it follows that if $P(\varphi, \sigma) \notin \{\top, \bot\}$, then there exist traces $w, w' \in \Sigma^\omega$, such that $\sigma \cdot w \models \varphi$ and $\sigma \cdot w' \not\models \varphi$ hold. Let us now get back to [12], which introduces a finite-trace semantics for LTL monitoring called LTL_3. It is captured by the following definition.

Definition 3. *Let $u \in \Sigma^*$, the satisfaction relation of* LTL_3, $\models_3 \colon \Sigma^* \times \mathrm{LTL} \to \mathbb{B}_3$, *with $\mathbb{B}_3 = \{\top, \bot, ?\}$, is defined as*

$$
u \models_3 \varphi = \begin{cases} \top & \text{if } u \in \mathrm{good}(\varphi), \\ \bot & \text{if } u \in \mathrm{bad}(\varphi), \\ ? & \text{otherwise}. \end{cases}
$$

Based on this definition, it now becomes obvious how progression *could* serve as a monitoring algorithm for LTL_3.

Theorem 1. *Let $u = u(0) \cdots u(t) \in \Sigma^+$ be a trace, and $v \in \mathrm{LTL}$ be the verdict, obtained by $t + 1$ consecutive applications of the progression function of φ on u, i.e., $v = P(\ldots(P(\varphi, u(0)), \ldots, u(t)))$. The following cases arise: If $v = \top$, then $u \models_3 \varphi = \top$ holds. If $v = \bot$, then $u \models_3 \varphi = \bot$ holds. Otherwise, $u \models_3 \varphi = ?$ holds.*

Note that in comparison with the monitoring procedure for LTL_3, described in [12], our algorithm, implied by this theorem, has the disadvantage that the formula, which is being progressed, may grow in size relative to the number of events. However, in practice, the addition of some practical simplification rules to the progression function usually prevents this problem from occurring.

4 Decentralised progression

Conceptually, a monitor, M_i, attached to component C_i, which observes events over $\Sigma_i \subseteq \Sigma$, is a rewriting engine that accepts as input an event $\sigma \in \Sigma_i$, and an LTL formula φ, and then applies LTL progression rules. Additionally at each time t, in our

n-component architecture, a monitor can send a message and receive up to $n-1$ messages in order to communicate with the other monitors in the system, using the same synchronous bus that the system's components communicate on. The purpose of these messages is to send future or even past obligations to other monitors, encoded as LTL formulae. In a nutshell, a formula is sent by some monitor M_i, whenever the most urgent outstanding obligation imposed by M_i's current formula at time t, φ_i^t, cannot be checked using events from Σ_i alone. Intuitively, the urgency of an obligation is defined by the occurrences (or lack of) certain temporal operators in it. For example, in order to satisfy $p \wedge \mathbf{X}q$, a trace needs to start with p, followed by a q. Hence, the obligation imposed by the subformula p can be thought of as "more urgent" than the one imposed by $\mathbf{X}q$. A more formal definition is given later in this section.

When progressing an LTL formula, e.g., in the domain of planning to rewrite a temporally extended LTL goal during plan search, the rewriting engine, which implements the progression rules, will progress a state formula $p \in AP$, with an event σ such that $p \notin \sigma$, to $\bot$, i.e., $P(p, \emptyset) = \bot$ (see Definition 2). However, doing this in the decentralised setting, could lead to wrong results. In other words, we need to make a distinction as to why $p \notin \sigma$ holds locally, and then to progress accordingly. Consequently, the progression rule for atomic propositions is simply adapted by parameterising it with a local set of atomic propositions AP_i:

$$P(p, \sigma, AP_i) = \begin{cases} \top & \text{if } p \in \sigma, \\ \bot & \text{if } p \notin \sigma \wedge p \in AP_i, \\ \overline{\mathbf{X}}p & \text{otherwise,} \end{cases} \tag{1}$$

where for every $w \in \Sigma^\omega$ and $j > 0$, we have $w^j \models \overline{\mathbf{X}}\varphi$ if and only if $w^{j-1} \models \varphi$. In other words, $\overline{\mathbf{X}}$ is the dual to the $\mathbf{X}$-operator, sometimes referred to as the "previously-operator" in past-time LTL (cf. [16]). To ease presentation, the formula $\overline{\mathbf{X}}^m\varphi$ is a short for $\overbrace{\overline{\mathbf{X}}\overline{\mathbf{X}}\ldots\overline{\mathbf{X}}}^{m}\varphi$. Our operator is somewhat different to the standard use of $\overline{\mathbf{X}}$: it can only precede an atomic proposition or an atomic proposition which is preceded by further $\overline{\mathbf{X}}$-operators. Hence, the restricted use of the $\overline{\mathbf{X}}$-operator does not give us the full flexibility (or succinctness gains [17]) of past-time LTL. Using the $\overline{\mathbf{X}}$-operator, let us now formally define the *urgency* of a formula φ using a pattern matching as follows:

Definition 4. *Let φ be an LTL formula, and $\Upsilon : \mathrm{LTL} \to \mathbb{N}^{\geq 0}$ be an inductively defined function assigning a level of* urgency *to an LTL formula as follows.*

$$\begin{aligned} \Upsilon(\varphi) = \textit{match } \varphi \textit{ with } & \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 && \to \max(\Upsilon(\varphi_1), \Upsilon(\varphi_2)) \\ & \mid \overline{\mathbf{X}}\varphi' && \to 1 + \Upsilon(\varphi') \\ & \mid \_ && \to 0. \end{aligned}$$

A formula φ is said to be more urgent *than formula ψ, if and only if $\Upsilon(\varphi) > \Upsilon(\psi)$ holds. A formula φ where $\Upsilon(\varphi) = 0$ holds is said to be not urgent.*

Moreover, the above modification to the progression rules has obviously the desired effect: If $p \in \sigma$, then nothing changes, otherwise if $p \notin \sigma$, we return $\overline{\mathbf{X}}p$ in case that the monitor M_i cannot observe p at all, i.e., in case that $p \notin AP_i$ holds. This effectively means, that M_i cannot decide whether or not p occurred, and will therefore turn the state formula p into an obligation for some other monitor to evaluate rather than produce a truth-value. Of course, the downside of rewriting future goals into past goals that have

to be processed further, is that violations or satisfactions of a global goal will usually be detected *after* they have occurred. However, since there is no central observer which records all events at the same time, the monitors *need* to communicate their respective results to other monitors, which, on a synchronous bus, occupies one or more time steps, depending on how often a result needs to be passed on until it reaches a monitor which is able to actually state a verdict. We shall later give an upper bound on these communication times, and show that our decentralised monitoring framework does not introduce any additional delay under the given assumptions (see Theorem 2).

Example 1. Let us assume we have a decentralised system consisting of components A, B, C, s.t. $AP_A = \{a\}$, $AP_B = \{b\}$, and $AP_C = \{c\}$, and that a formula $\varphi = \mathbf{F}(a \wedge b \wedge c)$ needs to be monitored in a decentralised manner. Let us further assume that, initially, $\varphi_A^0 = \varphi_B^0 = \varphi_C^0 = \varphi$. Let $\sigma = \{a, b\}$ be the system event at time 0; that is, M_A observes $\Pi_A(\sigma) = \{a\}$ (resp. $\Pi_B(\sigma) = \{b\}$, $\Pi_C(\sigma) = \emptyset$ for M_B and M_C) when σ occurs. The rewriting that takes place in all three monitors to generate the next local goal formula, using the modified set of rules, and triggered by σ, is as follows:

$$\begin{aligned}
\varphi_A^1 = P(\varphi, \{a\}, \{a\}) &= P(a, \{a\}, \{a\}) \wedge P(b, \{a\}, \{a\}) \wedge P(c, \{a\}, \{a\}) \vee \varphi \\
&= \overline{\mathbf{X}} b \wedge \overline{\mathbf{X}} c \vee \varphi \\
\varphi_B^1 = P(\varphi, \{b\}, \{b\}) &= P(a, \{b\}, \{b\}) \wedge P(b, \{b\}, \{b\}) \wedge P(c, \{b\}, \{b\}) \vee \varphi \\
&= \overline{\mathbf{X}} a \wedge \overline{\mathbf{X}} c \vee \varphi \\
\varphi_C^1 = P(\varphi, \emptyset, \{c\}) &= P(a, \emptyset, \{c\}) \wedge P(b, \emptyset, \{c\}) \wedge P(c, \emptyset, \{c\}) \vee \varphi \\
&= \overline{\mathbf{X}} a \wedge \overline{\mathbf{X}} b \wedge \bot \vee \varphi = \varphi
\end{aligned}$$

But we have yet to define progression for past goals: For this purpose, each monitor has local storage to keep a *bounded* number of past events. The event that occurred at time $t - k$ is referred as $\sigma(-k)$. On a monitor observing Σ_i, the progression of a past goal $\overline{\mathbf{X}}^m \varphi$, at time $t \geq m$, is defined as follows:

$$P(\overline{\mathbf{X}}^m \varphi, \sigma, AP_i) = \begin{cases} \top & \text{if } \varphi = p \text{ for some } p \in AP_i \cap \Pi_i(\sigma(-m)), \\ \bot & \text{if } \varphi = p \text{ for some } p \in AP_i \setminus \Pi_i(\sigma(-m)), \\ \overline{\mathbf{X}}^{m+1} \varphi & \text{otherwise,} \end{cases} \tag{2}$$

where, for $i \in [1, n]$, Π_i is the projection function associated to each monitor M_i, respectively. Note that since we do not allow $\overline{\mathbf{X}}$ for the specification of a global system monitoring property, our definitions will ensure that the local monitoring goals, φ_i^t, will never be of the form $\overline{\mathbf{X}}\mathbf{X}\mathbf{X}p$, which is equivalent to a future obligation, despite the initial $\overline{\mathbf{X}}$. In fact, our rules ensure that a formula preceded by the $\overline{\mathbf{X}}$-operator is either an atomic proposition, or an atomic proposition which is preceded by one or many $\overline{\mathbf{X}}$-operators. Hence, in rule (2), we do not need to consider any other cases for φ.

5 Semantics

In the previous example, we can clearly see that monitors M_A and M_B cannot determine whether or not σ, if interpreted as a trace of length 1, is a good prefix for the global goal formula φ.[3] Monitor M_C on the other hand did not observe an action c and, therefore, is the only monitor after time 0, which knows that σ is not a good prefix and that, as before, after time 1, φ is the goal that needs to be satisfied by the system under scrutiny. Intuitively, the other two monitors know that if their respective past goals were

[3] Note that $\mathcal{L}(\varphi)$, being a *liveness* language, does not have any bad prefixes.

satisfied, then σ would be a good prefix, but in order to determine this, they need to send and receive messages to and from each other, containing LTL obligations.

Before we outline how this is done in our setting, let us discuss the semantics, obtained from this decentralised application of progression. We already said that monitors detect good and bad prefixes for a global formula; that is, if a monitor's progression yields $\top$ (resp. $\bot$), then the trace seen so far is a good (resp. bad) prefix, and if neither monitor yields a Boolean truth-value as verdict, we keep monitoring. The latter case indicates that, so far, the trace is neither a good nor a bad prefix for the global formula.

Definition 5. *Let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be the set of system components, $\varphi \in \mathrm{LTL}$ be a global goal, and $\mathcal{M} = \{M_1, \ldots, M_n\}$ be the set of component monitors. Further, let $\mathbf{u} = u_1(0) \cup \ldots \cup u_n(0) \cdot u_1(1) \cup \ldots \cup u_n(1) \cdots u_1(t) \cup \ldots \cup u_n(t)$ be the global behavioural trace, at time $t \in \mathbb{N}^{\geq 0}$. If for some component C_i, with $i \leq n$, containing a local obligation φ_i^t, M_i reports $P(\varphi_i^t, u_i(t), AP_i) = \top$ (resp. $\bot$), then $\mathbf{u} \models_D \varphi = \top$ (resp. $\bot$). Otherwise, $\mathbf{u} \models_D \varphi = ?$.*

By $\models_D$ we denote the satisfaction relation on finite traces in the decentralised setting to differentiate it from LTL_3 as well as standard LTL which is defined on infinite traces. Obviously, $\models_3$ and $\models_D$ both yield values from the same truth-domain. However, the semantics are not equivalent, since the modified progression function used in the above definition sometimes rewrites a state formula into an obligation concerning the past rather than returning a verdict. On the other hand, in the case of a one-component system (i.e., all propositions of a formula can be observed by a single monitor), the definition of $\models_D$ matches Theorem 1, in particular because our progression rule (1) is then equivalent to the standard case. Monitoring LTL_3 with progression becomes a special case of decentralised monitoring, in the following sense:

Corollary 1. *If $|\mathcal{M}| = 1$, then $\forall u \in \Sigma^*. \forall \varphi \in \mathrm{LTL}.\ u \models_3 \varphi = u \models_D \varphi$.*

6 Communication and decision making

Let us now describe the communication mechanism that enables local monitors to determine whether a trace is a good or a bad prefix. Recall that each monitor only sees a projection of an event to its locally observable set of actions, encoded as a set of atomic propositions, respectively.

Generally, at time t, when receiving an event σ, a monitor, M_i, will progress its current obligation, φ_i^t, into $P(\varphi_i^t, \sigma, AP_i)$, and send the result to another monitor, $M_{j \neq i}$, whenever the most urgent obligation, $\psi \in \mathrm{sus}(P(\varphi_i^t, \sigma, AP_i))$, is such that $\mathrm{Prop}(\psi) \subseteq (AP_j)$ holds, where $\mathrm{sus}(\varphi)$ is the *set of urgent subformulae* of φ and $\mathrm{Prop} : \mathrm{LTL} \to 2^{AP}$ yields the set of occurring propositions of an LTL formula.

Definition 6. *The function $\mathrm{sus} : \mathrm{LTL} \to 2^{\mathrm{LTL}}$ is inductively defined as follows:*

$$
\begin{array}{lll}
\mathrm{sus}(\varphi) = \textit{match } \varphi \textit{ with} & \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 & \to \mathrm{sus}(\varphi_1) \cup \mathrm{sus}(\varphi_2) \\
& \mid \neg\varphi' & \to \mathrm{sus}(\varphi') \\
& \mid \overline{\mathbf{X}}\varphi' & \to \{\overline{\mathbf{X}}\varphi'\} \\
& \mid \_ & \to \emptyset
\end{array}
$$

The set $\mathrm{sus}(\varphi)$ contains the past sub-formulae of φ, i.e., sub-formulae starting with a future temporal operator are discarded. It uses the fact that, in decentralised progression, $\overline{\mathbf{X}}$-operators are only introduced in front of atomic propositions. Thus, only the cases mentioned explicitly in the pattern matching need to be considered. Moreover, for

formulae of the form $\overline{\mathbf{X}}\varphi'$, i.e., starting with an $\overline{\mathbf{X}}$-operator, it is not needed to apply sus to φ' because φ' is necessarily of the form $\overline{\mathbf{X}}^d p$ with $d \geq 0$ and $p \in AP$, and does not contain more urgent formulae than $\overline{\mathbf{X}}\varphi'$. Note that, if there are several equally urgent obligations for distinct monitors, then M_i sends the formula to only one of the corresponding monitors according to a priority order between monitors. This order ensures that the delay induced by evaluating the global system specification in a decentralised fashion is bounded, as we shall see in Theorem 2. For simplicity in the following, for a set of component monitors $\mathcal{M} = \{M_1, \ldots, M_n\}$, the sending order is the natural order on the interval $[1, n]$. This choice of the local monitor to send the obligation is encoded through the function $\mathrm{Mon} : \mathcal{M} \times 2^{AP} \to \mathcal{M}$. For a monitor $M_i \in \mathcal{M}$ and a set of atomic propositions $AP' \in 2^{AP}$, $\mathrm{Mon}(M_i, AP')$ is the monitor $M_{j_{\min}}$ s.t. $j_{\min}$ is the smallest integer in $[1, n]$ s.t. there is a monitor for an atomic proposition in AP'. Formally: $\mathrm{Mon}(M_i, AP') = j_{\min} = \min\{j \in [1, n] \setminus \{i\} \mid AP' \cap AP_j \neq \emptyset\}$.

Once M_i has sent $P(\varphi_i^t, \sigma, AP_i)$, it sets $\varphi_i^{t+1} = \#$, where $\# \notin AP$ is a special symbol for which we define progression by

$$P(\#, \sigma, AP_i) = \#, \tag{3}$$

and $\forall \varphi \in \mathrm{LTL}.\ \varphi \wedge \# = \varphi$. On the other hand, whenever M_i receives a formula, $\varphi_{j \neq i}$, sent from a monitor M_j, it will add the new formula to its existing obligation, i.e., its current obligation φ_i^t will be replaced by the conjunction $\varphi_i^t \wedge \varphi_{j \neq i}$. Should M_i receive further obligations from other monitors but j, it will add each new obligation as an additional conjunct in the same manner.

Let us now summarise the above steps in the form of an explicit algorithm that describes how the local monitors operate and make decisions.

Algorithm L (*Local Monitor*). Let φ be a global system specification, and $\mathcal{M} = \{M_1, \ldots, M_n\}$ be the set of component monitors. The algorithm Local Monitor, executed on each M_i, returns $\top$ (resp. $\bot$), if $\sigma \models_D \varphi_i^t$ (resp. $\sigma \not\models_D \varphi_i^t$) holds, where $\sigma \in \Sigma_i$ is the projection of an event to the observable set of actions of the respective monitor, and φ_i^t the monitor's current local obligation.

L1. [Next goal.] Let $t \in \mathbb{N}^{\geq 0}$ denote the current time step and φ_i^t be the monitor's current local obligation. If $t = 0$, then set $\varphi_i^t := \varphi$.
L2. [Receive event.] Read next σ.
L3. [Receive messages.] Let $\{\varphi_j\}_{j \in [1,n], j \neq i}$ be the set of received obligations at time t from other monitors. Set $\varphi_i^t := \varphi_i^t \wedge \bigwedge_{j \in [1,n], j \neq i} \varphi_j$.
L4. [Progress.] Determine $P(\varphi_i^t, \sigma, AP_i)$ and store the result in φ_i^{t+1}.
L5. [Evaluate and return.] If $\varphi_i^{t+1} = \top$ return $\top$, if $\varphi_i^{t+1} = \bot$ return $\bot$.
L6. [Communicate.] Let $\Psi \subseteq \mathrm{sus}(\varphi_i^{t+1})$ be the set of most urgent obligations of φ_i^{t+1}. Send φ_i^{t+1} to monitor $\mathrm{Mon}(M_i, \cup_{\psi \in \Psi} \mathrm{Prop}(\psi))$.
L7. [Replace goal.] If in step L6 a message was sent at all, set $\varphi_i^{t+1} := \#$. Then go back to step L1. □

The input to the algorithm, σ, will usually resemble the latest observation in a consecutively growing trace, $u_i = u_i(0) \cdots u_i(t)$, i.e., $\sigma = u_i(t)$. We then have that $\sigma \models_D \varphi_i^t$ (i.e., the algorithm returns $\top$) implies that $u \models_D \varphi$ holds (resp. for $\sigma \not\models_D \varphi_i^t$).

Example 2. To see how this algorithm works, let us continue the decentralised monitoring process initiated in Example 1. Table 1 shows how the situation evolves for all

Table 1: Decentralised progression of $\varphi = \mathbf{F}(a \wedge b \wedge c)$ in a 3-component system.

t:	0	1	2	3
σ:	$\{a, b\}$	$\{a, b, c\}$	$\emptyset$	$\emptyset$
M_A:	$\varphi_A^1 = P(\varphi, \sigma, AP_A)$ $= \overline{\mathbf{X}}b \wedge \overline{\mathbf{X}}c \vee \varphi$	$\varphi_A^2 = P(\varphi_B^1 \wedge \#, \sigma, AP_A)$ $= \overline{\mathbf{X}}^2 c \vee (\overline{\mathbf{X}}b \wedge \overline{\mathbf{X}}c \vee \varphi)$	$\varphi_A^3 = P(\varphi_C^2 \wedge \#, \sigma, AP_A)$ $= \overline{\mathbf{X}}^2 b \vee (\overline{\mathbf{X}}b \wedge \overline{\mathbf{X}}c \vee \varphi)$	$\varphi_A^4 = P(\varphi_C^3 \wedge \#, \sigma, AP_A)$ $= \overline{\mathbf{X}}^3 b \vee (\overline{\mathbf{X}}b \wedge \overline{\mathbf{X}}c \vee \varphi)$
M_B:	$\varphi_B^1 = P(\varphi, \sigma, AP_B)$ $= \overline{\mathbf{X}}a \wedge \overline{\mathbf{X}}c \vee \varphi$	$\varphi_B^2 = P(\varphi_A^1 \wedge \#, \sigma, AP_B)$ $= \overline{\mathbf{X}}^2 c \vee (\overline{\mathbf{X}}a \wedge \overline{\mathbf{X}}c \vee \varphi)$	$\varphi_B^3 = P(\#, \sigma, AP_B)$ $= \#$	$\varphi_B^4 = P(\varphi_A^3 \wedge \#, \sigma, AP_B)$ $= \top$
M_C:	$\varphi_C^1 = P(\varphi, \sigma, AP_C)$ $= \varphi$	$\varphi_C^2 = P(\varphi, \sigma, AP_C)$ $= \overline{\mathbf{X}}a \wedge \overline{\mathbf{X}}b \vee \varphi$	$\varphi_C^3 = P(\varphi_A^2 \wedge \varphi_B^2 \wedge \#, \sigma, AP_C)$ $= \overline{\mathbf{X}}^2 a \wedge \overline{\mathbf{X}}^2 b \vee \varphi$	$\varphi_C^4 = P(\#, \sigma, AP_C)$ $= \#$

three monitors, when the global LTL specification in question is $\mathbf{F}(a \wedge b \wedge c)$ and the ordering between components is $A < B < C$. An evolution of M_A's local obligation, encoded as $P(\varphi_B^1 \wedge \#, \sigma, AP_A)$ (see cell M_A at $t = 1$) indicates that communication between the monitors has occurred: M_B (resp. M_A) sent its obligation to M_A (resp. to another monitor), at the end of step 0. Likewise for the other obligations and monitors. The interesting situations are marked in grey: In particular at $t = 0$, M_C is the only monitor who knows for sure that, so far, no good nor bad prefix occurred (see grey cell at $t = 0$). At $t = 1$, we have the desired situation $\sigma = \{a, b, c\}$, but because none of the monitors can see the other monitors' events, it takes another two rounds of communication until both M_A and M_B detect that, indeed, the global obligation had been satisfied at $t = 1$ (see grey cell at $t = 3$).

This example highlights a worst case *delay* between the occurrence and the detection of a good (resp. bad) trace by a good (resp. bad) prefix, caused by the time it takes for the monitors to communicate obligations to each other. This delay depends on the number of monitors in the system, and is also the upper bound for the number of past events each monitor needs to store locally to be able to progress all occurring past obligations:

Theorem 2. *Let, for any $p \in AP$, $\overline{\mathbf{X}}^m p$ be a local obligation obtained by Algorithm L executed on some monitor $M_i \in \mathcal{M}$. At any time $t \in \mathbb{N}^{\geq 0}$, $m \leq \min(|\mathcal{M}|, t + 1)$.*

Proof. For a full proof cf. [9]. Here, we only provide a sketch, explaining the intuition behind the theorem. Recall that $\overline{\mathbf{X}}$-operators are only introduced directly in front of atomic propositions according to rule (1) when M_i rewrites a propositional formula p with $p \notin AP_i$. Further $\overline{\mathbf{X}}$-operators can only be added according to rule (2) when M_i is unable to evaluate an obligation of the form $\overline{\mathbf{X}}^h p$. The interesting situation occurs when a monitor M_i maintains a set of urgent obligations of the form $\{\overline{\mathbf{X}}^h p_1, \ldots, \overline{\mathbf{X}}^j p_l\}$ with $h, j \in \mathbb{N}^{\geq 0}$, then, according to step L6 of Algorithm L, M_i will transmit the obligations to one monitor only thereby adding one additional $\overline{\mathbf{X}}$-operator to the remaining obligations: $\{\overline{\mathbf{X}}^{h+1} p_2, \ldots, \overline{\mathbf{X}}^{j+1} p_l\}$. Obviously, a single monitor cannot have more than $|\mathcal{M}| - 1$ outstanding obligations that need to be sent to the other monitors at any time t. So, the worst case delay is initiated during monitoring, if at some time *all* outstanding obligations of each monitor M_i, $i \in [1, |\mathcal{M}|]$, are of the form $\{\overline{\mathbf{X}}p_1, \ldots, \overline{\mathbf{X}}p_l\}$ with $p_1, \ldots, p_l \notin AP_i$ (i.e., the obligations are all equally urgent), in which case it takes $|\mathcal{M}| - 1$ time steps until the last one has been chosen to be sent to its respective monitor M_j. Using an ordering between components ensures here that each set of obligations will decrease in size after being transmitted once. Finally, a last monitor, M_j will receive an obligation of the form $\overline{\mathbf{X}}^{|\mathcal{M}|} p_k$ with $1 \leq k \leq l$ and $p_k \in AP_j$. □

Consequently, the monitors only need to memorise a *bounded history* of the trace read so far, i.e., the last $|\mathcal{M}|$ events.

Example 2 also illustrates the relationship to the LTL_3 semantics discussed earlier in Sec. 3. This relationship is formalised by the two following theorems stating the soundness and completeness of the algorithm.

Theorem 3. *Let* $\varphi \in \mathrm{LTL}$ *and* $u \in \Sigma^*$*, then* $u \models_D \varphi = \top/\bot \Rightarrow u \models_3 \varphi = \top/\bot$*, and* $u \models_3 \varphi = ? \Rightarrow u \models_D \varphi = ?$.

In particular, the example shows how the other direction of the theorem does not necessarily hold. Consider the trace $u = \{a, b\} \cdot \{a, b, c\}$: clearly, $u \models_3 \mathbf{F}(a \wedge b \wedge c) = \top$, but we have $u \models_D \mathbf{F}(a \wedge b \wedge c) = ?$ in our example. Again, this is a direct consequence of the delay introduced in our setting. However, Algorithm L detects all verdicts for a specification as if the system was not distributed.

Theorem 4. *Let* $\varphi \in \mathrm{LTL}$ *and* $u \in \Sigma^*$*, then* $u \models_3 \varphi = \top/\bot \Rightarrow \exists u' \in \Sigma^*.\ |u'| \leq n \wedge u \cdot u' \models_D \varphi = \top/\bot$*, where* n *is the number of components in the system.*

7 Experimental results

DECENTMON is an implementation, simulating the above distributed LTL monitoring algorithm in 1,800 LLOC, written in the functional programming language OCaml. It can be freely downloaded and run from [18]. The system takes as input multiple traces (that can be automatically generated), corresponding to the behaviour of a distributed system, and an LTL formula. Then the formula is monitored against the traces in two different modes: a) by merging the traces to a single, global trace and then using a "central monitor" for the formula (i.e., all local monitors send their respective events to the central monitor who makes the decisions regarding the trace), and b) by using the decentralised approach introduced in this paper (i.e., each trace is read by a separate monitor). We have evaluated the two different monitoring approaches (i.e., centralised vs. decentralised) using two different set-ups described in the remainder of this section.

Evaluation using randomly generated formulae. DECENTMON randomly generated 1,000 LTL formulae of various sizes in the architecture described in Example 1. How both monitoring approaches compared on these formulae can be seen in Table 2. The first column shows the size of the monitored LTL formulae. Note, our system measures formula size in terms of the operator entailment[4] inside it (state formulae excluded), e.g., $\mathbf{G}(a \wedge b) \vee \mathbf{F}c$ is of size 2. The entry |trace| denotes the average length of the traces needed to reach a verdict. For example, the last line in Table 2 says that we monitored 1,000 randomly generated LTL formulae of size 6. On average, traces were of length 11.808 when the central monitor came to a verdict, and of length 12.366 when one of the local monitors came to a verdict. The difference ratio, given in the second last column, then shows the average delay; that is, on average the traces were 1.0472 times longer in the decentralised setting. The number of messages, #msg., in the cen-

	centralised		decentralised		*diff. ratio*	
$\lvert\varphi\rvert$	|trace|	#msg.	|trace|	#msg.	|trace|	#msg.
1	1.369	4.107	1.634	0.982	1.1935	0.2391
2	2.095	6.285	2.461	1.647	1.1747	0.262
3	3.518	10.554	4.011	2.749	1.1401	0.2604
4	5.889	17.667	6.4	4.61	1.0867	0.2609
5	9.375	28.125	9.935	7.879	1.0597	0.2801
6	11.808	35.424	12.366	9.912	1.0472	0.2798

Table 2: Benchmarks for random formulae

[4] Our experiments show that this way of measuring the size of a formula is more representative of how difficult it is to progress it in a decentralised manner. Formulae of size above 6 are not realistic in practice.

tralised setting, corresponds to the number of events sent by the local monitors to the central monitor (i.e., the length of the trace times the number of components) and in the decentralised setting to the number of obligations transmitted between local monitors. What is striking here is that the amount of communication needed in the decentralised setting is ca. only 25% of the communication overhead induced by central monitoring, where local monitors need to send each event to a central monitor.

Evaluation using specification patterns. In order to evaluate our approach also at the hand of realistic LTL specifications, we conducted benchmarks using LTL formulae following the well-known LTL specification patterns ([19], whereas the actual formulae underlying the patterns are available at this site [20] and recalled in [18]). In this context, to randomly generate formulae, we proceeded as follows. For a given specification pattern, we randomly select one of the formulae associated to it. Such a formula is "parametrised" by some atomic propositions. To obtain the randomly generated formula, using the distributed alphabet, we randomly instantiate the atomic propositions.

The results of this test are reported in Table 3: for each kind of pattern (absence, existence, bounded existence, universal, precedence, response, precedence chain, response chain, constrained chain), we generated again 1,000 formulae, monitored over the same architecture as used in Example 1.

Discussion. Both benchmarks substantiate the claim that decentralised monitoring of an LTL formula can induce a much lower communication overhead compared to a centralised solution. In fact, when considering the more realistic benchmark using the specification patterns, the communication overhead was significantly lower compared to monitoring randomly generated formulae. The same holdstrue for the delay: in case of monitoring LTL formulae corresponding to specification patterns, the delay is almost negligible; that is, the local monitors detect violation/satisfaction of a monitored formula at almost the same time as a global monitor with access to all observations.

Besides the above, we conducted further experiments to determine which are the parameters that make decentralised monitoring (less) effective w.r.t. a centralised solution, and whether or not the user can control them or at least estimate them prior to monitoring. To this end, we first considered a policy change for sending messages: Under the new policy, components send messages to the central observer only when the truth values have changed w.r.t. a previous event. The experimental results generally

Table 3: Benchmarks for LTL specification patterns.

	centralised		decentralised		*diff. ratio*	
pattern	|trace|	#msg.	|trace|	#msg.	|trace|	#msg.
absence	156.17	468.51	156.72	37.94	1.0035	0.0809
existence	189.90	569.72	190.42	44.41	1.0027	0.0779
bounded existence	171.72	515.16	172.30	68.72	1.0033	0.1334
universal	97.03	291.09	97.66	11.05	1.0065	0.0379
precedence	224.11	672.33	224.72	53.703	1.0027	0.0798
response	636.28	1,908.86	636.54	360.33	1.0004	0.1887
precedence chain	200.23	600.69	200.76	62.08	1.0026	0.1033
response chain	581.20	1,743.60	581.54	377.64	1.0005	0.2165
constrained chain	409.12	1,227.35	409.62	222.84	1.0012	0.1815

vary with the size of the formulae, but the decentralised case induced only around half the number messages under this policy. Moreover, the advantage remains in favour of decentralised monitoring as the size of the local alphabets was increased. We then extended this setting by considering specific *probability distributions* for the occurrence of local propositions. As one would expect, the performance of decentralised monitoring deteriorates when the occurrence of a local proposition has a very high or a very low probability since it induces a low probability for a change of the truth value of a local proposition to occur. Similar to the first setting, as the size of local alphabets grows, the performance of decentralised monitoring improves again.

Clearly, further experiments are needed to determine the conditions under which the decentralised case unambiguously outperforms alternatives, but the above gives first indications. The detailed results are available and continuously updated at [18].

8 Conclusions and related work

This work is by no means the first to introduce an approach to monitoring the behaviour of distributed systems. For example, the *diagnosis (of discrete-event systems)* has a similar objective (i.e., detect the occurrence of a fault after a finite number of discrete steps) (cf. [21–23]). In diagnosis, however, one tries to isolate root causes for failure (i.e., identify the component in a system which is responsible for a fault). A key concept is that of *diagnosability*: a system model is diagnosable if it is always the case that the occurrence of a fault can be detected after a finite number of discrete steps. In other words, in diagnosis the model of a system, which usually contains both faulty and nominal behaviour, is assumed to be part of the problem input, whereas we consider systems more or less as a "black box". Diagnosability does not transfer to our setting, because we need to assume that the local monitors always have sufficient information to detect violation (resp. satisfaction) of a specification. Also, it is common in diagnosis of distributed systems to assert a central decision making point, even if that reflects merely a Boolean function connecting the local diagnosers' verdicts, while in our setting the local monitors directly communicate without a central decision making point.

A natural counterpart of diagnosability is that of *observability* as defined in decentralised observation [24]: a distributed system is said to be x-observable, where x ranges over different parameters such as whether local observers have finite or infinite memory available to store a trace (i.e., jointly unbounded-memory, jointly bounded-memory, locally unbounded-memory, locally finite-memory), if there exists a total function, always able to combine the local observers' states after reading some trace to a truthful verdict w.r.t. the monitored property. Again, the main difference here is that we take observability for granted, in that we assume that the system can always be monitored w.r.t. a given property, because detailed system topology or architectural information is not part of our problem input. Moreover, unlike in our setting, even in the locally-observable cases, there is still a central decision making point involved, combining the local verdicts. Note also that, to the best of our knowledge, both observation and diagnosis do not concern themselves with minimising the communication overhead needed for observing/diagnosing a distributed system.

A specific temporal logic, MTTL, for expressing properties of asynchronous multi-threaded systems has been presented in [25]. Its monitoring procedure takes as input a *safety* formula and a partially ordered execution of a parallel asynchronous system.

It then establishes whether or not there exist runs in the execution that violate the MTTL formula. While the synchronous case can be interpreted as a special case of the asynchronous one, there are some noteworthy differences between [25] and our work. Firstly, we take LTL "off-the-shelf"; that is, we do not add modalities to express properties concerning the distributed/multi-threaded nature of the system under scrutiny. On the contrary, our motivation is to enable users to conceive a possibly distributed system as a single, monolithic system by enabling them to specify properties over the outside visible behaviour only—independent of implementation specific-details, such as the number of threads or components—and to automatically "distribute the monitoring" process for such properties for them. Secondly, we address the fact that in some distributed systems it may not be possible to collect a global trace or insert a global decision making point, thereby forcing the automatically distributed monitors to communicate. But at the same time we try and keep communication at a minimum. This aspect, on the other hand, does not play a role in [25] where the implementation was tried on parallel (Java) programs which are not executed on physically separated CPUs, and where one can collect a set of global behaviours to then reason about. Finally, our setting is not restricted to *safety* formulae, i.e., we can monitor any LTL formula as long as its set of good (resp. bad) prefixes is not empty. However, we have not investigated whether or not the restriction of safety formulae is inherent to [25] or made by choice. Other recent works like [26] target physically distributed systems, but do not focus on the communication overhead that may be induced by their monitoring. Similarly, this work also mainly addresses the problem of monitoring systems which produce partially ordered traces (à la Diekert and Gastin), and introduces abstractions to deal with the combinational explosion of these traces.

To the best of our knowledge, our work is the first to address the problem of automatically distributing LTL monitors, and to introduce a decentralised monitoring approach that not only avoids a global point of observation or any form of central trace collection, but also tries to keep the number of communicated messages between monitors at a minimum. What is more, our experimental results show that this approach does not only "work on paper", but that it is feasible to be implemented. Indeed, even the expected savings in communication overhead could be observed for the set of chosen LTL formulae and the automatically generated traces, when compared to a centralised solution in which the local monitors transmit all observed events to a global monitor.

References

1. Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, pages 46–57. IEEE, 1977.
2. Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. Aspect-oriented instrumentation with GCC. In Barringer et al. [27], pages 405–420.
3. Patrick O'Neil Meredith and Grigore Rosu. Runtime verification with the RV System. In Barringer et al. [27], pages 136–152.
4. Sylvain Hallé and Roger Villemaire. Runtime verification for the web-a tutorial introduction to interface contracts in web applications. In Barringer et al. [27], pages 106–121.
5. Michael Gunzert and Andreas Nägele. Component-based development and verification of safety critical software for a brake-by-wire system with synchronous software components. In *Intl. Symp. on SE for Parallel and Distributed Systems (PDSE)*, pages 134–. IEEE, 1999.

6. Martin Lukasiewycz, Michael Glaß, Jürgen Teich, and Paul Milbredt. FlexRay schedule optimization of the static segment. In *7th IEEE/ACM Intl. Conf. on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 363–372. ACM, 2009.
7. Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Syst.*, 39:205–235, 2008.
8. Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53:58–64, February 2010.
9. Andreas Bauer and Yliès Falcone. Decentralised LTL monitoring. *arXiv:1111.5133*, 2011.
10. Axel Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann, 2003.
11. Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *Logic and Computation*, 20(3):651–674, 2010.
12. Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 20(4):14, 2011.
13. Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22:5–27, 1998.
14. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.
15. Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.
16. Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In *Conf. on Logic of Programs*, pages 196–218. Springer, 1985.
17. Nicolas Markey. Temporal logic with past is exponentially more succinct, concurrency column. *Bulletin of the EATCS*, 79:122–128, 2003.
18. DECENTMON Website. http://decentmonitor.forge.imag.fr.
19. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Intl. Conf. on Software Engineering (ICSE)*, pages 411–420. ACM, 1999.
20. Specification Patterns Website. http://patterns.projects.cis.ksu.edu/.
21. Yin Wang, Tae-Sic Yoo, and Stéphane Lafortune. New results on decentralized diagnosis of discrete event systems. October 2004.
22. Yin Wang, Tae-Sic Yoo, and Stéphane Lafortune. Diagnosis of discrete event systems using decentralized architectures. *Discrete Event Dynamic Systems*, 17:233–263, June 2007.
23. Franck Cassez. The complexity of codiagnosability for discrete event and timed systems. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *ATVA*, volume 6252 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2010.
24. Stavros Tripakis. Decentralized observation problems. In *44th IEEE Conf. Decision and Control (CDC-ECC)*, pages 6–11. IEEE, 2005.
25. Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Decentralized runtime analysis of multithreaded applications. In *20th Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006.
26. Alexandre Genon, Thierry Massart, and Cédric Meuter. Monitoring distributed controllers. In *Formal Methods (FM)*, volume 4085 of *LNCS*, pages 557–572. Springer, 2006.
27. Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. *Proc. Intl. Conf. on Runtime Verification (RV)*, volume 6418 of *LNCS*. Springer, 2010.