Decentralised LTL Monitoring
Andreas Bauer, Yliès Falcone

To cite this version:
Andreas Bauer, Yliès Falcone. Decentralised LTL Monitoring. Formal Methods in System Design, 2016, 48 (1-2), pp.48. 10.1007/s10703-016-0253-8. hal-01313730

HAL Id: hal-01313730
https://inria.hal.science/hal-01313730v1
Submitted on 10 May 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 a linear-time temporal logic (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 negligible 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. Furthermore, our experiments strengthen the argument that the algorithm performs well in a wide range of different application contexts, given by different system/communication topologies and/or system event distributions over time.

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:

$$\varphi = G(\text{speed\_low} \lor (\text{pressure\_sensor\_1\_high} \Rightarrow \text{seat\_belt\_1\_on}) \land \ldots \land (\text{pressure\_sensor\_n\_high} \Rightarrow \text{seat\_belt\_n\_on}))$$

The formula $\varphi$ 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 ($\text{pressure\_sensor\_i\_high}$), it has to be the case that the corresponding seat belt is fastened ($\text{seat\_belt\_i\_on}$).

Moreover, one can build a monitor for $\varphi$, which receives the respective sensor values 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! When such a central observation point exists in the system, we refer to the setting as centralised monitoring.

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 as minimal as possible. 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. When the requirement is given in terms of an LTL formula, we refer to this setting as decentralised LTL monitoring. The requirement could also be given in terms of an automaton, for example. Such other specifications, however, are not subject of this paper.

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, as low as possible.

In the decentralised setting, we assume that the system under scrutiny consists of a set of components $C = \{C_1, C_2, \ldots, C_n\}$, communicating on a synchronous bus acting as global clock. Each component has a local set of atomic propositions $AP_i$, emits events synchronously, and has a local monitor attached to it. Moreover, we demand for all $i, j \leq$
n with \( i \neq j \) that \( \text{AP}_i \cap \text{AP}_j = \emptyset \) holds, i.e., atomic propositions are local w.r.t. the components.\(^1\) The set of all events is \( \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_n \), where \( \Sigma_i = 2^\text{AP}_i \) 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, \( \text{AP} = \bigcup_{i \in [1,n]} \text{AP}_i \), and has events in \( \Sigma = 2^\text{AP} \). Note that, in general, \( \Sigma \neq \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_n \).

**Remark 1** 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]). Moreover, 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. What is more, conceptually similar bus systems are also playing an increasing role in industrial automation facilities, where individual production systems (e.g., robots, conveyors, cranes, etc.) are either interconnected on synchronous fieldbusses for the transmission of control signals, or share a common Programmable Logic Controller (PLC), for the exchange of a safety “heart-beat” signal, for example (cf. [9]). Typical such fieldbus systems include EtherCAT, ProfiBus, ProfiNet (also known as “Industrial Ethernet”, cf. [10]), and will include the upcoming standard for Time Sensitive Networking (TSN) by the IEEE’s 802 working group (sometimes also referred to as “Deterministic Ethernet” (cf. IEEE802.1 or [11] for an overview). One should stress again, however, that the results in this paper do not directly target FlexRay or any other specific (field-) bus system as the monitoring approach, which is being developed here, is generic.

Now, let as before \( \varphi \) 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, \( \varphi_i^t \), 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 \( u = (u_1, u_2, \ldots, u_n) \) 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., \( u = u_1(0) \cup \ldots \cup u_n(0) \cup u_1(1) \cup \ldots \cup u_n(1) \cup \ldots \cup u_n(t) \cup \ldots \cup u_n(t) \), a sequence of union sets). Note that from this point forward we will use \( u \) 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 is based on formula rewriting (also known as progress or derivation) which has been used for centralised monitoring, as seen for instance in [12–14]. The algorithm evaluates the global trace \( u \) 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 \( u_\Sigma^\omega \subseteq \Sigma^\omega \setminus \mathcal{L}(\varphi) \) (resp. \( u_\Sigma^\omega \subseteq \mathcal{L}(\varphi) \)) holds where \( \mathcal{L}(\varphi) \) is the set of infinite sequences in \( \Sigma^\omega \) described by \( \varphi \). That is, a locally observed violation (resp. satisfaction) is, in fact, a global violation (resp. satisfaction). Or, in other words, \( u \) is a bad (resp. good) prefix for \( \varphi \).

\(^1\) The assumption of atomic proposition locality is not a restriction of our approach but it simplifies the presentation.
• If centralised monitoring by progression would detect that \( u \Sigma^\omega \subseteq \Sigma^\omega \setminus \mathcal{L}(\varphi) \) (resp. \( u \Sigma^\omega \subseteq \mathcal{L}(\varphi) \)), one of the local monitors on some component \( C_i \) yields \( \varphi^t_i = \bot \) (resp. \( \varphi^t_i = \top \)), for an observation \( u'_i \), an extension of \( u_i \), the local observation of \( u \) 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 significantly reduced 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).

At this point, one should also add that this article constitutes an extended version of a conference paper. In comparison to [15], its most noteworthy additions are as follows: this paper now contains a complete formalisation of our monitoring algorithm (Sec. 6.1) as well as a proof of its correctness based on said formalisation (Sec. 6.2). Based on this formalisation, we have added a number of formal statements that help clarify the relationship between the central and the decentralised case, all of which contain full proofs. Yet we have decided to add some proofs to a separate appendix for improved readability of the paper (Appendix A). Also, we have undertaken further practical experiments with our implementation of the monitoring algorithm (Sec. 7), in order to evaluate different system topologies as well as distributions of system events. Both are interesting parameters to consider, since the experiments strengthen our argument that the monitoring algorithm performs well in a wide range of application contexts and not just those we have previously shown.

Outline. The rest of this paper is structured as follows. In the next section, we introduce basic notions and notation. 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 for this purpose. Experimental results, showing the feasibility of our approach, are presented in Sec. 7. Section 8 concludes and gives pointers to related work. Appendix A contains detailed, formal proofs of theorems.

2 Preliminaries

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

As our system operates under the perfect synchrony hypothesis (cf. [16]), 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 respective 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. [16]).

We use a projection function \( \Pi_i \) 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, \( \Pi_i : 2^{AP} \rightarrow 2^{AP} \) and we denote \( AP_i = \Pi_i(\Sigma) \) for \( i \in [1, n] \). For events, \( \Pi_i : 2^\Sigma \rightarrow 2^\Sigma \) 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 \( \Sigma \) are elements of the set \( \Sigma^* \) and are typically encoded by \( u, u', \ldots \), whereas infinite traces over \( \Sigma \) are elements of the set \( \Sigma^\omega \) and are typically encoded by \( w, w', \ldots \). The set of all traces is given by the set \( \Sigma^\omega = \Sigma^* \cup \Sigma^\omega \). The set \( \Sigma^* \setminus \{\epsilon\} \) is noted \( \Sigma^+ \). The finite or infinite sequence \( w^i \) is the suffix of the trace \( w \in \Sigma^\omega \), starting at time \( t \), i.e., \( w^i = w(t) \cdot w(t+1) \cdots \). The system’s global behaviour, \( u = (u_1, u_2, \ldots, u_n) \) can now be described as a sequence of pair-wise unions of the local events in component traces, each of which at time \( t \) is of length \( t + 1 \) i.e., \( u = u(0) \cdots u(t) \).

We monitor a system w.r.t. a global specification, expressed as an LTL [1] formula, that does not state anything about its distribution or the system’s architecture. Formulae of LTL can be described using the following grammar:

\[
\varphi ::= p \mid (\varphi) \mid \neg \varphi \mid \varphi \lor \varphi \mid X \varphi \mid \varphi U \varphi,
\]

where \( p \in AP \). Additionally, we allow the following operators, each of which is defined in terms of the above ones: \( \top = p \lor \neg p \), \( \bot = \neg \top \), \( \varphi_1 \land \varphi_2 = \neg (\neg \varphi_1 \lor \neg \varphi_2) \), \( \square \varphi = T U \varphi \), and \( G \varphi = \neg F (\neg \varphi) \). The operators typeset in bold are the temporal operators. Formulae without temporal operators are called state formulae. We describe the set of all LTL formulae over \( AP \) by the set \( LTL(AP) \), or just LTL when the set of atomic propositions is clear from the context or does not matter. For reasons of self-containedness, we also recall the formal semantics of LTL in detail as follows.

**Definition 1** ([17]) The semantics of LTL is defined w.r.t. infinite traces. Let for this purpose \( w \in \Sigma^\omega \) be an infinite trace and \( i \in \mathbb{N}^{\geq 0} \). Satisfaction of an LTL formula by \( w \) at time \( i \) is inductively defined as follows.

\[
\begin{align*}
    w^i &\models p \quad \Leftrightarrow \quad p \in w(i), \text { for any } p \in AP \\
    w^i &\models \neg \varphi \quad \Leftrightarrow \quad w^i \not\models \varphi \\
    w^i &\models \varphi_1 \lor \varphi_2 \quad \Leftrightarrow \quad w^i \models \varphi_1 \lor w^i \models \varphi_2 \\
    w^i &\models X \varphi \quad \Leftrightarrow \quad w^{i+1} \models \varphi \\
    w^i &\models \varphi U \varphi_2 \quad \Leftrightarrow \quad \exists k \in [i, \infty[. w^k \models \varphi_2 \land \forall l \in [i, k[. w^l \models \varphi_1
\end{align*}
\]

When \( w^0 \models \varphi \) holds, we also write \( w \models \varphi \) to denote the fact that \( w \) is a logical model for \( \varphi \). Finally, for some \( \varphi \in LTL(AP) \), \( \mathcal{L}(\varphi) \subseteq \Sigma^\omega \) denotes the individual models of \( \varphi \) (i.e., set of traces). A set \( L \subseteq \Sigma^\omega \) is also called a language (over \( \Sigma \)).

### 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:
Proof

By structural induction on \( \text{good}(\varphi) \) (resp. \( \text{bad}(\varphi) \)).

Although there exist a myriad of different approaches to monitoring LTL formulae,

Based on various finite-trace semantics (cf. [18]), one valid way of looking at the

monitoring problem for some formula \( \varphi \in \text{LTL} \) is the following: The monitoring problem of \( \varphi \in \text{LTL} \)

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 \( \varphi \). One monitoring approach along those lines is described

in [19]. We review an alternative monitoring procedure based on formula rewriting, which

is also known as formula progression, or just progression in the domain of planning with

temporally extended goals (cf. [20]).

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, instead of only the

expressively complete set as given in Definition 1.

Definition 3 Let \( \varphi, \varphi_1, \varphi_2 \in \text{LTL} \), and \( \sigma \in \Sigma \) be an event. Then, the progression function

\( P : \text{LTL} \times \Sigma \rightarrow \text{LTL} \) is inductively defined as follows.

\[
\begin{align*}
P(p \in AP, \sigma) & = \begin{cases}
\top & \text{if } p \in \sigma, \\
\bot & \text{otherwise}
\end{cases} \\
P(\top, \sigma) & = \top \\
P(\bot, \sigma) & = \bot \\
P(\neg \varphi, \sigma) & = \neg P(\varphi, \sigma) \\
P(X \varphi, \sigma) & = \varphi \\
P(\varphi_1 \lor \varphi_2, \sigma) & = P(\varphi_1, \sigma) \lor P(\varphi_2, \sigma) \\
P(\varphi_1 U \varphi_2, \sigma) & = P(\varphi_2, \sigma) \lor P(\varphi_1, \sigma) \land \varphi_1 U \varphi_2 \\
P(G \varphi, \sigma) & = P(\varphi, \sigma) \land G(\varphi) \\
P(F \varphi, \sigma) & = P(\varphi, \sigma) \lor F(\varphi)
\end{align*}
\]

Note that monitoring using rewriting with similar rules as above has been described, for

e.g., in [12–14], 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 \( \sigma \), as it is stated by the following lemmas.

Lemma 1 Let \( \varphi \) be an LTL formula, \( \sigma \) an event and \( w \) an infinite trace, we then have

\( \sigma \cdot w \models \varphi \iff w \models P(\varphi, \sigma) \).

Proof By structural induction on LTL formulae (see Appendix A.1).

Lemma 2 Let \( \varphi \) be an LTL formula, \( \sigma \) an event, we then have if \( P(\varphi, \sigma) = \top \), then \( \sigma \in \text{good}(\varphi) \).

Proof By structural induction on LTL formulae and using Lemma 1 (see Appendix A.1).

Let us now get back to [19], which introduces a finite-trace semantics for LTL monitoring

called \( LTL_3 \). It is captured by the following definition.

Definition 4 Let \( u \in \Sigma^* \), the satisfaction relation of \( LTL_3 \), \( \models_3 : \Sigma^* \times \text{LTL} \rightarrow \{\top, \bot, \?\} \), is defined as follows.

\[
\begin{align*}
u \models_3 \varphi & = \begin{cases}
\top & \text{if } u \in \text{good}(\varphi), \\
\bot & \text{if } u \in \text{bad}(\varphi), \\
? & \text{otherwise.}
\end{cases}
\end{align*}
\]
Based on this definition, it now becomes clear how progression could be used as a somewhat naïve monitoring algorithm for a 3-valued finite-trace semantics akin to LTL₃:

**Definition 5** Given a formula \( \varphi \in \text{LTL} \) and a trace \( u = u(0) \cdot \ldots \cdot u(t) \in \Sigma^+ \), the application of extended progression function \( \mathcal{P} \) to \( \varphi \) and \( u \) is obtained by \( t + 1 \) consecutive applications of the progression function of \( \varphi \) on \( u \):

\[
\mathcal{P}(\varphi,u(0) \cdot \ldots \cdot u(t)) = \mathcal{P}(\varphi,u) = \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\varphi,u(0)),\ldots,u(t)))\ldots))))
\]

Moreover, \( \mathcal{P}(\varphi,\epsilon) = \varphi \). The semantic relation of (centralised) monitoring by progression, \( \models_C : \Sigma^* \times \text{LTL} \to \mathbb{B}_3 \), is defined as follows.

\[
\models_C \varphi = \begin{cases} 
\top & \text{if } \mathcal{P}(\varphi,u) = \top, \\
\bot & \text{if } \mathcal{P}(\varphi,u) = \bot, \\
? & \text{otherwise.}
\end{cases}
\]

For the sake of readability, in the remainder, we overload the notation of the progression function on events to traces, i.e., \( \mathcal{P}(\varphi,u) \) is denoted \( P(\varphi,u) \).

**Theorem 1** Let \( \varphi \in \text{LTL} \) and \( u \in \Sigma^* \), then \( u \models_C \varphi = \top / \bot \Rightarrow u \models_3 \varphi = \top / \bot \), and \( u \models_3 \varphi = ? \Rightarrow u \models_C \varphi = ? \).

**Proof** The theorem can be shown by an induction based on Definitions 2–4 and Lemma 2. The formal proof can be found in Appendix A.1.

However, in comparison with the monitoring procedure for LTL₃, described in [19], the algorithm that is implied by this theorem has the disadvantage that the formula, which is being progressed, may grow in size relative to the number of events. It should be added though that in practice, the use of some simplification rules in the progression function normally prevents this problem from occurring.

**Remark 2** As progression is purely syntax-driven, one can construct pathological cases that “defeat” it, if one does not take precautions [21,22]. For example, consider \( \varphi = \text{XX}\top \). As \( \varphi \) is a tautology, every prefix of length one is already a good prefix for \( \varphi \), however, \( \mathcal{P}(\varphi,\sigma) = \text{X}\top \) for any \( \sigma \in \Sigma \), meaning that progression would detect the good prefix with a delay of one extra step. Should the tautology be syntactically more involved than that, e.g., \( \varphi = \text{G(}\top \text{U(Fb V G¬b))} \), then progression may fail to detect it completely; in this example, progression would fail to detect it unless \( b \) became true along the trace. Arguably, however, these pathological cases are more of theoretical than practical merit and seldom occur in real specifications as they usually contain redundancy. In our experiments, in which we generated literally thousands of LTL specifications, we did not encounter this problem once, for example. Moreover, one could check the syntactic closure of a given specification for tautologies, prior to monitoring without any detriment to runtime performance. Checking each such formula \( \varphi' \) of the syntactic closure would correspond to solving a PSpace-complete problem (cf. [23]).

Remark 2 also indicates that the other direction of Theorem 1 does not hold.


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 $\varphi$, 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 $t$, $\varphi_i^t$, cannot be checked using events from $\Sigma_i$ 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 $p \land Xq$, 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 $Xq$. 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 $\varphi$ 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 $AP_i$:

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

where for every $w \in \Sigma^\omega$ and $j > 0$, we have $w^j \models \overline{X}\varphi$ if and only if $w^{j-1} \models \varphi$. In other words, $\overline{X}$ is the dual to the $X$-operator, sometimes referred to as the “previously-operator” in past-time LTL (cf. [24]). To ease presentation, the formula $\overline{X}^0 \varphi$ is a short for $\overline{XX} \ldots \overline{X} \varphi$. Our operator is somewhat different to the standard use of $\overline{X}$: it can only precede an atomic proposition or an atomic proposition which is preceded by further $\overline{X}$-operators. Hence, the restricted use of the $\overline{X}$-operator does not give us the full flexibility (or succinctness gains [25]) of past-time LTL. Using the $\overline{X}$-operator, let us now formally define the urgency of a formula $\varphi$ using a pattern matching as follows:

**Definition 6** Let $\varphi$ be an LTL formula, and $\Upsilon : \text{LTL} \to \mathbb{N}^{\geq 0}$ be an inductively defined function assigning a level of urgency to an LTL formula as follows.

$$\Upsilon(\varphi) = \text{match } \varphi \text{ with } \varphi_1 \lor \varphi_2 \mid \varphi_1 \land \varphi_2 \to \max(\Upsilon(\varphi_1), \Upsilon(\varphi_2))$$

$$\mid \overline{X}\varphi' \to 1 + \Upsilon(\varphi')$$

$$\mid \bot \to 0.$$  

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

Obviously, the above modification to the progression rules has the desired effect: If $p \in \sigma$, then nothing changes, otherwise if $p \notin \sigma$, we return $\overline{X}p$ 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 due to communication 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 $\varphi = \mathbf{F}(a \land b \land c)$ needs to be monitored in a decentralised manner. Let us further assume that, initially, $\varphi_A^1 = \varphi_B^1 = \varphi_C^1 = \varphi$. 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 $\sigma$ 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 $\sigma$, is as follows:

\[
\begin{align*}
\varphi_A^1 &= P(\varphi, \{a\}, \{a\}) = P(a, \{a\}, \{a\}) \land P(b, \{a\}, \{a\}) \land P(c, \{a\}, \{a\}) \lor \varphi \\
&= \mathbf{X}b \land \mathbf{X}c \lor \varphi, \\
\varphi_B^1 &= P(\varphi, \{b\}, \{b\}) = P(a, \{b\}, \{b\}) \land P(b, \{b\}, \{b\}) \land P(c, \{b\}, \{b\}) \lor \varphi \\
&= \mathbf{X}a \land \mathbf{X}c \lor \varphi, \\
\varphi_C^1 &= P(\varphi, \emptyset, \{c\}) = P(a, \emptyset, \{c\}) \land P(b, \emptyset, \{c\}) \land P(c, \emptyset, \{c\}) \lor \varphi \\
&= \mathbf{X}a \land \mathbf{X}b \land \bot \lor \varphi = \varphi.
\end{align*}
\]

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 $\Sigma_i$, the progression of a past goal $\mathbf{X}^m \varphi$, at time $t \geq m$, is defined as follows:

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

(2)

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

5 Semantics

In the previous example, we can clearly see that monitors $M_A$ and $M_B$ cannot determine whether or not $\sigma$, if interpreted as a trace of length 1, is a good prefix for the global goal formula $\varphi$.\footnote{Note that $L(\varphi)$, being a liveness language, does not have any bad prefixes.} 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 \( \sigma \) is not a good prefix and that, as before, after time 1, \( \varphi \) 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 satisfied, then \( \sigma \) 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 7** Let \( C = \{C_1, \ldots, C_n\} \) be the set of system components, \( \varphi \in \text{LTL} \) be a global goal, and \( \mathcal{M} = \{M_1, \ldots, M_n\} \) be the set of component monitors. Further, let \( u = u_1(0) \cup \ldots \cup u_n(0) \cdot u_1(1) \cup \ldots \cup u_n(1) \cdot \ldots \cup u_1(t) \cup \ldots \cup u_n(t) \) 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 \( \varphi'_i \), \( M_i \) reports \( P(\varphi'_i, u_i(t), AP_i) = \top \) (resp. \( \bot \)), then \( u \models_D \varphi = \top \) (resp. \( \bot \)). Otherwise, \( u \models_D \varphi = ? \).

By \( \models_D \) we denote the satisfaction relation on finite traces in the decentralised setting to differentiate it from \( \models_C \), standard LTL which is defined on infinite traces, and the satisfaction relation in the centralised setting. Obviously, \( \models_3 \), \( \models_C \), and \( \models_D \) 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 conclusive 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 the definition of \( \models_C \) and Theorem 1.

### 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 \( \sigma \), a monitor, \( M_i \), will progress its current obligation, \( \varphi'_i \), into \( P(\varphi'_i, \sigma, AP_i) \), and send the result to another monitor, \( M_j \), whenever the most urgent obligation, \( \psi \in \text{sus}(\varsigma(\varphi'_i, \sigma, AP_i)) \), is such that \( \text{Prop}(\psi) \subseteq (AP_j) \) holds, where \( \text{sus}(\varphi) \) is the set of urgent subformulae of \( \varphi \) and \( \text{Prop} : \text{LTL} \rightarrow 2^{AP} \) yields the set of occurring propositions of an LTL formula.

**Definition 8** Function \( \text{sus} : \text{LTL} \rightarrow 2^{\text{LTL}} \) is inductively defined as follows:

\[
\text{sus}(\varphi) = \text{match } \varphi \text{ with } \begin{align*}
\varphi_1 \lor \varphi_2 & \rightarrow \text{sus}(\varphi_1) \cup \text{sus}(\varphi_2) \\
\neg \varphi' & \rightarrow \text{sus}(\varphi') \\
\text{X} \varphi' & \rightarrow \{ \text{X} \varphi' \} \\
\_ & \rightarrow \emptyset
\end{align*}
\]

The set \( \text{sus}(\varphi) \) contains the past sub-formulae of \( \varphi \), i.e., sub-formulae starting with a future temporal operator are discarded. It uses the fact that, in decentralised progression, \( \text{X} \)-operators are only introduced in front of atomic propositions or other \( \text{X} \)-operators. Thus,
only the cases mentioned explicitly in the pattern matching need to be considered. Moreover, for formulae of the form $\mathbf{X}\varphi'$, i.e., starting with an $\mathbf{X}$-operator, it is not necessary to apply $\text{sus}$ to $\varphi'$ because $\varphi'$ is bound to be of the form $\mathbf{X}p$ with $d \geq 0$ and $p \in AP$, and does not contain more urgent formulae than $\mathbf{X}\varphi'$. 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 $\text{Mon} : \mathcal{M} \times 2^{AP} \rightarrow \mathcal{M}$. For a monitor $M_i \in \mathcal{M}$ and a set of atomic propositions $AP' \in 2^{AP}$, $\text{Mon}(M_i, AP')$ is the monitor $M_{j_{\text{min}}}$ s.t. $j_{\text{min}}$ is the smallest integer in $[1, n]$ s.t. there is a monitor for an atomic proposition in $AP'$. Formally: 
$$\text{Mon}(M_i, AP') = j_{\text{min}} = \min\{j \in [1, n] \mid \{i\} \cap AP' \cap AP_j \neq \emptyset\}.$$ 
Once $M_j$ has sent $P(\phi_i^t, \sigma, AP_i)$, it sets $\phi_i^{t+1} = \#$, where $\# \notin AP$ is a special symbol for which we define progression by 
$$P(\# , \sigma, AP_i) = \#,$$ and $\forall \varphi \in \text{LTL}$. $\varphi \land \# = \varphi$. Moreover, whenever $M_j$ receives a formula, $\varphi_j \neq \#, \# \notin AP$ from a monitor $M_i$, it will add the new formula to its existing obligation, i.e., its current obligation $\phi_i^t$ will be replaced by the conjunction $\varphi_i^t \land \varphi_j \neq \#$. 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 $\varphi$ 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 $\varphi_i^t$ the monitor's current local obligation.

L1. [Next goal.] Let $t \in \mathbb{N} \geq 0$ denote the current time step and $\varphi_i^t$ be the monitor’s current local obligation. If $t = 0$, then set $\varphi_i^t := \varphi$.
L2. [Receive event.] Read next $\sigma$.
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 $\varphi_i^t := \varphi_i^t \land \bigwedge_{j \in [1, n], j \neq i} \varphi_j$.
L4. [Progress:] Determine $P(\varphi_i^t, \sigma, AP_i)$ and store the result in $\varphi_i^{t+1}$.
L5. [Evaluate and return.] If $\varphi_i^{t+1} = \top$ return $\top$, if $\varphi_i^{t+1} = \bot$ return $\bot$.
L6. [Communicate.] Let $\Psi \subseteq \text{sus}(\varphi_i^{t+1})$ be the set of most urgent obligations of $\varphi_i^{t+1}$ determined with function $T$ (cf. Definition 6). Send $\varphi_i^{t+1}$ to monitor $\text{Mon}(M_i, \cup_{\psi \in \Psi} \text{Prop}(\psi))$.
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, $\sigma$, 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 $\sigma \not\models_D \varphi_i^t$).
Table 1: Decentralised progression of $\varphi = F(a \land b \land c)$ in a 3-component system.

| $t$ | 0 | 1 | 2 | 3 |
|-----|---|---|---|---|
| $\sigma$ | $\{a,b\}$ | $\{a,b,c\}$ | $\emptyset$ | $\emptyset$ |
| $M_A$ | $\varphi^0_A = P(\varphi, \sigma, AP_A)$ = $\square a \land \square b \land \square c \land \varphi$ | $\varphi^1_A = P(\varphi^0_A \land \varphi, \sigma, AP_A)$ = $\square a \land \square b \land \square c \land (\varphi \land \varphi)$ | $\varphi^2_A = P(\varphi^1_A \land \varphi, \sigma, AP_A)$ = $\square a \land \square b \land \square c \land (\varphi \land \varphi \land \varphi)$ | $\varphi^3_A = P(\varphi^2_A \land \varphi, \sigma, AP_A)$ = $\square a \land \square b \land \square c \land (\varphi \land \varphi \land \varphi \land \varphi)$ |
| $M_B$ | $\varphi^0_B = P(\varphi, \sigma, AP_B)$ = $\square a \land \square b \land \square c \land \varphi$ | $\varphi^1_B = P(\varphi^0_B \land \varphi, \sigma, AP_B)$ = $\square a \land \square b \land \square c \land (\varphi \land \varphi)$ | $\varphi^2_B = P(\varphi^1_B \land \varphi, \sigma, AP_B)$ = $\square a \land \square b \land \square c \land (\varphi \land \varphi \land \varphi)$ | $\varphi^3_B = P(\varphi^2_B \land \varphi, \sigma, AP_B)$ = $\square a \land \square b \land \square c \land (\varphi \land \varphi \land \varphi \land \varphi)$ |
| $M_C$ | $\varphi^0_C = P(\varphi, \sigma, AP_C)$ = $\square a \land \square b \land \square c \land \varphi$ | $\varphi^1_C = P(\varphi, \sigma, AP_C)$ = $\square a \land \square b \land \square c \land \varphi$ | $\varphi^2_C = P(\varphi^1_C \land \varphi, \sigma, AP_C)$ = $\square a \land \square b \land \square c \land (\varphi \land \varphi)$ | $\varphi^3_C = P(\varphi^2_C \land \varphi, \sigma, AP_C)$ = $\square a \land \square b \land \square c \land (\varphi \land \varphi \land \varphi)$ |

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 three monitors, when the global LTL specification in question is $F(a \land b \land c)$, the global trace is $\{a,b\} \cdot \{a,b,c\} \cdot \emptyset \cdot \emptyset$, and the ordering between components is $A < B < C$. An evolution of $M_A$’s local obligation, encoded as $P(\varphi^3_A \land \# \# \sigma, AP_A)$ (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. $M_B$), 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 communication 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 $p \in AP$, $\square^m p$ be a local obligation obtained by Algorithm L executed on some monitor $M_i \in M$. At any time $t \in \mathbb{N}^{\geq 0}$, $m \leq \min(|M|, t+1)$.

**Proof** The formal proof is based upon the formalisation of the algorithm, given in the next section and available in full later in the paper.

Here, we only want to provide a sketch, explaining the intuition behind the theorem. Recall that $\square$-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 $\square$-operators can only be added according to rule (2) when $M_i$ is unable to evaluate an obligation of the form $\square^m p$. The interesting situation occurs when a monitor $M_i$ maintains a set of urgent obligations of the form $\{\square^h p_1, \ldots, \square^l p_i\}$ with $h,j \in \mathbb{N}^{\geq 0}$, and, according to step L6 of Algorithm L, $M_i$ will transmit the obligations to one monitor only thereby adding one additional $\square$-operator to the remaining obligations: $\{\square^{h+1} p_2, \ldots, \square^{l+1} p_i\}$. Obviously, a single monitor cannot have more than $|M| - 1$ outstanding obligations that need to be sent to the other monitors at any time $t$. So, the worst case communication delay is initiated during monitoring, if at some time all outstanding obligations of each monitor $M_i$, $i \in [1,|M|]$, are of the form $\{\square^{h_i} p_{i+1}, \ldots, \square^{l_i} p_i\}$ with $p_{i+1}, \ldots, p_i \notin AP_i$ (i.e., the obligations are all equally urgent), in which case it takes $|M| - 1$ 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 $\bigwedge_{k=1}^{l} p_k$ 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 $|M|$ events.

6.1 Algorithm formalisation

In order to prove the above result, let us first formalise Algorithm L a bit more and introduce some additional notation:

- $\text{send}(i,t,j) \in \{\text{true}, \text{false}\}$ is a predicate indicating whether or not the monitor $i$ sends a formula to monitor $j$ at time $t$ with $i \neq j$.
- $\text{send}(i,t) \in \{\text{true}, \text{false}\}$ is a predicate indicating whether or not the monitor $i$ sends a formula to some monitor at time $t$.
- $\text{kept}(i,t) \in \text{LTL}$ is the local obligation kept by monitor $i$ at time $t$ for the next round $(time t+1)$.
- $\text{received}(i,t,j) \in \text{LTL}$ is the obligation received by monitor $i$ at time $t$ by monitor $j$ with $i \neq j$.
- $\text{received}(i,t) \in \text{LTL}$ is the obligation received by monitor $i$ at time $t$ from all monitors.
- $\text{inlo}(i,t,\phi) \in \text{LTL}$ is the local obligation of monitor $i$ at time $t$ when monitoring the global specification formula $\phi$, before applying the progression function, i.e., after applying step L3 of Algorithm L.
- $\text{lo}(i,t,\phi) \in \text{LTL}$ is the local obligation of monitor $i$ at time $t$ when monitoring the global specification formula $\phi$ after applying the progression function, i.e., after applying step L4 of Algorithm L.
- $\text{mou}(\phi) \in \text{sus}(\phi)$ is the most urgent formula belonging to the set of urgent subformulae of $\phi$.
- $\text{ulo}(i,t,\phi) = \text{sus}(\text{lo}(i,t,\phi))$ is the set of urgent local obligations of monitor $i$ at time $t$ when monitoring the global specification formula $\phi$.

Based on the previous notation and Algorithm L, we have the following relations:

- $\text{send}(i,t,j)$ is true if monitor $M_j$ is the first monitor containing the most urgent obligation contained in the local obligation of $M_i$, according to the order in $[1,m]$. Formally:
  $$\text{send}(i,t,j) = \begin{cases} \text{true} & \text{if } M_j = \text{Mon}(M_i, \text{Prop(ulo}(i,t,\phi)))) \land \text{ulo}(i,t,\phi) \neq \emptyset, \\ \text{false} & \text{otherwise.} \end{cases}$$
- $\text{send}(i,t)$ is true if monitor $M_i$ sends his local obligation to some monitor. Formally:
  $$\text{send}(i,t) = \exists j \in [1,n] \setminus \{i\}. \text{send}(i,t,j).$$
- $\text{kept}(i,t) \in \text{LTL}$ is neither $\#$ if $M_i$ sends its local obligation to some monitor at time $t-1$ or its local obligation at time $t-1$ otherwise. Formally:
  $$\text{kept}(i,t) = \begin{cases} \# & \text{if } \exists j \in [1,n] \setminus \{i\}. \text{send}(i,t-1,j), \\ \text{lo}(i,t-1,\phi) & \text{otherwise.} \end{cases}$$
- $\text{received}(i,t,j)$ is the local obligation of $M_j$ received by $M_i$ at time $t$ if $t \geq 1$ and $M_j$ sends actually something to $M_i$. Formally:
  $$\text{received}(i,t,j) = \begin{cases} \text{lo}(j,t-1,\phi) & \text{if } \exists j \in [1,n] \setminus \{i\}. \text{send}(j,t-1,i) \land t \geq 1, \\ \# & \text{otherwise.} \end{cases}$$
received\((i, t)\) is the conjunction of all obligations received by monitor \(i\) from all other monitors at time \(t\). Formally:

\[
\text{received}(i, t) = \bigwedge_{j=1, j \neq i}^{[M]} \text{received}(i, t, j).
\]

inlo\((i, t, \varphi)\) is

- at time \(t \geq 1\) what was kept by \(M_i\) at time \(t-1\) and the received obligation at time \(t\);
- at time \(t = 0\) the initial obligation, i.e., the global specification \(\varphi\).

Formally:

\[
inlo(i, t, \varphi) = \begin{cases} 
\varphi & \text{if } t = 0, \\
\text{kept}(i, t-1) \land \text{received}(i, t) & \text{otherwise}
\end{cases}.
\]

lo\((i, t, \varphi)\) is

- at time \(t \geq 1\) the result of progressing what was kept by \(M_i\) at time \(t-1\) and the received obligation at time \(t\) with the current local event \(u_i(t)\);
- at time \(t = 0\) the result of progressing the initial obligation, i.e., the global specification with the current local event \(u_i(0)\).

Formally:

\[
\text{lo}(i, t, \varphi) = \begin{cases} 
P(\varphi, u_i(0), AP_i) & \text{if } t = 0, \\
P(\text{kept}(i, t-1) \land \text{received}(i, t), u_i(t), AP_i) & \text{otherwise}
\end{cases}.
\]

Now, we can clearly state the theorem as follows:

\[
\forall t \in \mathbb{N}^\geq 0. \forall \varphi \in \text{LTL}. \forall i \in [1, n]. \forall X, d \in \text{ulo}(i, t, \varphi). d \leq \min(n, t + 1).
\]

**Preliminaries to the proof.** Let us first start with some remarks. At step L3 in Algorithm L, the local obligation of a monitor \(M_i\) is defined to be \(\varphi^t_i \land \bigwedge_{j=1}^{[M]} \varphi^t_j\) where \(\varphi^t_j\) is an obligation received from monitor \(M_j\) and \(\varphi^t_i\) is the local obligation kept from time \(t-1\) (if \(t = 0\), \(\varphi^t_i = \varphi\)). Let us note that the local obligation kept by the monitor from time \(t-1\) to time \(t\), with \(t \geq 1\), are not urgent. The result should thus be established on the urgent local obligations transmitted and rewritten by local monitors. More formally, this is stated by the following lemma.

**Lemma 3** According to Algorithm L, we have:

\[
\text{ulo}(i, t, \varphi) = \bigcup_{j=1, j \neq i}^{[M]} \text{sus}(P(\text{received}(i, t), u_i(t), AP_i)).
\]

**Proof** First let us notice that the formulae kept by any monitor \(M_i\) at any time \(t\) are not urgent. Indeed, we have: \(\forall i \in [1, n]. \forall t \in \mathbb{N}^\geq 0.\)

\[
\text{sus}(\text{kept}(i, t)) = \begin{cases} 
\text{sus}(\#) & \text{if } \exists j \in [1, n] \setminus \{i\}. \text{send}(i, t, j), \\
\text{sus}(\text{lo}(i, t-1, \varphi)) & \text{if } \text{sus}(\text{lo}(i, t-1, \varphi)) = \emptyset.
\end{cases}
\]
That is \( \forall i \in [1, n], \forall t \geq 0. \) \( \text{sus}(\text{kept}(i, t)) = \emptyset. \) Thus, \( \forall i \in [1, n], \forall t \in \mathbb{N}^{\geq 0}, \forall \varphi \in \text{LTL}. \)

\[
\begin{align*}
\text{ulo}(i, t, \varphi) &= \text{sus}(P(\text{received}(i, t), u_i(t), AP_i)) \\
&= \text{sus}(P(\bigwedge_{j=1, j \neq i}^{\lvert M \rvert} \text{received}(i, t, j), u_i(t), AP_i)) \quad \text{(definition of received}(i, t, j)) \\
&= \text{sus}(\bigcup_{j=1, j \neq i}^{\lvert M \rvert} P(\text{received}(i, t), u_i(t), AP_i)) \quad \text{(progression on events)} \\
&= \bigcup_{j=1, j \neq i}^{\lvert M \rvert} \text{sus}(P(\text{received}(i, t), u_i(t), AP_i)) \quad \text{(definition of sus)}
\end{align*}
\]

\[\Box\]

Another last lemma will be needed before entering specifically into the proof. This lemma states that if a past obligation \( X^d p \) is part of a progressed formula, then the past obligation \( X^{d-1} p \) is part of its un-progressed form. More formally, this is stated by the following lemma.

**Lemma 4** Let us consider \( M = \{M_1, \ldots, M_n\} \) where each monitor \( M_i \) has a set of local atomic propositions \( AP_i = \Pi_i(\text{AP}) \) and observes the set of events \( \Sigma_i \), we have:

\( \forall i \in [1, n], \forall \sigma \in \Sigma_i, \forall \varphi \in \text{LTL}. \ \forall X^d p \in \text{sus}(P(\varphi, \sigma, AP_i)). \ d > 1 \implies X^{d-1} p \in \text{sus}(\varphi). \)

**Proof** The proof is done by structural induction on \( \varphi \in \text{LTL}. \)

### 6.2 Algorithm correctness

Example 2 also illustrates the relationship to the LTL\(_3\) and centralised semantics discussed earlier in Sec. 3. This relationship is formalised by the two following theorems stating essentially the soundness and completeness of the algorithm.

**Theorem 3** Let \( \varphi \in \text{LTL} \) and \( u \in \Sigma^* \), then \( u \models_D \varphi = \top / \bot \implies u \models_C \varphi = \top / \bot, \) and \( u \models_C \varphi = ? \implies u \models_D \varphi = ?. \)

**Proof** The proof is performed by showing that the initial obligation (the global specification) is “propagated” along monitors’ local obligations. The formal proof can be found in Appendix A.2.

**Corollary 1** Let \( \varphi \in \text{LTL} \) and \( u \in \Sigma^* \), then \( u \models_D \varphi = \top / \bot \implies u \models_3 \varphi = \top / \bot, \) and \( u \models_3 \varphi = ? \implies u \models_D \varphi = ?. \)

In particular, Example 2 shows how the other direction of Theorem 2 does not necessarily hold. Consider the trace \( u = \{a, b\} \cdot \{a, b, c\} \): clearly, \( u \models_3 F(a \land b \land c) = \top, \) but we have \( u \models_D F(a \land b \land c) = ? \) in our example. Again, this is a direct consequence of the communication delay introduced in our setting. However, Algorithm L eventually detects all verdicts for a specification as it could be done with centralised progression if the system was not distributed.

**Theorem 4** Let \( \varphi \in \text{LTL}, u \in \Sigma^*, \) and \( n \) be the number of components in the system. Whenever \( u \models_C \varphi = \top / \bot, \) we have \( \forall u' \in \Sigma^*, \ |u'| \geq n \implies u \cdot u' \models_D \varphi = \top / \bot. \)

**Proof** For a full proof of this theorem including several additional intermediate lemmas, see Appendix A.2.
7 Implementation and experimental results

7.1 DECENTMon: an OCaml benchmark for decentralised monitoring of LTL formulae

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 [26]. 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).

7.2 Experimental setup

We have evaluated the two different monitoring approaches (i.e., centralised vs. decentralised) using randomly generated LTL formulae and LTL formulae generated from specification patterns. Each of the following experiments were conducted on three architectures:

- an architecture where the alphabet is \{a, b, c\} and distributed alphabet is \{a|b\} (each atomic proposition is observed on a local component);
- an architecture where the alphabet is \{a_1, a_2, b_1, b_2, c_1, c_2\} and distributed alphabet is \{a_1|a_2|b_1|b_2|c_1|c_2\} (each atomic proposition is observed on a local component);
- an architecture where the alphabet is \{a_1, a_2, b_1, b_2, c_1, c_2\} and distributed alphabet is \{a_1, a_2|b_1, b_2|c_1, c_2\} (atomic propositions \(a_1, a_2\) (resp. \(b_1, b_2,\) and \(c_1, c_2\)) are observed on a local component);

We also extended this setting by considering specific probability distributions for the occurrence of local propositions: a so-called flipcoin distribution where each atomic proposition has probability 0.5 of occurring on each event, and Bernouilli distribution with parameters 0.1 and 0.01.

Monitoring metrics. Several monitoring metrics were used.

- The length of the trace, |trace|, needed for the monitor to reach a verdict.
- The number of messages, #msg., exchanged between monitors. In the centralised setting, it 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). In the decentralised setting, it corresponds to the number of obligations transmitted between local monitors.
- The maximal and average delay of decentralised monitoring compared to centralised monitoring.

Remark 3 We have used continuous simplification of the goal formulae in order to avoid a formula explosion problem caused by rewriting. In DECENTMon, advanced syntactic simplification rules\(^3\) were introduced and sufficient for the purpose of our experiments.

\(^3\) Compared to RuleR [14], the state-of-art rule-based runtime verification tool, for LTL specifications, our simplification function produced better results (see [26]).
Table 2: Benchmarks for random LTL formulae - flipcoin distrib.

(a) Alphabet \(\{a, b, c\}\) - Distributed alphabet \(\{a, b, c\}\)

| \(|\varphi|\) | centralised | decentralised | diff. ratio | delay |
|---|---|---|---|---|
| 1 | 1.344 | 1.642 | 1.2217 | 0.2601 | 1 3 |
| 2 | 6.673 | 7.038 | 1.0546 | 0.1324 | 1 3 |
| 3 | 12.196 | 12.694 | 1.0408 | 0.155 | 1 3 |
| 4 | 29.415 | 29.949 | 1.0181 | 0.1487 | 1 3 |
| 5 | 64.935 | 65.501 | 1.0087 | 0.1888 | 1 3 |
| 6 | 72.437 | 73.013 | 1.0079 | 0.2415 | 1 3 |

(b) Alphabet \(\{a_1, a_2, b_1, b_2, c_1, c_2\}\) - Distributed alphabet \(\{a_1, a_2, b_1, b_2, c_1, c_2\}\)

| \(|\varphi|\) | centralised | decentralised | diff. ratio | delay |
|---|---|---|---|---|
| 1 | 1.698 | 2.082 | 1.2261 | 0.2689 | 1 3 |
| 2 | 2.69 | 3.212 | 1.194 | 0.2525 | 1 3 |
| 3 | 8.19 | 8.748 | 1.0681 | 0.1786 | 1 3 |
| 4 | 10.546 | 11.368 | 1.0779 | 0.2621 | 1 3 |
| 5 | 11.284 | 11.99 | 1.0625 | 0.207 | 1 3 |
| 6 | 20.984 | 21.676 | 1.0259 | 0.2777 | 1 3 |

(c) Alphabet \(\{a_1, a_2, b_1, b_2, c_1, c_2\}\) - Distributed alphabet \(\{a_1, a_2, b_1, b_2, c_1, c_2\}\)

| \(|\varphi|\) | centralised | decentralised | diff. ratio | delay |
|---|---|---|---|---|
| 1 | 1.948 | 2.64 | 1.552 | 0.2979 | 1 3 |
| 2 | 4.194 | 5.128 | 1.2226 | 0.2161 | 1 3 |
| 3 | 6.748 | 7.844 | 1.1624 | 0.2156 | 1 3 |
| 4 | 8.4 | 9.746 | 1.1602 | 0.2114 | 1 3 |
| 5 | 10.154 | 11.512 | 1.1337 | 0.1863 | 1 3 |
| 6 | 22.536 | 24.014 | 1.0655 | 0.1812 | 1 3 |

7.3 Evaluation using randomly generated formulae

DECENTMon randomly generated 1,000 LTL formulae of various sizes in the architectures described above. Note, our system measures formula size in terms of the operator entailment inside it (state formulae excluded), e.g., \(G(a \land b) \lor Fc\) is of size 2. How both monitoring approaches compared on these formulae can be seen in Tables 2, 3, 4, and 5. For each experiment, the first column shows the size of the monitored LTL formulae. The entry |trace| denotes the average length of the traces needed to reach a verdict. For example, the last line in Table 2 (a) says that we monitored 1,000 randomly generated LTL formulae of size 6. On average, traces were of length 72.437 when the central monitor came to a verdict, and of length 73.013 when one of the local monitors came to a verdict. The difference ratio, given in the second last column, then shows the average communication delay; that is, on average the traces were 1.0079 times longer in the decentralised setting than the traces in

---

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.
Table 3: Benchmarks for random LTL formulae - flipcoin distrib. - only change.

(a) Alphabet \{a, b, c\} - Distributed alphabet \{a|b|c\}

| \(\varphi\) | centralised | decentralised | \(\text{diff. ratio}\) | delay |
|---|---|---|---|---|
| trace | #msg. | trace | #msg. |
| avg | max |
| 1 | 3.558 | 1.73 | 1.171 | 1.2455 | 0.3291 | 1 | 2 |
| 2 | 4.214 | 2.315 | 2.083 | 1.2853 | 0.4943 | 1 | 3 |
| 3 | 12.767 | 8.041 | 3.562 | 1.0823 | 0.279 | 1 | 3 |
| 4 | 31.424 | 20.529 | 9.742 | 1.0309 | 0.31 | 1 | 3 |
| 5 | 25.073 | 19.594 | 1.65 | 1.0309 | 0.5156 | 1 | 3 |
| 6 | 52 | 46.84 | 1.0151 | 0.5979 | 1 | 3 |

(b) Alphabet \{a_1, a_2, b_1, b_2, c_1, c_2\} - Distributed alphabet \{a_1|a_2|b_1|b_2|c_1|c_2\}

| \(\varphi\) | centralised | decentralised | \(\text{diff. ratio}\) | delay |
|---|---|---|---|---|
| trace | #msg. | trace | #msg. |
| avg | max |
| 1 | 4.056 | 1.93 | 1.478 | 1.3183 | 0.3643 | 1 | 2 |
| 2 | 5.636 | 2.728 | 2.246 | 1.2653 | 0.3985 | 1 | 2 |
| 3 | 10.198 | 4.942 | 4.11 | 1.1744 | 0.403 | 1 | 3 |
| 4 | 13.524 | 6.506 | 5.6 | 1.143 | 0.414 | 1 | 2 |
| 5 | 22.27 | 10.37 | 9.476 | 1.0838 | 0.4255 | 1 | 2 |
| 6 | 22.84 | 10.71 | 9.668 | 1.0931 | 0.5114 | 1 | 3 |

(c) Alphabet \{a_1, a_2, b_1, b_2, c_1, c_2\} - Distributed alphabet \{a_1|a_2|b_1|b_2|c_1|c_2\}

| \(\varphi\) | centralised | decentralised | \(\text{diff. ratio}\) | delay |
|---|---|---|---|---|
| trace | #msg. | trace | #msg. |
| avg | max |
| 1 | 3.351 | 1.55 | 1.075 | 1.2663 | 0.3207 | 1 | 2 |
| 2 | 3.942 | 2.381 | 2.056 | 1.2835 | 0.5215 | 1 | 2 |
| 3 | 5.383 | 4.844 | 3.587 | 1.1511 | 0.6663 | 1 | 2 |
| 4 | 8.404 | 10.044 | 7.834 | 1.0746 | 0.9321 | 1 | 2 |
| 5 | 11.344 | 14.681 | 11.174 | 1.0574 | 0.985 | 1 | 3 |
| 6 | 14.314 | 19.892 | 19.475 | 1.0414 | 1.3605 | 1 | 3 |

the centralised setting. What is striking here, however, 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.

7.4 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 ([27], whereas the actual formulae underlying the patterns are available at this site [28] and recalled in [26]). 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 “parameterised” by some atomic propositions. To obtain the randomly generated formula, using the distributed alphabet, we randomly instantiate the atomic propositions.
The results of this evaluation are reported in Tables 6, 7, 8, 9: for each setting, 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 three architectures.

7.5 Discussion

Generally speaking, 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 holds true for the communication delay: in case of monitoring LTL formulae corresponding to specification patterns, the communication 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.
Table 5: Benchmarks for random LTL formulae - Bernouilli distr. 0.01 - only change.

(a) Alphabet \{a, b, c\} - Distributed alphabet \{a|b,c\}.

| \(\phi\) | centralised | decentralised | diff ratio | delay |
|-----|-------------|---------------|------------|-------|
|     | [trace]     | #msg.         | [trace]    | #msg. | avg | max |
| 1   | 1.226       | 3.363         | 1.289      | 1.3278| 0.3832| 1 | 2 |
| 2   | 2.78        | 3.644         | 2.423      | 1.1751| 0.6649| 1 | 2 |
| 3   | 6.091       | 3.992         | 6.382      | 1.095 | 1.3481| 1 | 3 |
| 4   | 10.548      | 4.416         | 10.397     | 1.0631| 2.3543| 1 | 3 |
| 5   | 14.056      | 4.765         | 14.493     | 1.0508| 3.0415| 1 | 3 |
| 6   | 19.463      | 5.004         | 20.114     | 1.0334| 4.7402| 1 | 3 |

(b) Alphabet \{a_1, a_2, b_1, b_2, c_1, c_2\} - Distributed alphabet \{a_1|a_2\|b_1\|b_2\|c_1\|c_2\}.

| \(\phi\) | centralised | decentralised | diff ratio | delay |
|-----|-------------|---------------|------------|-------|
|     | [trace]     | #msg.         | [trace]    | #msg. | avg | max |
| 1   | 1.236       | 3.552         | 1.544      | 1.411 | 0.4346| 1 | 2 |
| 2   | 2.204       | 4.3           | 2.916      | 2.815 | 0.7032| 1 | 2 |
| 3   | 3.992       | 4.652         | 5.835      | 6.69  | 1.1688| 1 | 3 |
| 4   | 10.604      | 5.353         | 12.076     | 1.0807| 2.2559| 1 | 3 |
| 5   | 12.473      | 5.733         | 17.651     | 1.0695| 3.0788| 1 | 3 |
| 6   | 19.443      | 6.38          | 20.256     | 1.0418| 4.1960| 1 | 3 |

(c) Alphabet \{a_1, a_2, b_1, b_2, c_1, c_2\} - Distributed alphabet \{a_1|a_2\|b_1\|b_2\|c_1\|c_2\}.

| \(\phi\) | centralised | decentralised | diff ratio | delay |
|-----|-------------|---------------|------------|-------|
|     | [trace]     | #msg.         | [trace]    | #msg. | avg | max |
| 1   | 1.205       | 6.669         | 2.053      | 3.889 | 1.7037| 0.5831| 1 | 4 |
| 2   | 2.752       | 7.411         | 3.994      | 6.861 | 1.4513| 0.9257| 1 | 5 |
| 3   | 6.75        | 8.233         | 8.357      | 15.632| 1.238 | 1.6557| 1 | 6 |
| 4   | 10.329      | 8.798         | 12.042     | 18.158| 1.1658| 2.0638| 1 | 5 |
| 5   | 13.723      | 9.247         | 15.537     | 23.124| 1.1321| 2.5007| 1 | 6 |
| 6   | 17.732      | 9.862         | 19.613     | 29.316| 1.106 | 2.9726| 1 | 6 |

Let us consider the settings with traces where local propositions have the same probability of occurring as of not occurring (flipcoin distribution).

- When the policy is such that, in the centralised case, components report the truth value of atomic propositions at each time step (see Tables 2 and 6), the experimental results are clearly in favour of decentralised monitoring. Decentralised monitoring is more effective as the size of formulae grows (until formulae of size 3-4). The performance of decentralised monitoring seems to deteriorate with formulae with a nesting level above 4. We explain this phenomenon by the fact that decentralised monitoring generate more complicated obligations and our syntactic simplification techniques handles semantically equivalent formulae until 3 nesting levels.

- When the policy is such that, in the centralised case, components report the truth value of atomic propositions only in case of a change w.r.t. the previous event (see Tables 3 and 7), the experimental results generally 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.
Table 6: Benchmarks for LTL specification patterns - flipcoin distrib.

(a) Alphabet \(\{a, b, c\}\) - Distributed alphabet \(\{a|b|c\}\).

| pattern          | centralised | decentralised | diff ratio | delay  |
|------------------|-------------|---------------|------------|--------|
|                  | #msg | #msg | #msg | #msg | avg  | max  |
| absence          | 136.56 409.68 | 137.16 27.451 | 1.0043 0.067 | 0.597 2 |
| existence        | 189.27 567.81 | 189.77 38.737 | 1.0026 0.0682 | 0.501 3 |
| bounded existence| 157.20 471.60 | 157.88 62.96 | 1.0043 0.1335 | 0.682 3 |
| universal        | 93.64 290.92 | 94.29 5.952 | 1.0069 0.0211 | 0.65 2 |
| precedence       | 294.93 749.80 | 250.48 55.85 | 1.0022 0.0744 | 0.552 3 |
| response         | 636.44 1909.32 | 636.71 386.11 | 1.0004 0.2022 | 0.27 3 |
| precedence chain | 198.96 596.89 | 199.47 54.23 | 1.0025 0.0908 | 0.506 3 |
| response chain   | 573.92 1721.78 | 574.26 374.22 | 1.0005 0.2173 | 0.334 3 |
| constrained chain| 385.96 1156.22 | 385.96 198.84 | 1.0014 0.1719 | 0.556 2 |

(b) Alphabet \(\{a_1, a_2, b_1, b_2, c_1, c_2\}\) - Distributed alphabet \(\{a_1|b_1|b_2|c_1|c_2\}\).

| pattern          | centralised | decentralised | diff ratio | delay  |
|------------------|-------------|---------------|------------|--------|
|                  | #msg | #msg | #msg | #msg | avg  | max  |
| absence          | 146.49 439.47 | 147.23 38.58 | 1.005 0.0877 | 0.744 2 |
| existence        | 203.20 609.81 | 203.91 57.41 | 1.0026 0.1418 | 0.62 3 |
| bounded existence| 214.34 643.03 | 214.96 91.20 | 1.0037 0.0761 | 0.658 2 |
| universal        | 82.93 248.79 | 83.48 17.27 | 1.011 0.0694 | 0.914 2 |
| precedence       | 176.61 529.85 | 177.27 40.33 | 1.0037 0.0761 | 0.658 2 |
| response         | 632.56 1897.68 | 632.93 394.87 | 1.0005 0.208 | 0.368 2 |
| precedence chain | 181.84 545.53 | 182.43 52.69 | 1.0032 0.0965 | 0.592 3 |
| response chain   | 584.28 1752.84 | 584.70 434.80 | 1.0007 0.248 | 0.426 2 |
| constrained chain| 376.63 1129.89 | 377.33 249.71 | 1.0018 0.221 | 0.706 3 |

(c) Alphabet \(\{a_1, a_2, b_1, b_2, c_1, c_2\}\) - Distributed alphabet \(\{a_1|b_1|b_2|c_1|c_2\}\).

| pattern          | centralised | decentralised | diff ratio | delay  |
|------------------|-------------|---------------|------------|--------|
|                  | #msg | #msg | #msg | #msg | avg  | max  |
| absence          | 159.37 956.22 | 160.73 73.76 | 1.0065 0.0771 | 1.36 6 |
| existence        | 222.57 1329.45 | 222.9 92.29 | 1.0059 0.0694 | 1.324 5 |
| bounded existence| 276.94 1607.67 | 269.11 171.242 | 1.0043 0.1065 | 1.166 5 |
| universal        | 73.62 441.75 | 75.44 25.86 | 1.0246 0.0585 | 1.814 5 |
| precedence       | 187.22 1123.34 | 188.66 67.78 | 1.0076 0.0603 | 1.438 6 |
| response         | 690.02 4140.14 | 690.73 521.52 | 1.001 0.1259 | 0.71 5 |
| precedence chain | 171.25 1027.54 | 172.62 83.36 | 1.0079 0.0811 | 1.364 5 |
| response chain   | 588.74 3532.48 | 589.66 489.67 | 1.0015 0.1386 | 0.912 5 |
| constrained chain| 364.86 2189.17 | 366.37 300.82 | 1.0041 0.1374 | 1.512 6 |

Let us consider the settings with traces where the occurrence of a local proposition has a very high or a very low probability (see Tables 4, 5, 8, and 9)\(^5\). In this setting, we still favour centralised monitoring by considering that, in case of centralised monitoring, components send the value of atomic propositions only if there is change w.r.t. previous events. As one would expect, the performance of decentralised monitoring deteriorates since it induces a

\(^5\) We conducted benchmarks (not reported here) with traces generated with the Bernouilli probability distributions with parameters 0.9 and 0.99 and the results followed the same trends.
Table 7: Benchmarks for LTL specification patterns - flipcoin distrib. - only change.

(a) Alphabet \{a, b, c\} - Distributed alphabet \{a|b|c\}.

| pattern            | centralised | decentralised | diff ratio | delay | avg | max |
|--------------------|-------------|---------------|------------|-------|-----|-----|
| absence            | 140.74      | 212.67        | 1.0041     | 0.1586| 0.589| 2   |
| existence          | 181.23      | 273.56        | 1.0029     | 0.1212| 0.537| 3   |
| bounded existence  | 177.91      | 268.37        | 1.003      | 0.212 | 0.54 | 2   |
| universal          | 89.86       | 135.99        | 1.0077     | 0.0645| 0.692| 3   |
| precedence         | 239.84      | 361.54        | 1.002      | 0.1349| 0.49 | 2   |
| response chain     | 617.4       | 927.87        | 1.0004     | 0.3862| 0.287| 3   |
| response chain     | 588.65      | 883.89        | 1.0006     | 0.4353| 0.359| 3   |
| constrained chain  | 382.07      | 574.46        | 1.0013     | 0.3667| 0.534| 2   |

(b) Alphabet \{a_1, a_2, b_1, b_2, c_1, c_2\} - Distributed alphabet \{a_1|a_2|b_1|b_2|c_1|c_2\}.

| pattern            | centralised | decentralised | diff ratio | delay | avg | max |
|--------------------|-------------|---------------|------------|-------|-----|-----|
| absence            | 130.23      | 293.75        | 1.005      | 0.1431| 0.692| 2   |
| existence          | 189.83      | 426.33        | 1.0037     | 0.1294| 0.718| 3   |
| bounded existence  | 265.51      | 597.78        | 1.012      | 0.1809| 0.588| 2   |
| universal          | 83.36       | 188.24        | 1.0116     | 0.0959| 0.972| 2   |
| precedence         | 228.54      | 515.43        | 1.0028     | 0.1138| 0.657| 3   |
| response chain     | 671.2       | 1,511.66      | 1.0005     | 0.286 | 0.34 | 2   |
| response chain     | 600.9       | 1,352.32      | 1.0007     | 0.3285| 0.436| 3   |
| constrained chain  | 349.41      | 786.06        | 1.0021     | 0.2956| 0.762| 3   |

(c) Alphabet \{a_1, a_2, b_1, b_2, c_1, c_2\} - Distributed alphabet \{a_1|a_2|b_1|b_2|c_1|c_2\}.

| pattern            | centralised | decentralised | diff ratio | delay | avg | max |
|--------------------|-------------|---------------|------------|-------|-----|-----|
| absence            | 122.70      | 370.93        | 1.011      | 0.182 | 1.419| 5   |
| existence          | 195.22      | 590.00        | 1.0065     | 0.0694| 1.324| 5   |
| bounded existence  | 231.20      | 695.66        | 1.0051     | 0.2196| 1.88 | 5   |
| universal          | 86.65       | 262.6         | 1.0206     | 0.1667| 1.79 | 6   |
| precedence         | 220.53      | 665.01        | 1.0061     | 0.1489| 1.359| 6   |
| response chain     | 669.78      | 2,012.61      | 1.0001     | 0.2636| 0.675| 5   |
| response chain     | 181.91      | 548.66        | 1.0075     | 0.1621| 1.376| 5   |
| constrained chain  | 608.81      | 1,830.5       | 1.0014     | 0.2814| 0.884| 5   |

low probability for 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.

22
Table 8: Benchmarks for LTL specification patterns - Bernouilli distrib. 0.1 - only change.

(a) Alphabet \{a, b, c\} - Distributed alphabet \{a|b|c\}.

| Pattern          | Centralised | Decentralised | Diff. ratio | Delay |
|------------------|-------------|---------------|-------------|-------|
|                  | [trace]     | #msg.         | [trace]     | #msg. | avg | max |
| Absence          | 205.864     | 114.083       | 206.373     | 46.29 | 1.0024 | 0.4057 | 0.509 | 2 |
| Existence        | 209.101     | 116.307       | 209.616     | 44.902 | 1.0024 | 0.386 | 0.515 | 2 |
| Bounded existence| 240.913     | 132.553       | 241.395     | 101.907 | 1.002 | 0.7688 | 0.482 | 2 |
| Universal        | 115.629     | 64.682        | 116.269     | 19.286 | 1.0055 | 0.2981 | 0.64 | 2 |
| Precedence       | 275.837     | 151.444       | 276.357     | 58.52 | 1.0018 | 0.3864 | 0.52 | 2 |
| Response         | 674.052     | 367.017       | 674.294     | 382.044 | 1.0003 | 1.0409 | 0.245 | 2 |
| Precedence chain | 262.873     | 144.468       | 263.324     | 87.233 | 1.0017 | 0.6038 | 0.451 | 2 |
| Response chain   | 600.545     | 327.678       | 600.884     | 364.14 | 1.0005 | 1.1112 | 0.339 | 2 |
| Constrained chain| 414.809     | 226.17        | 415.331     | 251.869 | 1.0012 | 1.1163 | 0.522 | 2 |

(b) Alphabet \{a_1, a_2, b_1, b_2, c_1, c_2\} - Distributed alphabet \{a_1|a_2|b_1|b_2|c_1|c_2\}.

| Pattern          | Centralised | Decentralised | Diff. ratio | Delay |
|------------------|-------------|---------------|-------------|-------|
|                  | [trace]     | #msg.         | [trace]     | #msg. | avg | max |
| Absence          | 230.052     | 247.7861      | 230.61      | 100.18 | 1.0024 | 0.4041 | 0.558 | 2 |
| Existence        | 268.23      | 288.80        | 268.45      | 106.454 | 1.0022 | 0.3686 | 0.607 | 2 |
| Bounded existence| 279.834     | 301.00        | 280.35      | 170.31 | 1.0018 | 0.5658 | 0.524 | 2 |
| Universal        | 160.29      | 174.023       | 161.11      | 36.879 | 1.0051 | 0.2119 | 0.82 | 2 |
| Precedence       | 304.71      | 327.66        | 305.25      | 119.57 | 1.0017 | 0.3649 | 0.536 | 3 |
| Response         | 703.48      | 757.00        | 706.79      | 475.053 | 1.0004 | 0.6275 | 0.307 | 2 |
| Precedence chain | 234.17      | 253.82        | 234.67      | 111.248 | 1.0021 | 0.4382 | 0.506 | 2 |
| Response chain   | 643.90      | 689.22        | 644.26      | 486.95 | 1.0006 | 0.7065 | 0.36 | 3 |
| Constrained chain| 449.05      | 482.39        | 449.66      | 348.43 | 1.0013 | 0.7223 | 0.609 | 3 |

(c) Alphabet \{a_1, a_2, b_1, b_2, c_1, c_2\} - Distributed alphabet \{a_1|a_2|b_1|b_2|c_1|c_2\}.

| Pattern          | Centralised | Decentralised | Diff. ratio | Delay |
|------------------|-------------|---------------|-------------|-------|
|                  | [trace]     | #msg.         | [trace]     | #msg. | avg | max |
| Absence          | 70.216      | 90.342        | 71.02       | 44.054 | 1.0114 | 0.4876 | 0.804 | 5 |
| Existence        | 70.424      | 89.15         | 71.402      | 49.974 | 1.0138 | 0.5381 | 0.978 | 5 |
| Bounded existence| 349.78      | 383.04        | 350.64      | 260.72 | 1.0024 | 0.6807 | 0.864 | 5 |
| Universal        | 40.37       | 53.984        | 41.932      | 28.07 | 1.0386 | 0.5199 | 1.562 | 6 |
| Precedence       | 78.412      | 98.72         | 79.352      | 53.302 | 1.0119 | 0.5399 | 0.94 | 5 |
| Response         | 148.25      | 181.846       | 148.774     | 133.324 | 1.0035 | 0.7331 | 0.524 | 5 |
| Precedence chain | 58.66       | 75.23         | 59.6        | 44.486 | 1.016 | 0.5913 | 0.94 | 5 |
| Response chain   | 140.366     | 172.744       | 140.916     | 119.324 | 1.0039 | 0.6907 | 0.55 | 2 |

8 Conclusions and related work

Centralised monitoring of LTL requirements has been addressed by using automata-based techniques (cf. [22,19]) or rewriting-based techniques (cf. [12–14] for instance). While, automata-based approaches do not suffer from the “incompleteness problem” of rewriting-based techniques as illustrated in Remark 2, we intentionally chose to have a rewriting-based monitoring algorithm as it confers several advantages in the decentralised setting (see also the end of the discussion section).
Table 9: Benchmarks for LTL specification patterns - Bernouilli distrib. 0.01 - only change.

(a) Alphabet \{a, b, c\} - Distributed alphabet \{a|b,c\}.

| pattern          | centralised | decentralised | diff ratio | delay |
|------------------|-------------|---------------|------------|-------|
|                  | #msg.       | #msg.         |            |       |
|                  | [trace]     | [trace]       | avg | max  |
| absence          | 93.93       | 9.314         | 97.162     | 28.2894 |
| existence        | 102.9       | 9.406         | 103.136    | 41.52  |
| bounded existence| 512.44      | 34.33         | 512.67     | 309.31 |
| universal        | 63.664      | 7.794         | 60.092     | 22.942 |
| precedence       | 91.958      | 8.952         | 92.222     | 32.656 |
| response         | 155.262     | 13.022        | 155.372    | 105.336|
| bounded existence| 549.37      | 69.11         | 549.68     | 404.98 |
| universal        | 83.972      | 13.874        | 84.222     | 42.69  |
| precedence       | 102.132     | 16.386        | 102.376    | 68.208 |
| response         | 169.29      | 24.878        | 169.352    | 153.266|
| bounded existence| 107.12      | 17.716        | 107.62     | 86.134 |
| universal        | 110.974     | 20.636        | 111.452    | 92.148 |
| precedence       | 532.72      | 71.74         | 533.23     | 455.47 |
| response         | 76.602      | 17.55         | 77.692     | 54.562 |
| bounded existence| 97.616      | 19.458        | 98.196     | 71.196 |
| universal        | 166.492     | 28.322        | 166.662    | 176.488|
| precedence       | 98.04       | 19.01         | 98.608     | 90.046 |
| response         | 162.5       | 28.084        | 162.804    | 160.982|
| bounded existence| 149.92      | 25.952        | 150.38     | 173.814|

(b) Alphabet \{a_1, a_2, b_1, b_2, c_1, c_2\} - Distributed alphabet \{a_1|b_1, b_2|c_1, c_2\}.

| pattern          | centralised | decentralised | diff ratio | delay |
|------------------|-------------|---------------|------------|-------|
|                  | #msg.       | #msg.         |            |       |
|                  | [trace]     | [trace]       | avg | max  |
| absence          | 110.68      | 17.008        | 110.93     | 75.17 |
| existence        | 114.256     | 17.716        | 114.498    | 59.954|
| bounded existence| 549.37      | 69.11         | 549.68     | 404.98|
| universal        | 83.972      | 13.874        | 84.222     | 42.69 |
| precedence       | 102.132     | 16.386        | 102.376    | 68.208|
| response         | 169.29      | 24.878        | 169.352    | 153.262|
| bounded existence| 103.198     | 15.93         | 103.426    | 85.126|
| universal        | 161.816     | 23.466        | 161.976    | 150.63|
| precedence       | 147.836     | 22.136        | 148.06     | 143.086|
| response         | 166.492     | 28.322        | 166.662    | 176.488|
| bounded existence| 107.12      | 20.18         | 107.62     | 86.134|
| universal        | 110.974     | 20.636        | 111.452    | 92.148|
| precedence       | 532.72      | 71.74         | 533.23     | 455.47 |
| response         | 76.602      | 17.55         | 77.692     | 54.562|
| bounded existence| 97.616      | 19.458        | 98.196     | 71.196|
| universal        | 166.492     | 28.322        | 166.662    | 176.488|
| precedence       | 98.04       | 19.01         | 98.608     | 90.046|
| response         | 162.5       | 28.084        | 162.804    | 160.982|
| bounded existence| 149.92      | 25.952        | 150.38     | 173.814|

(c) Alphabet \{a_1, a_2, b_1, b_2, c_1, c_2\} - Distributed alphabet \{a_1|a_2|b_1,b_2|c_1,c_2\}.

| pattern          | centralised | decentralised | diff ratio | delay |
|------------------|-------------|---------------|------------|-------|
|                  | #msg.       | #msg.         |            |       |
|                  | [trace]     | [trace]       | avg | max  |
| absence          | 107.12      | 20.18         | 107.62     | 86.134|
| existence        | 110.974     | 20.636        | 111.452    | 92.148|
| bounded existence| 532.72      | 71.74         | 533.23     | 455.47 |
| universal        | 76.602      | 17.55         | 77.692     | 54.562|
| precedence       | 97.616      | 19.458        | 98.196     | 71.196|
| response         | 166.492     | 28.322        | 166.662    | 176.488|
| bounded existence| 98.04       | 19.01         | 98.608     | 90.046|
| universal        | 162.5       | 28.084        | 162.804    | 160.982|
| precedence       | 149.92      | 25.952        | 150.38     | 173.814|

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. [29–31]). 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 diagnostics: 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 diagnosters’ 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 [32]: 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 attempting to minimise the communication overhead needed for observing/diagnosing a distributed system.

A specific temporal logic, $\mathcal{MTT}$, for expressing properties of asynchronous multi-threaded systems has been presented in [33]. 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 $\mathcal{MTT}$ formula. While the synchronous case can be interpreted as a special case of the asynchronous one, there are some noteworthy differences between [33] 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-dets, 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 reduce overall communication overhead. This aspect, on the other hand, does not play a role in [33] 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 [33] or made by choice. Other recent works like [34] 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. More-
over, 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 gen-
erated traces, when compared to a centralised solution in which the local monitors transmit
all observed events to a global monitor.

Subsequent to our work, [35] alternatively studies decentralised monitoring of finite-
state automata. While the monitoring algorithm and communication protocol in [35] make
use of smaller messages and have a lower memory consumption, decentralised monitoring of
LTL formulae keeps several advantages over [35]. First, in terms of monitoring metrics, our
approach imposes lower delays to reach the verdict and requires a lower number of messages
between the monitors. Second, the approach in [35] requires monitors to exchange chunks of
(incomplete) global traces, which, from a security perspective, is a confidentiality downside.
On the contrary, our approach better “hides” what happens on local components within the
exchanged (LTL) obligations. Third, our approach is formally-proven sound and complete.

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. [36], pages
405–420.
3. Patrick O’Neil Meredith and Grigore Rosu. Runtime verification with the RV System. In Barringer et al.
[36], pages 136–152.
4. Sylvain Halé and Roger Villemaire. Runtime verification for the web-a tutorial introduction to interface
contracts in web applications. In Barringer et al. [36], 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 Luksiewycz, Michael Glädl, 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. Raimond Pegan and Mark Metter. Automating with PROFINET: Industrial Communication Based on
Industrial Ethernet. Wiley-VCH, 2008.
10. Max Fels. Real-time ethernet–industry prospective. Proceedings of the IEEE, 93(6):1118–1129, 2005.
11. Ramon Serna Oliver, Silvius S Craciunas, and Georg Stoger. Analysis of deterministic ethernet scheduling
for the industrial internet of things. In Computer Aided Modeling and Design of Communication Links
and Networks (CAMAD). 2014 IEEE 19th International Workshop on, pages 320–324. IEEE, 2014.
12. Klaus Havelund and Grigore Rosu. Monitoring programs using rewriting. In 16th IEEE International
Conference on Automated Software Engineering (ASE 2001), pages 135–143, 2001.
13. Grigore Rosu and Klaus Havelund. Rewriting-based techniques for runtime verification. Automated
Software Engineering, 12(2):151–197, 2005.
14. 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.
15. Andreas Bauer and Yliès Falcone. Decentralised LTL monitoring. In D. Giannakopoulou and D. Mery,
editors, Proceedings of the 18th International Symposium on Formal Methods (FM), volume 7436 of
Lecture Notes in Computer Science, pages 85–100, Berlin, Heidelberg, August 2012. Springer-Verlag.
16. Axel Jantsch. Modeling Embedded Systems and SoC’s: Concurrency and Time in Models of Computa-
tion. Morgan Kaufmann, 2003.
A Proofs

A.1 Proofs for Sec. 3

A.1.1 Proof of Lemma 1 (p. 6)

The following inductive proof follows the argument conveyed by Proposition 3 of [20]. For completeness sake, here we want to give the complete, formal, detailed proof.

The lemma is a direct consequence of the semantics of LTL and the definition of progression (Definition 2). Recall that this lemma states that the progression function “mimics” LTL semantics on some event $\sigma$. 

27
Proof: We shall prove the following statement:
\[ \forall \sigma \in \Sigma, \forall w \in \Sigma^\omega, \forall \varphi \in \text{LTL}, \sigma \cdot w \models \varphi \iff w \models P(\varphi, \sigma). \]

Let us consider an event \( \sigma \in \Sigma \) and an infinite trace \( w \in \Sigma^\omega \), the proof is done by a structural induction on \( \varphi \in \text{LTL} \).

**Base Case:** \( \varphi \in \{ \top, \bot, p \in \text{AP} \} \).
- Case \( \varphi = \top \): This case is trivial since, according to the definition of the progression function, \( \forall \sigma \in \Sigma, P(\top, \sigma) = \top \). Moreover, according to the LTL semantics of \( \top \), \( \forall w \in \Sigma^\omega, w \models \top \).
- Case \( \varphi = \bot \): This case is symmetrical to the previous one.
- Case \( \varphi = p \in \text{AP} \): Recall that, according to the progression function for atomic propositions, we have \( P(p, \sigma) = \top \) if \( p \in \sigma \) and \( \bot \) otherwise.
  - Let us suppose that \( \sigma \cdot w \models p \). According to the LTL semantics of atomic propositions, it means that \( p \in \sigma \), and thus \( P(p, \sigma) = \top \). And, due to the LTL semantics of \( \top \), we have \( \forall w \in \Sigma^\omega, w \models \top \).
  - Let us suppose that \( w \models P(p, \sigma) \). Since \( P(p, \sigma) \in \{ \top, \bot \} \), we have necessarily \( P(p, \sigma) = \top \).

According to the progression function, \( P(p, \sigma) = \top \) amounts to \( p \in \sigma \). Using the LTL semantics of atomic propositions, we deduce that \( \sigma \cdot w \models p \).

**Induction Case:** \( \varphi \in \{ \neg \varphi', \varphi_1 \lor \varphi_2, \varphi_1 \land \varphi_2, G \varphi', P \varphi', X \varphi', \varphi_1 U \varphi_2 \} \). Our induction hypothesis states that the lemma holds for some formulas \( \varphi', \varphi_1, \varphi_2 \in \text{LTL} \).

- Case \( \varphi = \neg \varphi' \): On the one hand, using the progression function for \( \neg \), we have \( P(\neg \varphi', \sigma) = \neg P(\varphi', \sigma) \). On the other hand, using the LTL semantics of operator \( \neg \), we have \( w \models \neg \varphi \iff \neg \varphi \). Thus, we have \( \sigma \cdot w \models \neg \varphi \iff \sigma \cdot w \not\models \varphi' \) (induction hypothesis on \( \varphi' \)) \( w \models P(\varphi', \sigma) \) if \( w \models \neg P(\varphi', \sigma) \) if \( w \models P(\neg \varphi', \sigma) \).
- Case \( \varphi = \varphi_1 \lor \varphi_2 \): Recall that, according to the progression function for operator \( \lor \), we have \( P(\varphi_1 \lor \varphi_2, \sigma) = P(\varphi_1, \sigma) \lor P(\varphi_2, \sigma) \).
  - Let us suppose that \( \sigma \cdot w \models \varphi_1 \lor \varphi_2 \). We distinguish again two sub-cases: \( \varphi_1 \lor \varphi_2 = \top \) or \( \varphi_1 \lor \varphi_2 \neq \top \). If \( \varphi_1 \lor \varphi_2 = \top \), then this case reduces to the case where \( \varphi = \top \), already treated. If \( \varphi_1 \lor \varphi_2 \neq \top \), it means that either \( \sigma \cdot w \models \varphi_1 \) or \( \sigma \cdot w \models \varphi_2 \). Let us treat the case where \( \sigma \cdot w \models \varphi_1 \) (the other case is similar). From \( \sigma \cdot w \models \varphi_1 \), we can apply the induction hypothesis on \( \varphi_1 \) to obtain \( w \models P(\varphi_1, \sigma) \), then \( w \models P(\varphi_1, \sigma) \lor P(\varphi_2, \sigma) = P(\varphi_1 \lor \varphi_2, \sigma) \).
  - Let us suppose that \( w \models P(\varphi_1 \lor \varphi_2, \sigma) = P(\varphi_1, \sigma) \lor P(\varphi_2, \sigma) \). We distinguish again two sub-cases: \( P(\varphi_1, \sigma) = \top \) or \( P(\varphi_1 \lor \varphi_2, \sigma) \neq \top \).
    - If \( P(\varphi_1, \sigma) = \top \), then we again distinguish two sub-cases:
      - If \( P(\varphi_1, \sigma) = \top \) or \( P(\varphi_2, \sigma) = \top \), let us treat the case where \( P(\varphi_1, \sigma) = \top \) (the other case is similar). Applying the induction hypothesis on \( \varphi_1 \), we have \( \sigma \cdot w \models \varphi_1 \) if \( \sigma \cdot w \models \varphi_1 \) and consequently \( \sigma \cdot w \models \varphi_1 \lor \varphi_2 \).
      - If \( P(\varphi_1, \sigma) \neq \top \) and \( P(\varphi_2, \sigma) \neq \top \), then we have \( P(\varphi_1, \sigma) = \neg P(\varphi_2, \sigma) \). Applying the induction hypothesis on \( \varphi_1 \) and \( \varphi_2 \), we obtain \( \sigma \cdot w \models \varphi_1 \iff \sigma \cdot w \not\models \varphi_2 \). Let us consider \( w \in \Sigma^\omega \). If \( \sigma \cdot w \models \varphi_1 \lor \varphi_2 \), then we have \( \sigma \cdot w \models \varphi_1 \lor \varphi_2 \). Else (\( \sigma \cdot w \models \varphi_1 \)), we have \( \sigma \models \varphi_2 \), and then \( \sigma \cdot w \models \varphi_1 \lor \varphi_2 \).
    - If \( P(\varphi_1 \lor \varphi_2, \sigma) \neq \top \), then we either have \( w \models P(\varphi_1, \sigma) \) or \( w \models P(\varphi_2, \sigma) \). Let us treat the case where \( w \models P(\varphi_1, \sigma) \) (the other case is similar). From \( w \models P(\varphi_1, \sigma) \), we can apply the induction hypothesis on \( \varphi_1 \) to obtain \( \sigma \cdot w \models \varphi_1 \), and thus \( \sigma \cdot w \models \varphi_1 \lor \varphi_2 \).
- Case \( \varphi = G \varphi' \): This case is similar to the previous one.
- Case \( \varphi = P \varphi' \): Recall that, according to the progression function for operator \( G \), \( P(G \varphi', \sigma) = P(\varphi', \sigma) \land G \varphi' \).
  - Let us suppose that \( \sigma \cdot w \models G \varphi' \). According to the LTL semantics of operator \( G \), we have \( \forall i \in \mathbb{N}^\omega, (\sigma \cdot w^i) \models \varphi' \). In particular, it implies that \( (\sigma \cdot w)^i \models \varphi' \) i.e., \( \sigma \cdot w \models \varphi' \), and \( \forall i \in \mathbb{N}^\omega, (\sigma \cdot w^i) \models \varphi' \). i.e., \( (\sigma \cdot w)^i \models \varphi' \), i.e., \( \sigma \cdot w \models \varphi' \). Using the induction hypothesis on \( \varphi' \), from \( \sigma \cdot w \models \varphi' \), we obtain \( w \models P(\varphi', \sigma) \). As expected, according to the LTL semantics of operator \( \land \), we have \( w \models P(\varphi', \sigma) \land G \varphi' = P(G \varphi', \sigma) \).
  - Let us suppose that \( w \models P(G \varphi', \sigma) \land G \varphi' \). It follows that \( w \models P(\varphi', \sigma) \), and thus, using the induction hypothesis on \( \varphi' \), \( \sigma \cdot w \models \varphi' \). Using the LTL semantics of operator \( G \), from \( \sigma \cdot w \models \varphi' \) and \( w \models G \varphi' \), we deduce \( \forall i \in \mathbb{N}^\omega, w^i \models \varphi' \), and then \( \forall i \in \mathbb{N}, (\sigma \cdot w)^i \models \varphi' \), i.e., \( \sigma \cdot w \models G \varphi' \).
- Case \( \varphi = X \varphi' \): This case is similar to the previous one.
- Case \( \varphi = X \varphi' \). On one hand, using the progression function for \( X \), we have \( P(X \varphi', \sigma) = \varphi' \). On the other hand, using the LTL semantics of operator \( X \), we have \( \sigma \cdot w \models X \varphi' \) iff \( w \models \varphi' \). Thus, we have \( \sigma \cdot w \models X \varphi' \) iff \( w \models \varphi' \) (induction hypothesis on \( \varphi' \)) \( w \models P(X \varphi', \sigma) \).
The proof uses the definition of the LTL semantics (Definition 1), the definition of good and bad prefixes (Definition 2), the progression function (Definition 3), and Lemma 1.

**Proof** According to Lemma 1, we have \( \forall \sigma \in \Sigma. \forall w \in \Sigma^\omega. \sigma \cdot w \models \varphi \iff w \models P(\varphi, \sigma) \). Consequently, we have \( \forall \sigma \in \Sigma. \forall w \in \Sigma^\omega. \sigma \cdot w \models \varphi \iff \forall \sigma \in \Sigma. \forall w \in \Sigma^\omega. w \models P(\varphi, \sigma) \). Similarly, when \( \varphi = \top \), we have \( \forall \sigma \in \Sigma. \forall w \in \Sigma^\omega. \sigma \cdot w \models \varphi \). Using, \( \sigma \cdot w \models \varphi \), i.e., \( \sigma \in \text{good}(\varphi) \). Similarly, when \( \varphi = \bot \), we have \( \forall \sigma \in \Sigma. \forall w \in \Sigma^\omega. \sigma \cdot w \not\models \varphi \). The proof can also be obtained in a more detailed manner as shown below. Let us consider \( \sigma \in \Sigma \) and \( \varphi \in \text{LTL} \). The proof is performed by a structural induction on \( \varphi \).

**Base Case:** \( \varphi \in \{ \top, \bot, p \in AP \} \).

- Case \( \varphi = \top \). In this case, the proof is trivial since \( P(\top, \sigma) = \top \) and, according to the LTL semantics of \( \top \) and the definition of good prefixes, \( \text{good}(\top) = \Sigma^\omega \).
- Case $\varphi = \bot$. Similarly, in this case, the proof is trivial since $P(\bot, \sigma) = \bot$ and bad(\bot) = $\Sigma^*$.
- Case $\varphi = p \in AP$.
  Let us suppose that $P(\varphi, \sigma) = \top$. According to the progression function, it means that $p \in \sigma$. Moreover, since $\varphi = p$, according to the LTL semantics of atomic propositions, for any $w \in \Sigma^*$, we have $\sigma \cdot w \models \varphi$. According to the definition of good prefixes, it means that $\sigma \in \text{good}(\varphi)$.
  The proof for $P(\varphi, \sigma) = \bot \iff \sigma \in \text{bad}(\varphi)$ is similar.

**Induction Case:** $\varphi \in \{\neg \varphi’, \varphi \lor \varphi_1 \land \varphi_2, \varphi \land \varphi_2, G\varphi', F\varphi', X\varphi', \varphi_1 U \varphi_2\}$. The induction hypothesis states that the lemma holds for some formulas $\varphi', \varphi_1, \varphi_2 \in \text{LTL}$.

- Case $\varphi = \neg \varphi'$. In this case, the result is obtained by using the induction hypothesis on $\varphi'$ and the equality $\bot = \neg \top$ and $\neg(\neg \varphi) = \varphi$.
- Case $\varphi = \varphi_1 \lor \varphi_2$. Recall that, according to the progression function for operator $\lor$, $P(\varphi_1 \lor \varphi_2, \sigma) = P(\varphi_1, \sigma) \lor P(\varphi_2, \sigma)$.
  Let us suppose that $P(\varphi, \sigma) = \top$. We distinguish two cases:
  - If $P(\varphi_1, \sigma) = \top$ or $P(\varphi_2, \sigma) = \top$. Let us treat the case where $P(\varphi_1, \sigma) = \top$. Using the induction hypothesis on $\varphi_1$, we have $\sigma \in \text{good}(\varphi_1)$. According to the definition of good prefixes, we have $\forall w \in \Sigma^*, \sigma \cdot w \models \varphi_1$. We easily deduce, using the LTL semantics of operator $\lor$, that $\forall w \in \Sigma^*, \sigma \cdot w \models \varphi_1 \lor \varphi_2$, that is, $\sigma \in \text{good}(\varphi_1 \lor \varphi_2)$.
  - If $P(\varphi_1, \sigma) \neq \top$ and $P(\varphi_2, \sigma) \neq \top$. Since $P(\varphi, \sigma) = \top$, we have $P(\varphi_1, \sigma) = \neg P(\varphi_2, \sigma)$. Using Lemma 1, we have $\forall w \in \Sigma^*, \sigma \cdot w \models \varphi_1 \iff w \models P(\varphi_1, \sigma)$ and $\forall w \in \Sigma^*, \sigma \cdot w \models \varphi_2 \iff w \models P(\varphi_2, \sigma)$. We deduce that $\forall w \in \Sigma^*, \sigma \cdot w \models \varphi_1 \iff \sigma \cdot w \not\models \varphi_2$.
    Let us consider $w \in \Sigma^*$. If $\sigma \cdot w \models \varphi_1$, we have $\sigma \cdot w \models \varphi_1 \lor \varphi_2$. Else (i.e., $\sigma \cdot w \not\models \varphi_1$), we have $\sigma \cdot w \models \varphi_2$, and then $\sigma \cdot w \models \varphi_1 \lor \varphi_2$. That is, $\forall w \in \Sigma^*, \sigma \cdot w \models \varphi_1 \lor \varphi_2$, i.e., $\sigma \in \text{good}(\varphi_1 \lor \varphi_2)$.

Let us suppose that $P(\varphi, \sigma) = \bot$. In this case, we have $P(\varphi_1, \sigma) = \bot$ and $P(\varphi_2, \sigma) = \bot$. Similarly, we can apply the induction hypothesis on $\varphi_1$ and $\varphi_2$ to find that $\sigma$ is bad prefix of both $\varphi_1$ and $\varphi_2$, and thus is a bad prefix of $\varphi_1 \lor \varphi_2$ (using the LTL semantics of operator $\lor$).

- Case $\varphi = \varphi_1 \land \varphi_2$. This case is symmetrical to the previous one.
- Case $\varphi = G\varphi'$. Recall that, according to the progression function for operator $G$, $P(G\varphi', \sigma) = P(\varphi', \sigma) \land G\varphi'$.
  Let us suppose that $P(\varphi, \sigma) = \top$. It means that $P(\varphi', \sigma) = \top$ and $G\varphi' = \top$. This case reduces to the case where $\varphi = \top$.
  Let us suppose that $P(\varphi, \sigma) = \bot$. We distinguish two cases.
  - If $P(\varphi', \sigma) = \bot$ or $G\varphi' = \bot$. We distinguish again two sub-cases.
    - Sub-case $P(\varphi', \sigma) = \bot$. Using the induction hypothesis on $\varphi'$, we deduce that $\sigma \in \text{bad}(\varphi')$, i.e., $\forall w \in \Sigma^*, \sigma \cdot w \not\models \varphi'$. Following the LTL semantics of operator $G$, we deduce that $\forall w \in \Sigma^*, \sigma \cdot w \not\models G\varphi'$, i.e., $\sigma \in \text{bad}(G\varphi')$.
    - Sub-case $G\varphi' = \bot$. This case reduces to the case where $\varphi = \bot$.
  - If $P(\varphi', \sigma) \neq \bot$ and $G\varphi' \neq \bot$. From $P(\varphi', \sigma) \land G\varphi' = \bot$, we deduce that $P(\varphi', \sigma) = \neg G\varphi'$. Using Lemma 1 on $\varphi'$, we have $\forall w \in \Sigma^*, \sigma \cdot w \not\models \varphi' \iff w \models P(\varphi', \sigma)$. Thus $\forall w \in \Sigma^*, \sigma \cdot w \not\models \varphi' \iff w \models G\varphi'$. Let us consider $w \in \Sigma^*$. If $\sigma \cdot w \models \varphi'$, then we have $w \not\models G\varphi'$. According to the LTL semantics of operator $G$, it means that $\exists i \in \mathbb{N}^+\cup \{0\}, w^i \not\models \varphi'$. Thus, still following the LTL semantics of operator $G$, $(\sigma \cdot w)^{i+1} \not\models \varphi'$, and, consequently $(\sigma \cdot w)^{i+1} \not\models G\varphi'$. Else $w \not\models G\varphi'$. We have $w \not\models G\varphi'$. According to the LTL semantics of operator $G$, it means that $\exists i \in \mathbb{N}^+\cup \{0\}, w^i \models \varphi'$, and thus $(\sigma \cdot w)^{i+1} \models \varphi'$. Consequently $w \models G\varphi'$.

- Case $\varphi = F\varphi'$. Recall that, according to the progression function for operator $F$, $P(F\varphi', \sigma) = P(\varphi', \sigma) \lor F\varphi'$.
  Let us suppose that $P(\varphi, \sigma) = \top$. We distinguish two cases.
  - If $P(\varphi', \sigma) = \top$ or $F\varphi' = \top$. Following the previous reasoning, using the induction hypothesis on $\varphi'$, the LTL semantics of operator $F$, and the definition of good prefixes, we obtain the expected result.
  - Sub-case $F\varphi' = \top$. This case reduces to the case where $\varphi = \top$.
  - If $P(\varphi', \sigma) \neq \top$ and $F\varphi' \neq \top$. From $P(\varphi', \sigma) \lor F\varphi' = \bot$, we deduce that $P(\varphi', \sigma) = \neg F\varphi'$. Using Lemma 1 on $\varphi'$, we have $\forall w \in \Sigma^*, \sigma \cdot w \models \varphi' \iff w \models P(\varphi', \sigma)$. We thus have $\forall w \in \Sigma^*, \sigma \cdot w \models \varphi' \iff w \not\models F\varphi'$. Let us consider $w \in \Sigma^*$. If $\sigma \cdot w \models \varphi'$, using the LTL semantics of operator $F$, we have directly $\sigma \cdot w \models F\varphi'$. Else $\sigma \cdot w \not\models F\varphi'$, we have $w \not\models F\varphi'$. According to the LTL semantics of operator $F$, it means that $\exists i \in \mathbb{N}^+\cup \{0\}, w^i \models \varphi'$, and thus $(\sigma \cdot w)^{i+1} \models \varphi'$. Consequently $\sigma \cdot w \models F\varphi'$. That is, $\sigma \in \text{good}(F\varphi')$.

Let us suppose that $P(\varphi', \sigma) = \bot$. It means that $P(\varphi', \sigma) = \bot$ and $F\varphi' = \bot$. A similar reasoning as the one used for the case $\varphi = G\varphi'$ and $P(\varphi, \sigma) = \top$ can be applied to obtain the expected result.

30
Case $\varphi = X\varphi'$. Recall that, according to the progression function for operator $X$, $P(X\varphi', \sigma) = \varphi'$.

Let us suppose that $P(\varphi, \sigma) = \top$. It means that $\varphi' = \top$. According to the LTL semantics of $\top$, we have $\forall w \in \Sigma^\omega, w \models \varphi'$. Then, $\forall w \in \Sigma^\omega, \sigma \cdot w \models X\varphi' = \varphi$. That is, $\sigma \in \text{good}(X\varphi')$.

Let us suppose that $P(\varphi, \sigma) = \bot$. It means that $\varphi' = \bot$. According to the LTL semantics of $\bot$, we have $\forall w \in \Sigma^\omega, w \not\models \varphi'$, which is $\forall w \in \Sigma^\omega, \sigma \cdot w \not\models X\varphi' = \varphi$. That is, $\sigma \in \text{bad}(X\varphi')$.

Case $\varphi = \phi_1 U \phi_2$. Recall that, according to the progression function for operator $U$, $P(\phi_1 U \phi_2, \sigma) = P(\phi_2, \sigma) \lor (P(\phi_1, \sigma) \land \phi_1 U \phi_2)$.

Let us suppose that $P(\varphi, \sigma) = \top$. We distinguish two cases.

- If $P(\phi_2, \sigma) \neq \top$ and $P(\phi_1, \sigma) \land \phi_1 U \phi_2 = \top$.
  - Sub-case $P(\phi_2, \sigma) = \top$. Using the induction hypothesis on $\phi_2$, we have $\sigma \in \text{good}(\phi_2)$.
    Let us consider $w \in \Sigma^\omega$, we have $\sigma \cdot w \models \phi_2$, i.e., $(\sigma \cdot w)^\omega \models \phi_1 U \phi_2$. According to the LTL semantics of $U$, we have $\sigma \cdot w \models \phi_1 \lor \phi_2$, i.e., $\sigma \cdot w \in L(\phi_1 U \phi_2)$. We deduce that $\sigma \in \text{good}(\phi_1 U \phi_2)$.
    - Sub-case $P(\phi_1, \sigma) \land \phi_1 U \phi_2 = \top$. Necessarily, $\phi_1 U \phi_2 = \top$ and this case reduces to the first one already treated.

- If $P(\phi_2, \sigma) = \top$ and $P(\phi_1, \sigma) \land \phi_1 U \phi_2 \neq \top$. From $P(\phi_1 U \phi_2, \sigma) = \top$, we deduce that $P(\phi_2, \sigma) = P(\phi_1, \sigma) \land \phi_1 U \phi_2)$. Applying Lemma 1 to $\phi_2$, we obtain $\forall w \in \Sigma^\omega, \sigma \cdot w \models \phi_2 \iff w \models P(\phi_2, \sigma)$. We thus have $\forall w \in \Sigma^\omega, \sigma \cdot w \models \phi_2 \iff w \not\models P(\phi_1, \sigma) \land \phi_1 U \phi_2$.
    Let us consider $w \in \Sigma^\omega$. Let us distinguish two cases. If $\sigma \cdot w \models \phi_2$, according to the LTL semantics of $U$, we have $\sigma \cdot w \models \phi_1 U \phi_2$. Else ($\sigma \cdot w \not\models \phi_2$), it implies that $\sigma \cdot w \not\models P(\phi_1, \sigma) \land \phi_1 U \phi_2$.

A.1.3 Some Intermediate Lemmas

The following lemma states some equality’s that directly follow from an inductive application of the definition of the progression function on events. (Proofs of Lemmas 3 and 4 are given p. 14 and 36, respectively).

**Lemma 5** Given some formulae $\varphi, \phi_1, \phi_2 \in \text{LTL}$, and a trace $u \in \Sigma^\omega$, the progression function can be extended to the trace $u$ by successively applying the previously defined progression function to each event of $u$. Moreover, we have: $\forall \varphi, \phi_1, \phi_2 \in \text{LTL}, \forall u \in \Sigma^\omega$.

\[
P(T, u) = \top,
\]
\[
P(\bot, u) = \bot
\]
\[
P(p \in AP, u) = \top \text{ if } p \in u, \bot \text{ otherwise}
\]
\[
P(\neg \varphi, u) = \neg P(\varphi, u)
\]
\[
P(\phi_1 \lor \phi_2, u) = P(\phi_1, u) \lor P(\phi_2, u)
\]
\[
P(\phi_1 \land \phi_2, u) = P(\phi_1, u) \land P(\phi_2, u)
\]
\[
P(\phi \cdot \sigma, u) = \bigwedge_{i=1}^{\|u\|} P(\phi, u_i) \land \sigma
\]
\[
P(\phi \cdot u^\omega, u) = \bigwedge_{i=1}^{\|u\|} P(\phi, u_i) \land u^\omega
\]
\[
P(\phi \cdot \sigma, u) = \bigwedge_{i=1}^{\|u\|} P(\phi, u_i) \land \sigma
\]
\[
P(\phi \cdot u^\omega, u) = \bigwedge_{i=1}^{\|u\|} P(\phi, u_i) \land u^\omega
\]
\[
P(\phi_1 U \phi_2, u) = \begin{cases} P(\phi_2, u) \lor P(\phi_1, u) \land \phi_1 U \phi_2 & \text{if } \|u\| = 1, \\ \bigvee_{i=1}^{\|u\|} (P(\phi_2, u_i) \land \bigwedge_{j=0}^{i-1} P(\phi_1, u_j)) \lor \bigwedge_{i=1}^{\|u\|} P(\phi_1, u_i) \land \phi_1 U \phi_2 & \text{otherwise.} \end{cases}
\]

**Proof** The proof is done by two inductions: an induction on the length of the trace $u$ (which is also the number of times the progression function is applied) and a structural induction on $\varphi \in \text{LTL}$.

**Base Case**: $u = \sigma \in \Sigma, \|u\| = 1$.

In this case, the result holds thanks to the definition of the progression function.

**Induction case**:

Let us suppose that the lemma holds for any trace $u \in \Sigma^\omega$ of some length $t \in \mathbb{N}$ and let us consider the trace $u \cdot \sigma \in \Sigma^\omega$, we perform a structural induction on $\varphi \in \text{LTL}$.

**Structural Base case**: $\varphi \in \{ T, \bot, p \in AP \}$.

- Case $\varphi = T$. In this case the result is trivially since we have:
  \[
P(T, u \cdot \sigma) = P(\{ T, u \}, \sigma) \text{ (extended progression)}
  \]
  \[
P(T, \sigma) \text{ (induction hypothesis on u)}
  \]
  \[
  = T \text{ (progression on events)}
  \]

- Case $\varphi = \bot$. This case is symmetrical to the previous one.

31
− Case \( \varphi = p \in AP \). Let us distinguish two cases: \( p \in u(0) \) or \( p \notin u(0) \).

− If \( p \in u(0) \), we have:

\[
P(p, u \cdot \sigma) = P(P(p, u), \sigma) \quad \text{(extended progression)}
\]
\[
P(\top, \sigma) = \top \quad \text{(progression on events)}
\]

− If \( p \notin u(0) \), we have:

\[
P(p, u \cdot \sigma) = P(P(p, u), \sigma) \quad \text{(extended progression)}
\]
\[
P(\bot, \sigma) = \bot \quad \text{(progression on events)}
\]

\[\text{Induction Case: } \varphi \in \{ \neg \varphi', \varphi_1 \lor \varphi_2, \varphi_1 \land \varphi_2, G \varphi', F \varphi', X \varphi', \varphi_1 U \varphi_2 \}. \text{ Our induction hypothesis states that the lemma holds for some formulae } \varphi', \varphi_1, \varphi_2 \in \text{LTL.} \]

− Case \( \varphi = \neg \varphi' \). We have:

\[
P(\neg \varphi', u \cdot \sigma) = P(P(\neg \varphi', u), \sigma) \quad \text{(extended progression)}
\]
\[
= P(P(\neg \varphi', u), \sigma) \quad \text{(induction hypothesis on } u \text{ and } \varphi')
\]
\[
= \neg P(P(\varphi', u), \sigma) \quad \text{(progression on events)}
\]
\[
= \neg P(\varphi', u \cdot \sigma) \quad \text{(extended progression)}
\]

− Case \( \varphi = X \varphi' \). We have:

\[
P(X \varphi', u \cdot \sigma) = P(P(X \varphi', u), \sigma) \quad \text{(extended progression)}
\]
\[
= P(P(\varphi', u^1), \sigma) \quad \text{(induction hypothesis on } u \text{ and } \varphi')
\]
\[
= P(\varphi', (u \cdot \sigma)^1) \quad \text{(extended progression)}
\]

− Case \( \varphi = \varphi_1 \lor \varphi_2 \). We have:

\[
P(\varphi_1 \lor \varphi_2, u \cdot \sigma) = P(P(\varphi_1 \lor \varphi_2, u), \sigma) \quad \text{(extended progression)}
\]
\[
= P(P(\varphi_1, u) \lor P(\varphi_2, u), \sigma) \quad \text{(induction hypothesis on } u \text{ and } \varphi_1, \varphi_2)
\]
\[
= P(P(\varphi_1, u) \lor P(P(\varphi_2, u), \sigma) \quad \text{(progression on events)}
\]
\[
= P(\varphi_1, u \cdot \sigma) \lor P(\varphi_2, u \cdot \sigma) \quad \text{(extended progression)}
\]

− Case \( \varphi = \varphi_1 \land \varphi_2 \). This case is similar to the previous one.

− Case \( \varphi = G \varphi' \). We have:

\[
P(G \varphi', u \cdot \sigma)
\]
\[
= P(P(G \varphi', u), \sigma) \quad \text{(extended progression)}
\]
\[
= P\left(\bigwedge_{t=0}^{u-1} P(\varphi', u^t) \land G \varphi', \sigma\right) \quad \text{(induction hypothesis on } u \text{ and } \varphi')
\]
\[
= P\left(\bigwedge_{t=0}^{u-1} P(\varphi', u^t), \sigma\right) \land P(G \varphi', \sigma) \quad \text{(progression on events for } \land\text{)}
\]
\[
= P\left(\bigwedge_{t=0}^{u-1} P(\varphi', u^t), \sigma\right) \land P(G \varphi', \sigma) \quad \text{(extended progression for } \land\text{)}
\]
\[
= P\left(\bigwedge_{t=0}^{u-1} P(\varphi', u^t \cdot \sigma) \land P(G \varphi', \sigma) \quad \text{(extended progression)}
\]
\[
= P\left(\bigwedge_{t=0}^{u-1} P(\varphi', u^t \cdot \sigma) \land P(\varphi', \sigma) \land G \varphi' \quad \text{(progression on events for } G\text{)}
\]
\[
= P\left(\bigwedge_{t=0}^{u-1} P(\varphi', (u \cdot \sigma)^t) \land P(\varphi', (u \cdot \sigma)^{u-\sigma-1}) \land G \varphi' \quad \text{(induction hypothesis on } u \text{ and } \varphi', \sigma = (u \cdot \sigma)^{u-\sigma-1})
\]
\[
= P\left(\bigwedge_{t=0}^{u-1} P(\varphi', (u \cdot \sigma)^t) \land G \varphi' \quad \text{(progression for } \land\text{)}
\]

− Case \( \varphi = F \varphi' \). We have:

\[
P(F \varphi', u \cdot \sigma)
\]
\[
= P(P(F \varphi', u), \sigma) \quad \text{(extended progression)}
\]
\[
= P\left(\bigvee_{t=0}^{u-1} P(\varphi', u^t) \lor F \varphi', \sigma\right) \quad \text{(induction hypothesis on } u \text{ and } \varphi')
\]
\[
= P\left(\bigvee_{t=0}^{u-1} P(\varphi', u^t), \sigma\right) \lor P(F \varphi', \sigma) \quad \text{(progression on events)}
\]
\[
= \bigvee_{t=0}^{u-1} P(\varphi', u^t \cdot \sigma) \lor P(\varphi', \sigma) \lor F \varphi' \quad \text{(extended progression for } \lor\text{)}
\]
\[
= \bigvee_{t=0}^{u-1} P(\varphi', (u \cdot \sigma)^t) \lor P(\varphi', (u \cdot \sigma)^{u-\sigma-1}) \lor F \varphi' \quad \text{(progression on events for } F\text{)}
\]
\[
= \bigvee_{t=0}^{u-1} P(\varphi', (u \cdot \sigma)^t) \lor F \varphi' \quad \text{(induction hypothesis on } u \text{ and } \varphi', \sigma = (u \cdot \sigma)^{u-\sigma-1})
\]
\[
= \bigvee_{t=0}^{u-1} P(\varphi', (u \cdot \sigma)^t) \lor F \varphi' \quad \text{(progression for } \lor\text{)}
\]

− Case \( \varphi = \varphi_1 U \varphi_2 \). We have:

32
Moreover:

\[ P(\varphi_1 U \varphi_2, u \cdot \sigma) \]

(extended progression)

\[ = P(P(\varphi_1 U \varphi_2), u), \sigma) \]

(induction hypothesis on \( u \) and structural induction hypothesis on \( \varphi_1 \) and \( \varphi_2 \))

\[ = P \left( \bigvee_{i=0}^{u-1} \left( P(\varphi_2, u^i) \wedge \bigwedge_{j=0}^{u-1} P(\varphi_1, u^j) \right) \right) \wedge \bigwedge_{j=0}^{u-1} P(\varphi_1, u^j) \wedge \varphi_1 U \varphi_2, \sigma) \]

(presentation on events for \( \forall \))

\[ = P \left( \bigvee_{i=0}^{u-1} \left( P(\varphi_2, u^i) \wedge \bigwedge_{j=0}^{u-1} P(\varphi_1, u^j) \right) \right) \wedge \bigwedge_{j=0}^{u-1} P(\varphi(\varphi_1, u^j), \sigma) \wedge P(\varphi_1 U \varphi_2, \sigma) \]

(presentation on events for \( \forall \) and \( \lor \))

\[ = \bigvee_{i=0}^{u-1} \left( P(\varphi_2, u^i \cdot \sigma) \wedge \bigwedge_{j=0}^{u-1} P(\varphi_1, u^j \cdot \sigma) \right) \wedge \bigwedge_{j=0}^{u-1} P(\varphi_1, u^j \cdot \sigma) \wedge \varphi_1 U \varphi_2, \sigma) \]

(extended progression)

Moreover:

\[ \bigwedge_{i=0}^{u-1} P(\varphi_1, u^i \cdot \sigma) \wedge P(\varphi_1 U \varphi_2, \sigma) \]

(presentation on events for \( U \))

\[ = \bigwedge_{i=0}^{u-1} P(\varphi_1, u^i \cdot \sigma) \wedge \bigwedge_{i=0}^{u-1} P(\varphi_2, \sigma) \wedge P(\varphi_1 U \varphi_2) \]

(distribution of \( \land \) over \( \lor \))

\[ = \left( \bigwedge_{i=0}^{u-1} P(\varphi_1, u^i \cdot \sigma) \wedge P(\varphi_2, \sigma) \right) \lor \left( \bigwedge_{i=0}^{u-1} P(\varphi_1, u^i \cdot \sigma) \wedge \varphi_1 U \varphi_2 \right) \]

\[ (\sigma = \varphi_1 U \varphi_2) \]

\[ = \left( \bigwedge_{i=0}^{u-1} P(\varphi_1, u^i \cdot \sigma) \wedge P(\varphi_2, \sigma) \right) \lor \left( \bigwedge_{i=0}^{u-1} P(\varphi_1, u^i \cdot \sigma) \wedge \varphi_1 U \varphi_2 \right) \]

\[ (\sigma = \varphi_1 U \varphi_2) \]

\[ \\Box \]

We introduce another intermediate lemma, which is a consequence of the definition of LTL semantics (Definition 1) and the definition of the progression function (Definition 5). This lemma will be useful in the remaining proofs. This lemma states that the progression function “mimics” the semantics of LTL on a trace \( u \in \Sigma^\omega \).

**Lemma 6** Let \( \varphi \) be an LTL formula, \( u \in \Sigma^* \) a finite trace and \( w \in \Sigma^\omega \) an infinite trace, we have \( u \cdot w \models \varphi \iff w \models P(\varphi, u) \).

**Proof** We shall prove the following statement:

\[ \forall u \in \Sigma^*, \forall w \in \Sigma^\omega \forall \varphi \in \text{LTL}, u \cdot w \models \varphi \iff w \models P(\varphi, u). \]

Let us consider \( u \in \Sigma^* \), the proof is done by a structural induction on \( \varphi \in \text{LTL} \) (when \( u = \epsilon \), the lemma holds vacuously).

**Base case**: \( \varphi \in \{ \top, \bot, p \in AP \}. \)
- Case $\varphi = \top$. This case is trivial since, using Lemma 5 on $\top$ and $u$, we have $P(\top, u) = \top$. Moreover, according to the LTL semantics of $\top$, $\forall w \in \Sigma^\omega$. $u \cdot w \models \top$.
- Case $\varphi = \bot$. This case is symmetrical to the previous one.
- Case $\varphi = p \in AP$. Let us suppose that $u \cdot w \models p$. By applying Lemma 5 on $\top$ and $u$, we have $P(u, p) = \top$. Moreover, due to the LTL semantics of $\top$, we have $\forall w \in \Sigma^\omega$. $w \models \top = P(u, p)$.
- Let us suppose that $w \models P(p, u)$. Since $P(p, u) \in \{\top, \bot\}$, we have necessarily $P(p, u) = \top$. According to the progression function, $P(p, u) = \top$ necessitates that $p \in u(0)$. Using the LTL semantics of atomic propositions, we deduce that $(u \cdot w)^k \models p$, i.e., $u \cdot w \models p$.

**Induction Case:** $\varphi \in \{\neg \varphi', \varphi_1 \lor \varphi_2, \varphi_1 \land \varphi_2, G \varphi', F \varphi', X \varphi', \varphi_1 U \varphi_2\}$. Our induction hypothesis states that the lemma holds for some formulae $\varphi', \varphi_1, \varphi_2 \in \text{LTL}$.

- Case $\varphi = \varphi_1 \lor \varphi_2$. Recall that, by applying Lemma 5 on $\varphi_1 \lor \varphi_2$ and $u$, we have $P(\varphi_1 \lor \varphi_2, u) = P(\varphi_1, u) \lor P(\varphi_2, u)$.
- Let us suppose that $u \cdot w \models \varphi_1 \lor \varphi_2$. Let us distinguish two cases: $\varphi_1 \lor \varphi_2 = \top$ and $\varphi_1 \lor \varphi_2 \neq \top$. If $\varphi_1 \lor \varphi_2 = \top$, then this case reduces to the case where $\varphi = \top$ already treated. If $\varphi_1 \lor \varphi_2 \neq \top$, it means that either $u \cdot w \models \varphi_1$ or $u \cdot w \models \varphi_2$. Let us treat the case where $u \cdot w \models \varphi_1$ (the other case is similar). From $u \cdot w \models \varphi_1$, we can apply the structural induction hypothesis on $\varphi_1$ to obtain $w \models P(\varphi_1, u)$, and then, $w \models P(\varphi_1, u) \lor P(\varphi_2, u) = P(\varphi_1 \lor \varphi_2, u)$.
- Let us suppose that $w \models P(\varphi_1 \lor \varphi_2, u)$. Let us again distinguish two cases. If $P(\varphi_1, u) \lor P(\varphi_2, u) = \top$, then it reduces to the case where $\varphi = \top$ already treated. If $P(\varphi_1, u) \lor P(\varphi_2, u) \neq \top$, then we have either $w \models P(\varphi_1, u)$ or $w \models P(\varphi_2, u)$. Let us treat the case where $w \models P(\varphi_1, u)$ (the other case is similar). From $w \models P(\varphi_1, u)$, we can apply the structural induction hypothesis on $\varphi_1$ to obtain $u \cdot w \models \varphi_1$, and thus, using the LTL semantics of $\top$, we have $u \cdot w \models \top$. Furthermore, we have $\varphi_1 \lor \varphi_2 = \top$.
- Case $\varphi = G \varphi'$. Recall that, by applying Lemma 5 on $G \varphi'$ and $u$, we have $P(G \varphi', u) = \bigwedge_{i=0}^{n-1} P(\varphi', u^i) \land G \varphi'$.
- Let us suppose that $u \cdot w \models G \varphi'$. From the LTL semantics of operator $G$, we have $\forall i \in N \geq 0$, $(u \cdot w)^i \models \varphi'$. In particular, it implies that $\forall 0 \leq i \leq |u| - 1$. $u^i \cdot w \models \varphi'$ and $\forall i \geq 0$, $(u \cdot w)^{i+1} \models \varphi'$. Using, $\forall 0 \leq i \leq |u| - 1$. $u^i \cdot w \models \varphi'$ and applying the structural induction hypothesis on $\varphi'$ and the $u^i$'s, we obtain $\forall 0 \leq i \leq |u| - 1$. $u^i \cdot w \models P(\varphi', u^i)$, and thus $w \models \bigwedge_{i=0}^{n-1} P(\varphi', u^i)$. Using $\forall i \geq 0$, $u^i = (u \cdot w)^i \models \varphi'$, we obtain $w \models G \varphi'$. As expected, according to the LTL semantics of $\land$, we have $w \models \bigwedge_{i=0}^{n-1} P(\varphi', u^i) \land G \varphi' = P(G \varphi', u)$.
- Let us suppose that $w \models P(G \varphi', u)$. We have $\forall 0 \leq i \leq |u| - 1$. $u^i \cdot w \models \varphi'$ and $w \models G \varphi'$. Using the structural induction hypothesis on $\varphi'$ and the $u^i$'s, it follows that $\forall 0 \leq i \leq |u| - 1$. $u^i \cdot w \models \varphi'$. The semantics of operator $G$, from $w \models G \varphi'$ and $\forall 0 \leq i \leq |u| - 1$. $u^i \cdot w \models \varphi'$, we deduce $u \cdot w \models G \varphi'$.
- Case $\varphi = X \varphi'$. This case is similar to the previous one.
- Case $\varphi = X \varphi_1 \lor \varphi_2$. Recall that, by applying Lemma 5 on $\varphi_1 \lor \varphi_2$ and $u$, we have $P(X \varphi_1 \lor \varphi_2, u) = P(\varphi_1 \lor \varphi_2, u^0 \cdot \sigma)$.
- Using the LTL semantics of $X$, we have $u \cdot w \models X \varphi_1 \lor \varphi_2$ iff $u \cdot w \models \varphi_1 \lor \varphi_2$. Thus we have $u \cdot w \models X \varphi_1 \lor \varphi_2$ iff $u^0 \cdot w \models \varphi_1 \lor \varphi_2$ (induction hypothesis on $\varphi_1 \lor \varphi_2$) $w \models P(\varphi_1 \lor \varphi_2, u^0 \cdot \sigma) = P(X \varphi_1 \lor \varphi_2, u)$.
- Case $\varphi = \neg \varphi'$. Recall that, by applying Lemma 5 on $\neg \varphi'$ and $\neg \varphi'$, we have $P(\neg \varphi', u) = \neg P(\varphi', u)$.
- Using the LTL semantics of $\neg$, we have $\forall \varphi \in \text{LTL}, \forall w \in \Sigma^\omega$. $\varphi \iff w \not\models \neg w$. Thus, we have $u \cdot w \models \neg \varphi'$ iff $u \cdot w \not\models \varphi'$ iff (induction hypothesis on $\neg \varphi'$) $w \not\models P(\varphi', u)$.
- Case $\varphi = \varphi_1 U \varphi_2$. Recall that, by applying Lemma 5 on $\varphi_1$ and $\varphi_2$ and $u$, we have:
  $$P(\varphi_1 U \varphi_2, u) = \bigwedge_{i=0}^{n-1} (P(\varphi_2, u^i) \land \bigwedge_{j=0}^{i-1} P(\varphi_1, u^j)) \lor \bigwedge_{i=0}^{n-1} P(\varphi_1, u^i) \land \varphi_1 U \varphi_2$$.
- Let us suppose that $u \cdot w \models \varphi_1 U \varphi_2$. According to the LTL semantics of operator $U$, $\exists k \in N \geq 0$. $(u \cdot w)^k \models \varphi_2$ and $\forall 0 \leq i \leq k - 1$. $(u \cdot w)^i \models \varphi_1$. Let us distinguish two cases: $k > |u|$ and $k \leq |u|$.
  - If $k > |u|$, then we have in particular $\forall 0 \leq i \leq |u| - 1$. $u^i \cdot w \models \varphi_1$. Applying the structural induction hypothesis on $\varphi_1$ and the $u^i$'s, we find $\forall 0 \leq i \leq |u| \cdot w \models P(\varphi_1, u^i)$, i.e., $w \models \bigwedge_{i=0}^{n-1} P(\varphi_1, u^i)$. From $(\sigma \cdot w)^k \models \varphi_2$ and $k > |u| - 1$, we deduce that $\exists k \geq 0. w^k \models \varphi_2$ and $k = k - |u| + 1$. Furthermore, we have $\forall 0 \leq i \leq k$. $(u \cdot w)^i \models \varphi_2$. Finally, we have $w \models P(\varphi_1 U \varphi_2, u)$.
we have \( \forall \) \( \propto \quad \forall \). Proof

The proof uses the definition of \( A.1.4 \) Proof for Theorem 1 (p. 7)

\[
\forall \phi, u \in \Sigma \quad \forall \in \Sigma^+, u \cdot w \models \phi \iff w \models P(\phi, u). \]

Consequently, we have \( \forall u \in \Sigma^+, \forall w \in \Sigma^+, u \cdot w \models \phi \iff \forall u \in \Sigma^+, \forall w \in \Sigma^+, w \models P(\phi, u) \)

and \( \forall u \in \Sigma^+, \forall w \in \Sigma^+, u \cdot w \not\models \phi \iff \forall u \in \Sigma^+, \forall w \in \Sigma^+, w \not\models P(\phi, u) \).

Consequently, when \( P(\phi, u) = \top \), we have \( \forall u \in \Sigma^+, \forall w \in \Sigma^+, u \cdot w \models \phi \), i.e., \( u \in \text{good}(\phi) \). Also, when \( P(\phi, u) = \bot \), we have \( \forall u \in \Sigma^+, \forall w \in \Sigma^+, u \cdot w \not\models \phi \), i.e., \( u \in \text{bad}(\phi) \).

A.1.4 Proof for Theorem 1 (p. 7)

We shall prove the following statement:

\[
\forall u \in \Sigma^+, \forall w \in \Sigma^+, (u \cdot w)^k \models \varphi_2 \quad \Rightarrow \quad u \models \varphi_2 \quad \text{or} \quad (u \cdot w)^k \not\models \varphi_2 \quad \text{or} \quad (u \cdot w)^k \models \varphi_2.
\]

The proof uses the definition of LTL semantics (Definition 1), the definition of good and bad prefixes (Definition 5), the progression function (Definition 3), and Lemma 6.
A.2 Proofs for Section 6

A.2.1 Proof of Lemma 4 (p. 15)

Consider \(M = \{M_1, \ldots, M_n\}\) where each monitor \(M_i\) has a set of local atomic propositions \(AP_i = \Pi_i(\gL)\) and observes the set of events \(\Sigma_i\), we shall prove that:

\[
\forall i \in [1, n], \forall \sigma \in \Sigma_i, \forall \varphi \in \gL, \exists \sigma' \in \gL^{i+1} \text{ such that } \varphi \models \sigma' \Rightarrow \sigma' \models \varphi
\]

A.2.2 Proof of Theorem 2 (p. 12)

Consider \(\forall i \in [1, n], \forall \sigma \in \Sigma_i, \forall \varphi \in \gL, \exists \sigma' \in \gL^{i+1} \text{ such that } \varphi \models \sigma' \Rightarrow \sigma' \models \varphi\) and the set of events \(\Sigma\), we shall prove that:

\[
\forall i \in [1, n], \forall \sigma \in \Sigma_i, \forall \varphi \in \gL, \exists \sigma' \in \gL^{i+1} \text{ such that } \varphi \models \sigma' \Rightarrow \sigma' \models \varphi
\]

Let us consider \(\sigma \in \Sigma, \Sigma_i \subseteq \Sigma\). The proof is done by structural induction on \(\varphi \in \gL\).

**Base Case:** \(\varphi \in \{\top, \bot, p\} \text{ for some } p \in AP\).

- Case \(\varphi = \top\). In this case, the proof is trivial since \(P(\top, \sigma, AP_i) = \top\) and \(\sus(\top) = \emptyset\).
- Case \(\varphi = \bot\). This case is similar to the previous one.

**Induction Case:** \(\varphi \in \{\neg \varphi, \varphi_1 \lor \varphi_2, \varphi_1 \land \varphi_2, X \varphi, F\varphi, G\varphi, X\varphi, \varphi_1 U \varphi_2\}\). Our induction hypothesis states that the result holds for some formulae \(\varphi', \varphi_1, \varphi_2 \in \gL\).

- Case \(\varphi = \neg \varphi'\). On the one hand, we have

\[
\sus(P(\neg \varphi', \sigma, AP_i)) = \sus(\neg P(\varphi', \sigma, AP_i)) = \sus(\neg P(\varphi', \sigma, AP_i))
\]

On the other hand, we have \(\sus(\neg \varphi') = \sus(\varphi')\). Thus, by applying directly the induction hypothesis on \(\varphi'\), we obtain the expected result.

- Case \(\varphi = \varphi_1 \lor \varphi_2\). On the one hand, we have

\[
\sus(P(\varphi_1 \lor \varphi_2, \sigma, AP_i)) = \sus(P(\varphi_1, \sigma, AP_i) \lor P(\varphi_2, \sigma, AP_i)) = \sus(P(\varphi_1, \sigma, AP_i) \cup \sus(P(\varphi_2, \sigma, \Sigma_i)).
\]

Thus, \(X\varphi \in \sus(P(\varphi_1 \land \varphi_2, \sigma, AP_i)) \text{ implies that } X\varphi \in \sus(P(\varphi_1, \sigma, AP_i)) \text{ or } X\varphi \in \sus(P(\varphi_2, \sigma, AP_i))\). On the other hand, \(\sus(\varphi_1 \land \varphi_2) = \sus(\varphi_1) \cup \sus(\varphi_2)\). Hence, the result can be obtained by applying the induction hypothesis on either \(\varphi_1\) or \(\varphi_2\) depending on whether \(X\varphi \in \sus(P(\varphi_1, \sigma, AP_i))\) or \(X\varphi \in \sus(P(\varphi_2, \sigma, AP_i))\).

- Case \(\varphi = X\varphi'\) for some \(d' \in \mathbb{N}\) and \(p' \in AP\). One one hand, if \(p' \in AP_i\), then it implies that \(P(X\varphi, p', \sigma, AP_i) \in \{\top, \bot\}\). Else \((p' \notin AP_i)\), we have \(P(X\varphi, p', \sigma, AP_i) = X\varphi\). On the other hand, we have \(\sus(X\varphi) = \{\varphi\}\).

- Case \(\varphi = G\varphi'\). By definition of the progression rule for \(G\) and the definition of \(\sus\), we have

\[
\sus(P(G\varphi', \sigma, AP_i)) = \sus(P(\varphi', \sigma, AP_i) \land \varphi') = \sus(P(\varphi', \sigma, AP_i))
\]

Since \(\varphi'\) is behind a future temporal operator, the only case where \(\sus(P(\varphi', \sigma, AP_i)) \neq \emptyset\) is when \(\varphi'\) is a state-formula. In that case, we have \(X\varphi \in \sus(P(\varphi', \sigma, AP_i))\) implies that \(d = 1\).

- Cases \(\varphi \in \{F\varphi', X\varphi', \varphi_1 U \varphi_2\}\). These cases are similar to the previous one. \(\square\)

A.2.2 Proof of Theorem 2 (p. 12)

We have to prove that for any \(X^m\varphi \in \gL\), a local obligation of some monitor \(M_i \in M, m \leq \min(|M|, t + 1)\) at any time \(t \in \mathbb{N}^{\geq 0}\). We will suppose that there are at least two components in the system (otherwise, the proof is trivial), i.e., \(|M| \geq 2\). The proof is done by distinguishing three cases according to the value of \(t \in \mathbb{N}^{\geq 0}\).

**First case:** \(t = 0\). In this case, we shall prove that \(m \leq 1\). The proof is done by a structural induction on \(\varphi \in \gL\). Recall that for this case, where \(t = 0\), we have \(\forall i \in [1, |M|], \phi(i, 0, \varphi) = P(\varphi, \psi_i(0), AP_i)\).
**Base case:** \( \varphi \in \{ \top, \bot, p \in AP \} \).

- Case \( \varphi = \top \). In this case we have \( \forall t \in \mathbb{N}, \lo(i, 0, \top) = \top \). Moreover, \( \sus(T) = \top \).
- Case \( \varphi = \bot \). This case is symmetrical to the previous one.
- Case \( \varphi = p \in AP \). We distinguish two cases: \( p \in AP \) and \( p \notin AP \). If \( p \in AP \), then \( \lo(i, 0, p) = \bot \) else \( p \notin AP \), we have \( \lo(i, 0, p) = \top \).

**Structural Induction Case:** \( \varphi \in \{ \neg \varphi', \varphi_1 \lor \varphi_2, \varphi_1 \land \varphi_2, G\varphi', F\varphi', X\varphi', \varphi_1 U \varphi_2 \} \). Our induction hypothesis states that the result holds for some formulae \( \varphi', \varphi_1, \varphi_2 \in \text{LTL} \).

- Case \( \varphi = \varphi_1 \lor \varphi_2 \). We have:
  \[
  \begin{align*}
  \lo(i, 0, \varphi_1 \lor \varphi_2) &= \lo(i, 0, \varphi_1) \lor \lo(i, 0, \varphi_2) \\
  &= P(\varphi_1 \lor \varphi_2, u_i(0), AP_i) \quad \text{(lo definition for } t = 0) \\
  &= P(\varphi_1, u_i(0), AP_i) \lor P(\varphi_2, u_i(0), AP_i) \quad \text{(progression on events)} \\
  &= \lo(i, 0, \varphi_1) \lor \lo(i, 0, \varphi_2) \quad \text{(lo definition for } t = 0) \\
  \sus(\lo(i, 0, \varphi_1) \lor \lo(i, 0, \varphi_2)) &= \sus(\lo(i, 0, \varphi_1)) \lor \sus(\lo(i, 0, \varphi_2)) \quad \text{(sus definition)}
  \end{align*}
  \]

We can apply the induction hypothesis on \( \varphi_1 \) and \( \varphi_2 \) to obtain successively:

\[
\forall t \geq N \geq 0, \varphi \in \text{LTL}, \forall p \in \sus(\lo(i, t, \varphi_1)), m \leq 1
\]

\[
\forall t \geq N \geq 0, \varphi \in \text{LTL}, \forall p \in \sus(\lo(i, t, \varphi_2)), m \leq 1
\]

- Case \( \varphi = \neg \varphi' \). We have:
  \[
  \begin{align*}
  \lo(i, 0, \neg \varphi') &= \neg P(\varphi', u_i(0), AP_i) \quad \text{(lo definition)} \\
  &= \neg P(\varphi', u_i(0), AP_i) \quad \text{(progression on events)} \\
  \sus(\lo(i, 0, \neg \varphi')) &= \sus(\neg P(\varphi', u_i(0), AP_i)) \quad \text{(sus definition)} \\
  &= \sus(\lo(i, 0, \varphi')) \quad \text{(sus definition)}
  \end{align*}
  \]

- Case \( \varphi = X \varphi' \). We have:
  \[
  \begin{align*}
  \lo(i, 0, X \varphi') &= P(X \varphi', u_i(0), AP_i) \quad \text{(lo definition)} \\
  &= \varphi' \quad \text{(progression on events)} \\
  \sus(\lo(i, 0, X \varphi')) &= \sus(\varphi')
  \end{align*}
  \]

Since \( \varphi' \) is behind a future temporal operator, we have \( \sus(\varphi') = \emptyset \).

- Case \( \varphi = G \varphi' \). We have:
  \[
  \begin{align*}
  \lo(i, 0, G \varphi') &= P(G \varphi', u_i(0), AP_i) \quad \text{(lo definition)} \\
  &= P(\varphi', u_i(0), AP_i) \land G \varphi' \quad \text{(progression on events)} \\
  \sus(\lo(i, 0, G \varphi')) &= \sus(\lo(i, 0, \varphi') \land G \varphi') \quad \text{(sus definition)} \\
  &= \sus(\lo(i, 0, \varphi')) \quad \text{(sus definition)} \\
  &= \sus(G \varphi') \quad \text{(sus(\varphi') = } \emptyset)
  \end{align*}
  \]

- Case \( \varphi = F \varphi' \). This case is similar to the previous one.
- Case \( \varphi = \varphi_1 U \varphi_2 \). We have:
  \[
  \begin{align*}
  \lo(i, 0, \varphi_1 U \varphi_2) &= P(\varphi_1 U \varphi_2, u_i(0), AP_i) \quad \text{(lo definition)} \\
  &= P(\varphi_1 U \varphi_2, u_i(0), AP_i) \quad \text{(progression on events)} \\
  &= P(\varphi_2, u_i(0), AP_i) \lor (P(\varphi_1, u_i(0), AP_i) \land \varphi_1 U \varphi_2) \quad \text{(lo definition for } \varphi_1 \text{ and } \varphi_2) \\
  &= \lo(i, 0, \varphi_2) \lor \lo(i, 0, \varphi_1) \land \varphi_1 U \varphi_2
  \end{align*}
  \]

37
Moreover, from the same reasoning and using Lemma 4, we can find a set of indexes $t$.

For $t \geq 1$, the proof is done by reductio ad absurdum. Let us consider some $t \in \mathbb{N}$ and suppose that the theorem does not hold at time $t$. It means that:

$$\exists \varphi \in \text{LTL}. \exists i \in [1, |M|]. \exists \mathbf{x}^{d'} \in \text{ulo}(i, t, \varphi). d > \min(|M|, t + 1).$$

According to Lemma 3, since $\text{ulo}(i, t, \varphi) = \bigcup_{j=1}^{d} \text{Sus}(\text{Prop}(\text{ulo}(i, t, \varphi)))$, it means that $\exists j_1 \in [1, |M|] \setminus \{i\}. \mathbf{x}^{d-1} \in \text{Sus}(\text{Prop}(\text{ulo}(j_1, t-1, \varphi)))$. Using Lemma 4, we have $\mathbf{x}^{d-1} \in \text{Sus}(\text{Prop}(\text{ulo}(j_1, t-1, \varphi)))$. We deduce that $i = \min \{j \in [1, |M|] | \exists \mathbf{x} \in \text{ulo}(i, t, \varphi) \wedge p \in \text{AP}_{j-1} \}$. Moreover, from $\mathbf{x}^{d} \in \text{ulo}(i, t, \varphi)$, we find $p \notin \text{AP}_{i'}$, with $i < i'$. We can apply the same reasoning on $\mathbf{x}^{d'} \in \text{ulo}(i, t, \varphi)$ to find that $i < j_1 < i'$. Hence, we have $p \notin \bigcup_{j=1}^{d} \text{AP}_{j} \geq \text{AP}$. This is impossible.

A.2.3 Intermediate Lemmas and Notation

Let us first define some notations. Consider $\varphi \in \text{LTL}, u \in \Sigma^{+}, i \in [1, |M|]$: 

1. $\text{rp}(\varphi, u)$ is the formula $\varphi$ where past sub-formulae are removed and replaced by their evaluations using trace $u$. Formally:

\[
\text{rp}(\varphi, u) = \text{match } \varphi \text{ with } \\
| \mathbf{x}^{d} \mathbf{p} | \Rightarrow \begin{cases} 1 & \text{if } p \in \text{Prop}(u) \wedge d > 0 \\ \bot & \text{otherwise} \end{cases}
\]

2. $\text{rp}(\varphi, u, i)$ is the formula $\varphi$ where past sub-formulae are replaced by their evaluations using only the sub-trace $u_i$ of $u$.

\[
\text{rp}(\varphi, u, i) = \text{match } \varphi \text{ with } \\
| \mathbf{x}^{d} \mathbf{p} | \Rightarrow \begin{cases} 1 & \text{if } p \in \text{Prop}(u) \wedge d > 0 \\ \bot & \text{if } p \in \text{Prop}(u) \wedge d > 0 \text{ and } p \notin \text{AP}_{i} \end{cases}
\]

where $\varphi \in \Sigma^{*}$.
The following lemma exhibits some straightforward properties of function $rp$.

**Lemma 7** Let $\varphi$ be an LTL formula, $u \in \Sigma^+$ be a trace of length $t + 1$, $i \in \{1, |M|\}$ a monitor of one of the components, $u_i(t)$ the last event of $u$ on component $i$, we have:

1. $rp(P(\varphi, \sigma_i, AP_i), u) = rp(rp(\varphi, u(0) \ldots u(t - 1)), \sigma_i, AP_i), u)$;
2. $rp(P(\varphi, \sigma_i, AP_i), u) = P(\varphi, u(t), AP_i)$;
3. $P(\varphi, u_i(t), AP_i) = P(rp(\varphi, u(0) \ldots u(t - 1), i), u_i(t), AP_i)$;
4. $\bigcup_{i \in \sum_{\{\text{state}\}} P(\varphi, u, i)} \subseteq AP_i \implies rp(\varphi, u, i) = rp(\varphi, u)$;
5. For $\{i_1, \ldots, i_n\} = \{1, |M|\}$: $rp(rp(\ldots rp(\varphi, u, i_1), \ldots), i_n) = rp(\varphi, u)$.

**Proof** The proofs of these properties can be done by induction on $\varphi \in \text{LTL}$ and follow directly from the definitions of $rp$ and the progression function. □

**Lemma 8** Given a trace and a local obligation on a monitor obtained by running the decentralised algorithm from an initial obligation on the trace, the local obligation where past sub-formulae have been evaluated using the trace is equal to the initial obligation progressed with this same trace. Formally:

$$\forall u \in \Sigma^+, \forall i \in \{1, |M|\}, \forall t \in \mathbb{N}^+, \ |u| = t + 1 \land \text{lo}(i, t, \varphi) \neq \# \implies \text{rp(lo}(i, t, \varphi), u) = P(\varphi, u).$$

**Proof** We shall prove this lemma by induction on $t \in \mathbb{N}^*$. Let us consider some component $M_i$ where $i \in \{1, |M|\}$.

- For $t = 0$. In this case, $|u| = 1$ and we have $\text{rp(lo}(i, 0, \varphi), u) = \text{rp}(P(\varphi, \sigma_i, AP_i))$ where $\sigma_i = \Pi(u(0))$. We can obtain the expected result by doing an induction on $\varphi \in \text{LTL}$ where the only interesting case is $\varphi = p \in AP$. According to the definition of the progression function, we have:

$$P(p, \sigma_i, AP_i) = \begin{cases} T & \text{if } p \in \sigma_i, \\ \bot & \text{otherwise,} \end{cases}$$

Moreover, $p \in \sigma_i$ implies $p \in u(0)$ and $p \notin \sigma_i$ with $p \in AP_i$ implies $\forall j \in \{1, |M|\}, p \notin H_j(u(0))$, i.e., $p \notin u(0)$.

On the one hand, according to the definition of $rp$, we have:

$$\text{rp(Xp, u(0))} = \left\{ \begin{array}{ll} T, & \text{if } p \in u(0), \\ \bot, & \text{otherwise}. \end{array} \right.$$

Thus, we have:

$$\text{rp}(P(p, \sigma_i, AP_i)) = \left\{ \begin{array}{ll} T, & \text{if } p \in u(0), \\ \bot, & \text{otherwise}. \end{array} \right.$$

On the other hand, according to the definition of the progression function, we have:

$$P(\varphi, u(0)) = \left\{ \begin{array}{ll} T, & \text{if } p \in u(0), \\ \bot, & \text{otherwise}. \end{array} \right.$$

- Let us consider some $t \in \mathbb{N}^*$ and suppose that the property holds. We have:

$$\text{lo}(i, t + 1, \varphi) = P(\text{kept}(i, t) \land \text{received}(i, t), u_i(t + 1), AP_i).$$

Similarly to the proof of Lemma 10, let us distinguish four cases according to the communication that occurred at the end of time $t$.

- If $\text{send}(i, t) = \text{false}$ and $\forall j \in \{1, |M|\} \setminus \{i\}$, $\text{send}(j, t, i) = \text{false}$. Then, we have:

$$\text{lo}(i, t + 1, \varphi) = P(\text{lo}(i, t, \varphi), \text{u}_i(t + 1), AP_i).$$

Let us now compute $\text{rp(lo}(i, t + 1, \varphi), u(0) \ldots u(t + 1))$:

$$\text{rp(lo}(i, t + 1, \varphi), u(0) \ldots u(t + 1)) = \text{rp}(\text{rp(lo}(i, t, \varphi), \text{u}_i(t + 1), AP_i), u(0) \ldots u(t + 1)) \quad (\text{Lemma 7, item 1})$$

(by induction hypothesis)

$$= \text{rp}(\text{rp(lo}(i, t, \varphi), u(0) \ldots u(t)), \text{u}_i(t + 1), AP_i), u(0) \ldots u(t + 1)) \quad (\text{Lemma 7, item 2})$$

=$$ \text{P}(\text{P}(\varphi, u(0) \ldots u(t)), u(t + 1), AP) \quad (\text{P}(\varphi, u(0) \ldots u(t)) \text{is a future formula})$$

$$= \text{P}(\varphi, u(0) \ldots u(t + 1))$$

39
Lemma 9

Let \( \varphi \) be an LTL formula, \( \sigma \in \Sigma \) an event, \( \sigma_i \) a local event observed by monitor \( M_i \), and \( w \) an infinite trace, we have \( \sigma \cdot w \models \varphi \iff (\sigma \cdot w)^1 \models P(\varphi, \sigma_i, \Sigma_i) \).
Proof We shall prove that:

\[ \forall i \in [1, n], \forall \varphi \in \text{LTL}, \forall \sigma \in \Sigma, \forall \sigma_i \in \Sigma_i, \forall w \in \Sigma^w. \]

\[ \sigma \cdot w \models \varphi \iff \{\sigma \cdot w\}^l = P(\varphi, \sigma, AP_i). \]

The proof is done by induction on the formula \( \varphi \in \text{LTL} \). Notice that when \( \varphi \) is not an atomic proposition, the lemma reduces to Lemma 1. Thus, we just need to treat the case \( \varphi = p \in AP \).

Suppose \( \varphi = p \in AP \). We have \( \sigma \cdot w \models p \iff p \in \sigma \). Let us consider \( i \in [1, n] \), according to the definition of the progression function (1):

\[ P(p, \sigma, AP_i) = \begin{cases} 
\top & \text{if } p \in \sigma, \\
\perp & \text{if } p \notin \sigma \land p \in AP_i, \\
\Xi_p & \text{otherwise}.
\end{cases} \]

Let us distinguish three cases.

- Suppose \( p \in \sigma \). On the one hand, we have \( p \in \sigma \) and then \( \sigma \cdot w \models p \). On the other hand, we have \( P(p, \sigma, AP_i) = \top \) and thus \( w = P(p, \sigma, AP_i) \).

- Suppose \( p \notin \sigma \) and \( p \in AP_i \). One the one hand, we have \( p \in \sigma \) and, because \( p \in AP_i \) we have \( p \notin \sigma \) and thus \( \sigma \cdot w \models p \). On the other hand, we have \( P(p, \sigma, AP_i) = \perp \).

- Suppose \( p \notin \sigma \) and \( p \notin AP_i \), we have \( (\sigma \cdot w)^l \models \Xi_p \iff ((\sigma \cdot w)^{-1})^l \models \Xi_p \iff \sigma \cdot w \models p \).

\( \square \)

The following lemma states that “the satisfaction of an LTL formula” is propagated by the decentralised monitoring algorithm.

Lemma 10

\[ \forall t \in \mathbb{N}^\geq, \forall i \in [1, n], \forall \varphi \in \text{LTL}, \forall w \in \Sigma^w. \text{inlo}(i, t, \varphi) \neq \# \implies (w \models \varphi \iff w^t \models \text{inlo}(i, t, \varphi)). \]

Proof The proof is done by induction on \( t \in \mathbb{N}^\geq \).

- For \( t = 0 \), the proof is trivial since \( \forall i \in [1, n], \forall \varphi \in \text{LTL}, \text{inlo}(i, 0, \varphi) = \varphi \) and \( w^0 = w \).

- Let us consider some \( t \in \mathbb{N}^\geq \) and suppose that the lemma holds. Let us consider \( i \in [1, n] \), we have:

\[ \text{inlo}(i, t + 1, \varphi) = \text{kept}(i, t) \land \text{received}(1, t + 1). \]

Let us now distinguish four cases according to the communication performed by local monitors at the end of turn \( t \), i.e., according to \( \text{send}(i, t) \) and \( \text{send}(j, t, i) \), for \( j \in [1, n] \setminus \{i\} \).

- If \( \text{send}(i, t) = \text{false} \) and \( \exists j \in [1, n] \setminus \{i\}. \text{send}(j, t, i) = \text{true} \). Then, we have:

\[ \text{inlo}(i, t + 1, \varphi) = P(\text{inlo}(i, t, \varphi) \land \bigwedge_{j \in J} \text{inlo}(j, t, \varphi), u_i(t + 1), \Sigma_i). \]

where \( \forall j \in J. \text{send}(j, t, i) = \text{true} \). Applying the definition of the progression function, we have:

\[ \text{inlo}(i, t + 1, \varphi) = P(\text{inlo}(i, t, \varphi), u_i(t + 1), \Sigma_i) \land \bigwedge_{j \in J} P(\text{inlo}(j, t, \varphi), u_i(t + 1), \Sigma_i). \]

Now, we have:

\[ w^{t+1} \models \text{inlo}(i, t + 1, \varphi) \iff \begin{cases} 
(w^{t+1})^l \models P(\text{inlo}(i, t, \varphi), u_i(t + 1), \Sigma_i) \\
\land \left( \forall j \in J, w^{t+1} \models P(\text{inlo}(j, t, \varphi), u_i(t + 1), \Sigma_i) \right)
\end{cases} \]

With:

\[ w^{t+1} \models P(\text{inlo}(i, t, \varphi), u_i(t + 1), \Sigma_i) \iff (w^t)^l \models P(\text{inlo}(i, t, \varphi), u_i(t + 1), \Sigma_i) \]

\[ (w^{t+1})^l = (w^t)^l \]

(Induction Hypothesis)
And similarly:
\[ \forall j \in J. \, w^{t+1} \models P \{ \text{inlo}(j, t, \varphi), u_i(t + 1), \Sigma_i \} \iff w^t \models \text{inlo}(j, t, \varphi) \]

It follows that:
\[ w^{t+1} \models \text{inlo}(i, t + 1, \varphi) \iff w^t \models \text{inlo}(i, t, \varphi) \land \bigwedge_{j \in J} w^t \models \text{inlo}(i, t, \varphi). \]

And finally:
\[ w^{t+1} \models \text{inlo}(i, t + 1, \varphi) \iff w^t \models \text{inlo}(i, t, \varphi). \]

- If send\((i, t) = \text{true} \) and \( \exists j \in [1, n] \setminus \{i\}. \) send\((j, t, i) = \text{true} \). Then, we have:
  \[ \text{inlo}(i, t + 1, \varphi) = P(\# \land \bigwedge_{j \in J} \text{inlo}(j, t, \varphi), u_i(t + 1), \Sigma_i) \]
  \[ = P(\bigwedge_{j \in J} \text{inlo}(j, t, \varphi), u_i(t + 1), \Sigma_i) \]
  where \( \forall j \in J. \) send\((j, t, i) = \text{true} \). The previous reasoning can be followed in the same manner to obtain the expected result.

- If send\((i, t) = \text{false} \) and \( \forall j \in [1, n] \setminus \{i\}. \) send\((j, t, i) = \text{false} \). Then, we have:
  \[ \text{inlo}(i, t + 1, \varphi) = P(\#, u_i(t + 1), \Sigma_i) \]
  The previous reasoning can be followed in the exact same manner to obtain the expected result.

- If send\((i, t) = \text{true} \) and \( \forall j \in [1, n] \setminus \{i\}. \) send\((j, t, i) = \text{true} \). Then, we have:
  \[ \text{inlo}(i, t + 1, \varphi) = P(\#, u_i(t + 1), \Sigma_i) = \# \]
  In this case, the result holds vacuously.

\[ \square \]

Back to the proof of soundness. The soundness of Algorithm L is now a straightforward consequence of Lemmas 9 and 10. Indeed, let us consider \( u \in \Sigma^* \) s.t. \( |u| = t \). We have \( u \models \varphi \iff \text{true} \) implies that \( \exists i \in [1, n]. \) \( \text{lo}(i, t, \varphi) = \# \) and then \( \text{inlo}(i, t + 1, \varphi) = \# \). It implies that \( \forall w \in \Sigma^*, w \models \text{inlo}(i, t + 1, \varphi). \) Since \( |u| = t \), it follows that \( \forall w \in \Sigma^*, (u \cdot w)^t \models \text{inlo}(i, t + 1, \varphi). \) Applying Lemma 10, we have \( \forall w \in \Sigma^*, u \cdot w \models \varphi \), i.e., \( u \models \varphi \iff t \).

The proof for \( u \models \varphi = \bot \iff u \models \varphi = \bot \) is similar. \[ \square \]

A.2.6 Proof of Theorem 4 (p. 15)

The proof of Theorem 4 intuitively consists in showing that in a given architecture, we can take successively two components and merge them to obtain an equivalent architecture in the sense that they produce the same verdicts. The difference is that if in the merged architecture a verdict is emitted, then, in the non-merged architecture the same verdict will be produced with an additional delay.

**Lemma 11**  In a two-component architecture, if in the centralised case a verdict is produced for some trace \( u \), then, in the decentralised case, one of the monitors will produce the same verdict, after any event. Formally:
\[ \forall \varphi \in \text{LTL}, \forall u \in \Sigma^+. \, P(\varphi, u) = \text{T} \iff \forall \varphi \in \Sigma, \forall i \in [1, 2]. \, \text{lo}(i, u \cdot \sigma_i), \varphi = \text{T} \iff \text{T}. \]

**Proof**  Let us consider a formula \( \varphi \in \text{LTL} \) and a trace \( u \in \Sigma^+ \) s.t. \( |u| = t + 1 \). We shall only consider the case where \( P(\varphi, u) = \text{T} \). The other case is symmetrical. Let us suppose that \( \text{lo}(1, t, \varphi) \neq \text{T} \) and \( \text{lo}(2, t, \varphi) \neq \text{T} \) (otherwise the result holds immediately). Because of the correctness of the algorithm (Theorem 3), we know that \( \text{lo}(1, t, \varphi) \neq \bot \) and \( \text{lo}(2, t, 1) \neq \bot \). Moreover, according to Lemma 8, necessarily \( \text{lo}(1, t, \varphi) \) and \( \text{lo}(2, t, \varphi) \) are urgent formulae: \( \forall \varphi' \text{lo}(1, t, \varphi') > 0 \) and \( \forall \varphi' \text{lo}(2, t, \varphi') > 0 \). Since, there are only two components in the considered architecture, we have \( \bigcup_{\varphi' \in \text{Sus}(\text{lo}(1, t, \varphi))} \text{Prop}(\varphi') \subseteq \text{AP}_2 \) and \( \bigcup_{\varphi' \in \text{Sus}(\text{lo}(2, t, \varphi))} \text{Prop}(\varphi') \subseteq \text{AP}_1 \). According to Algorithm L, we have then send\((1, t, 2) = \text{true} \) and send\((2, t, 1) = \text{true} \). Then \( \text{inlo}(1, t + 1, \varphi) = \text{lo}(2, t, \varphi) \land \text{lo}(2, t, \varphi) = \# \). Hence: \( \text{lo}(1, t + 1, \varphi) = P(\text{rp}(\text{lo}(2, t, \varphi), u(0) \cdot \ldots \cdot u(t), 1), t + 1, \text{AP}_1) \). According to Lemma 7 item 4, we have \( \text{lo}(1, t + 1, \varphi) = P(\text{rp}(\text{lo}(2, t, \varphi), u(0) \cdot \ldots \cdot u(t), 1), t + 1, \text{AP}_1) \). Since
\[ \bigcup_{\varphi' \in \text{Sus}(\text{lo}(2, t, \varphi))} \text{Prop}(\varphi') \subseteq \text{AP}_1 \]
we have $rp\{lo(2, t, \varphi), u(0) \cdot \cdot \cdot u(t), 1\} = rp\{lo(2, t, \varphi), u(0) \cdot \cdot \cdot u(t)\}$. It follows that:

$$
lo(1, t, \varphi) = P(rp(lo(2, t - 1, \varphi), u(0) \cdot \cdot \cdot u(t)), u_1(t + 1), AP_1)
= P(P(\varphi, u(0) \cdot \cdot \cdot u(t)), u_1(t + 1), AP_1)
= P(\top, u_1(t + 1), AP_1) = \top.
$$

(Lemma 8)

Symmetrically, we can find that $lo(2, t, \varphi) = \top$. □

Given two components $C_1$ and $C_2$ with two monitors attached $M_1$ and $M_2$ observing respectively two partial traces $u_1$ and $u_2$ of some global trace $u$. The alphabets of $C_1$ and $C_2$ are $\Sigma_1$ and $\Sigma_2$ built over $AP_1$ and $AP_2$, respectively. Consider the architecture $\mathcal{C} = \{C_1, C_2\}$ with the set of monitors $\mathcal{M} = \{M_1, M_2\}$. Let us define the new component $merge(C_1, C_2)$ that produces events in $2^{AP_1 \cup AP_2}$. To component $merge(C_1, C_2)$ is attached a monitor $M$ observing events in the same alphabet. Then, let us consider architecture $\mathcal{C}' = \{merge(C_1, C_2)\}$ which is a one-component architecture with the set of monitors $\mathcal{M}' = \{merge(M_1, M_2)\}$.

We can parameterise the satisfaction relation of LTL formulae according to the considered architecture. The relation $|=_{D}$ becomes $|=_{D}'$ where $M$ is the considered architecture. The definition of $|=_{D}'$ is the same as the definition of $|=_{D}$ (Definition 7).

**Lemma 12** Considering the monitoring architectures $\mathcal{M} = \{M_1, M_2\}$ and $\mathcal{M}' = \{merge(M_1, M_2)\}$ (where monitors of $\mathcal{M}$ are merged), we have:

$$
\forall u \in \Sigma^+. \forall \varphi \in \text{LTL}. u \models_{D}' \varphi = \top/\bot \implies \forall \sigma \in \Sigma. u \cdot \sigma \models_{D}' \varphi = \top/\bot.
$$

**Proof** This is a direct consequence of Lemma 11. Indeed, $\mathcal{M}'$ is a one-component architecture, thus $u \models_{D}' \varphi = \top/\bot$ implies $P(\varphi, u) = \top/\bot$. Now, since $\mathcal{M}$ is a two-component architecture, using Lemma 11, for all $\sigma \in \Sigma$, there exists $i \in [1, |\mathcal{M}|]$ s.t. $lo(i, [u \cdot \sigma], \varphi) = \top/\bot$. That is $u \cdot \sigma \models_{D}' \varphi = \top/\bot$. □

In the remainder, we consider an $n$-component architecture $\mathcal{M}$, with $n \geq 2$ such that the priority between components is $M_1 < M_2 < \ldots < M_n$.\(^6\)

The following lemma relates verdict production in an $n$-component architecture and verdict production in the same architecture where the two components with the lowest priority have been merged.

**Lemma 13** Let us consider the architecture $\mathcal{M}' = \{merge(M_1, M_2), M_3, \ldots, M_n\}$ where the two components with the lowest priority $M_1$ and $M_2$ have been merged. We have:

$$
\forall u \in \Sigma^+. \forall \varphi \in \text{LTL}. u \models_{D}' \varphi = \top/\bot \implies \forall \sigma \in \Sigma. u \cdot \sigma \models_{D}' \varphi = \top/\bot.
$$

**Proof** We give a proof for the case where the verdict is $\top$ (the other case is symmetrical). Let us consider $u \in \Sigma^+, \varphi \in \text{LTL}$ s.t. $u \models_{D}' \varphi = \top$. Let $u'$ be the smallest prefix of $u$ s.t. $P(\varphi, u') = \top$. From the theorem about the maximal delay (Theorem 2), we have $|u| - |u'| \leq (n - 1)$. Thus, each of the local obligations in architecture $\mathcal{M}'$ will transit through at most $n$ monitors following the ordering between components. That is, in the worst case (i.e., if a verdict is not produced before time $|u|$), any obligation will be progressed according to all components. More precisely, each time a local obligation is progressed on some component $C_i$, past obligations w.r.t. component $C_i$ are removed (Lemma 7 - item 3). Using the compositionality of $rp$ and the progression function on conjunction, in the worst case the local obligation at time $|u'| + n$ will be a conjunction of formulae of the form

$$
P(\ldots, P(rp(\ldots rp(rp(\varphi, u', i_1), u', i_1), \ldots, u', i_n), u_1(|u'|), AP_1)$$
$$\ldots, u_{n-1}(|u'| + 1, AP_{n-1}),$$
$$\ldots, u_{n}(|u'| + n, AP_{n}))$$

\(^6\) Without loss of generality, we assume that the priority of a monitor is given by its index. If this hypothesis does not hold initially, the indexes of components can be re-arranged prior to monitoring so that this hypothesis holds.
where $\varphi$ is a local obligation at time $|u'|$ and $\{i_1, \ldots, i_n\} \supseteq [1, |\mathcal{M}'|]$ (because of the ordering between components). According to Lemma 7 - item 5, we have:

$$rp(\ldots rp(rp(\varphi, u', i), u', i_1) \ldots, i_n) = rp(\varphi, u') = \top.$$ 

Following the definition of the progression function for $\top$, necessarily, the resulting local obligation at time $|u'| + n$ is $\top$. □

**Lemma 14** Let us consider the architecture $\mathcal{M}' = \{\text{merge}(M_n, \text{merge}(\ldots, \text{merge}(M_2, M_1))\}$, then we have:

$$\forall u \in \Sigma^+. \forall \varphi \in \text{LTL}. (\forall u \models_{\mathcal{M}'} \varphi = \top / \bot \implies \forall u' \in \Sigma^+. |u'| \geq n \implies u \cdot u' \models_{\mathcal{M}'} = \top / \bot).$$

**Proof** By an easy induction on the number of components merged using Lemma 13. □

**Proof of Theorem 4 (p. 15).** Based on Lemma 14, we can easily show Theorem 4.

**Proof** Let us consider an $n$-component architecture $\mathcal{M} = \{M_1, \ldots, M_n\}$, a trace $u \in \Sigma^+$ and a formula $\varphi \in \text{LTL}$. Let us suppose that $u \models_{\mathcal{M}} \varphi = \top / \bot$. As the alphabets of monitors are respectively $\Sigma_1, \ldots, \Sigma_n$ and each monitor $M_i$ is observing a sub-trace $u_i$ of $u$ where the hypothesis about alphabets partitioning mentioned in Section 2 holds, we can consider the centralised setting equivalent to monitoring with architecture $\mathcal{M}' = \{\text{merge}(M_n, \text{merge}(\ldots, \text{merge}(M_2, M_1))\}$ where there is a unique monitor $M$ observing the same trace $u$. Using Lemma 14, from $u \models_{\mathcal{M}'} \varphi = \top / \bot$, we get $\forall u' \in \Sigma^+. |u'| \geq n \implies u \cdot u' \models_{\mathcal{M}'} = \top / \bot$, as required. □

44