Propositional Dynamic Logic and Asynchronous Cascade Decompositions for Regular Trace Languages

Bharat Adsul, Paul Gastin, Saptarshi Sarkar, Pascal Weil

Propositional Dynamic Logic and Asynchronous Cascade Decompositions for Regular Trace Languages

Bharat Adsul
IIT Bombay, Mumbai, India

Paul Gastin
Université Paris-Saclay, ENS Paris-Saclay, CNRS, LMF, 91190, Gif-sur-Yvette, France
CNRS, ReLaX, IRL 2000, Siruseri, India

Saptarshi Sarkar
IIT Bombay, Mumbai, India

Pascal Weil
CNRS, ReLaX, IRL 2000, Siruseri, India
Univ. Bordeaux, LaBRI, CNRS UMR 5800, F-33400 Talence, France

Abstract

One of the main motivations for this work is to obtain a distributed Krohn-Rhodes theorem for Mazurkiewicz traces. Concretely, we focus on the recently introduced operation of local cascade product of asynchronous automata and ask if every regular trace language can be accepted by a local cascade product of "simple" asynchronous automata.

Our approach crucially relies on the development of a *local* and *past-oriented* propositional dynamic logic (LocPastPDL) over traces which is shown to be expressively complete with respect to all regular trace languages. An event-formula of LocPastPDL allows to reason about the causal past of an event and a path-formula of LocPastPDL, localized at a process, allows to march along the sequence of past-events in which that process participates, checking for local regular patterns interspersed with local tests of other event-formulas. We also use additional constant formulas to compare the leading process events from the causal past. The new logic LocPastPDL is of independent interest, and the proof of its expressive completeness is rather subtle.

Finally, we provide a translation of LocPastPDL formulas into local cascade products. More precisely, we show that every LocPastPDL formula can be computed by a restricted local cascade product of the gossip automaton and localized 2-state asynchronous reset automata and localized asynchronous permutation automata.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Modal and temporal logics; Theory of computation → Algebraic language theory

Keywords and phrases Mazurkiewicz traces, propositional dynamic logic, regular trace languages, asynchronous automata, cascade product, Krohn Rhodes theorem

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2022.28

1 Introduction

Mazurkiewicz traces form a well-established model of concurrency [11], allowing in particular to describe distributed processes synchronising via shared actions. The concept of recognizability captures important properties of Mazurkiewicz trace languages and it is natural to ask whether recognizable (or regular) trace languages can always be decomposed as a product of simpler languages – in an appropriate formalism.

This is done in the case of regular languages of finite words by the Krohn-Rhodes theorem [16], which states that every regular language is recognized by a cascade product of simple automata, namely reset and permutation automata. The Krohn-Rhodes theorem has

wide-ranging applications, in particular it offers a path to the proof of certain properties of regular languages or of logical fragments by induction on the length of the cascade product [20]. One of the two main results of this paper is a distributed version of this theorem.

Earlier work by the authors [1, 2] established a similar result, but only for first-order definable trace languages. The main ingredient of that proof was the study of a fragment of local temporal logic on traces. As temporal logic specifies only first-order definable languages, it could not be used to cover all regular trace languages and new ideas had to be introduced: we define here a local and past-oriented propositional dynamic logic (LocPastPDL) over traces and our second main result is that this logic is expressively complete with respect to all regular trace languages, which is of independent interest.

Before we say more about our results, let us point out the general philosophy of our work: the remarkable development of the theory of regular languages of finite words has used a triple approach: automata-theoretic, logical and algebraic. An example of this approach is the characterization of star-free languages by counter-free automata, by the aperiodicity of their syntactic monoid, by first-order (FO) definability, or by definability in linear temporal logic. See [10] for a survey on first-order definable word languages.

Our work is situated in an effort to apply the same philosophy to the study of regular trace languages. Many results of this sort already exist. In particular, regular trace languages are characterized by Zielonka's asynchronous automata [24] (see Section 5), and by MSO (monadic second-order) definability (Thomas [22]). Star-freeness is equivalent to FO-definability (Ebinger, Muscholl [12]) and to definability in several global or local temporal logics (Thiagarajan, Walukiewicz [21], Diekert, Gastin [8, 9]). Star-free trace languages are also characterized by the aperiodicity of their syntactic monoids (Guaiana, Restivo, Salemi [15]), and Kufleitner [17] gave algebraic and combinatorial characterizations of certain fragments of local linear temporal logic. A discussion of these developments, and of why there are not more algebraic characterizations of significant classes of regular trace languages can be found in [1, 2].

In this paper, traces are viewed as implemented over a distributed architecture: each action (each letter of the alphabet) is *located* over a non-empty subset of *processes* from a finite set $\mathcal{P}$, and this location determines which pairs of letters are independent. This view of traces is what informs the definition of asynchronous automata [24]. As in [19, 1, 2], we use asynchronous automata not just as acceptors, using accepting states, but also as machines locally computing relabeling functions for input traces (similar in spirit to the sequential letter-to-letter transducers on words). The composition of these relabeling functions is captured by our notion of a *local cascade product*.

The precise statement of our Krohn-Rhodes theorem for trace languages uses also the notion of a *restricted* local cascade product with the *gossip automaton*. The latter is an asynchronous automaton introduced by Mukund and Sohoni [19], which is entirely determined by the distributed architecture under consideration. The information contributed by the gossip automaton in a *restricted* local cascade product is limited to the event-level and the trace-level comparisons of the order that may exist within the trace of the *latest views* of the different processes.

As indicated above, (local) temporal logic is not suitable to discuss trace languages that are not FO-definable and we turn to propositional dynamic logic (PDL). This logic was introduced by Fischer and Ladner [13] as a way to reason about programs. Over finite words, an appropriate version called LDL was shown to be expressively complete with respect to MSO by Giacomo and Vardi [7], see also [23]. PDL has been applied in different forms to various structures, e.g., Kripke structures [14], message-sequence charts (MSC) and message-passing

systems [6, 18]. Recently, Bollig, Fortin and Gastin [4] introduced a star-free version of PDL interpreted on MSC, and showed its expressively completeness with respect to first-order logic. There are however not many results using PDL on traces (see [5, Chapter 5]).

More specifically we introduce a past-oriented fragment of PDL, called PastPDL, and a local fragment of it, called LocPastPDL. An event-formula of LocPastPDL allows to reason about the causal past of an event and a path-formula of LocPastPDL, localized at a process, allows to march along the sequence of past-events in which that process participates, checking for local regular patterns interspersed with local tests of other event-formulas. We also use additional constant formulas to compare the leading events for each process, in the strict causal past of a given event. The proof of the expressive completeness of PastPDL and LocPastPDL is rather subtle and the result is of independent interest.

The paper is organized as follows. Section 2 lays down the basic notion and terminology for traces over distributed architecture and the PDL fragments PastPDL and LocPastPDL are introduced in Section 3.

The precise statement on the expressive completeness of PastPDL and LocPastPDL, Theorem 3, is proved in Section 4. The proof is by induction on the number of processes in the distributed architecture. The case of a single process corresponds to the expressive completeness of linear dynamic logic [7]. Generalizing this to several processes is highly non-trivial. It crucially depends on a lifting lemma which constructs a formula $\mathsf{lift}_P(\varphi)$ from a formula φ in PastPDL so that φ holds on a suffix s of a (prime) trace $t = rs$ if and only if $\mathsf{lift}_P(\varphi)$ holds on t – where the suffix s is determined by a subset P of processes.

In Section 5, we briefly describe asynchronous automata, and the important notion of *asynchronous labeling functions* computed by such automata, introduced in [1, 2]. As mentioned above, the latter notion generalizes sequential transducers, and is very close to the locally computable functions of Mukund and Sohoni [19]. In the same section, we explain how the composition of asynchronous labeling functions corresponds to the local cascade product operation on asynchronous automata, and we define the notion of the restricted local cascade products of the gossip automaton and an arbitrary asynchronous automaton.

Our main decomposition theorem, Theorem 16, is proved in Section 6. It shows how any LocPastPDL event formula is *computed* by a restricted local cascade product of a copy of the gossip automaton, followed by a cascade product of localized reset and permutation automata. A Krohn-Rhodes-like statement, Corollary 17, follows immediately.

2 Mazurkiewicz traces

We consider (Mazurkiewicz) traces as implemented over a distributed architecture. More precisely, we fix a finite set $\mathcal{P}$ of processes. A *distributed alphabet* over $\mathcal{P}$ is a pair (Σ, loc) where the *location function* $\mathsf{loc}\colon \Sigma \to 2^{\mathcal{P}} \setminus \{\emptyset\}$ assigns to each letter $a \in \Sigma$ the set of processes which *participate* in a. For $i \in \mathcal{P}$, we let $\Sigma_i = \{a \in \Sigma \mid i \in \mathsf{loc}(a)\}$. The location function induces an *independence relation* over Σ: letters a and b are *independent* if $\mathsf{loc}(a) \cap \mathsf{loc}(b) = \emptyset$, and they are *dependent* otherwise.

When dealing with posets, and in particular with traces, we use the following notation. If $(E, \leq)$ is a poset and $e \in E$, we let $\downarrow e$ (the *past* of e) be the set $\{f \in E \mid f \leq e\}$, and we let $\Downarrow e = \downarrow e \setminus \{e\}$ (the *strict past* of e). If $X \subseteq E$, we let $\downarrow X = \bigcup_{e \in X} \downarrow e$.

A *trace* over (Σ, loc) is a triple $t = (E, \leq, \lambda)$ where $(E, \leq)$ is a finite poset and $\lambda\colon E \to \Sigma$ is a labeling function, such that

- if $e, e' \in E$ and e' is an *immediate successor* of e (that is, $e < e'$ and $e \leq e'' \leq e'$ implies $e'' = e$ or $e'' = e'$), then $\lambda(e)$ and $\lambda(e')$ are dependent;
- if $e, e' \in E$ and $\lambda(e)$ and $\lambda(e')$ are dependent, then $e \leq e'$ or $e' \leq e$.

The elements of E are traditionally called *events*. Further, if $i \in \mathcal{P}$, E_i denotes the set of *i-events* (*i.e.*, events in which process i participates), namely $E_i = \{e \in E \mid i \in \mathsf{loc}(\lambda(e))\}$. It is clear that E_i is totally ordered by $\leq$.

We let $\mathrm{Tr}(\Sigma, \mathsf{loc})$ denote the set of all traces over (Σ, loc). We write simply $\mathrm{Tr}(\Sigma)$ if loc is clear from the context. The empty trace (where $E = \emptyset$) is written ε.

$\mathrm{Tr}(\Sigma)$ is a monoid for the following *concatenation* operation on traces. Let $t = (E, \leq, \lambda)$ and $t' = (E', \leq', \lambda')$ be elements of $\mathrm{Tr}(\Sigma)$. Without loss of generality, we can assume E and E' to be disjoint. We define tt' to be the trace $(E \cup E', \leq'', \lambda'')$ where

- $\leq''$ is the transitive closure of $\leq \cup \leq' \cup \{(e, e') \in E \times E' \mid \lambda(e) \text{ and } \lambda'(e') \text{ are dependent}\}$,
- $\lambda'' \colon E'' \to \Sigma$ where $\lambda''(e) = \lambda(e)$ if $e \in E$; otherwise, $\lambda''(e) = \lambda'(e)$.

This operation is associative, with the empty trace ε as unit. Hence, $\mathrm{Tr}(\Sigma)$ is a monoid.

A trace t' is said to be a *prefix* (resp. *suffix*) of a trace t if there exists t'' such that $t = t't''$ (resp. $t = t''t'$). Prefixes of t coincide with restrictions of t to downward-closed subsets of events. Prefixes of the form $\downarrow e$ or $\Downarrow e$ ($e \in E$) are important examples.

A *trace language* over Σ is a subset of $\mathrm{Tr}(\Sigma)$. Regular trace languages are characterized by different classical mechanisms: MSO logic, saturation of regular languages of words, asynchronous automata. In this paper, we say that a trace language L is regular if it is *recognized* by a morphism $\eta \colon \mathrm{Tr}(\Sigma) \to M$ to a finite monoid: that is, if $L = \eta^{-1}(\eta(L))$.

3 A propositional dynamic logic for traces

Past propositional dynamic logic. Inspired by the definition of PDL [13] and its version meant to be interpreted on finite words (LDL [7]), we introduce PastPDL, *past propositional dynamic logic*, to reason about Mazurkiewicz traces. The syntax of PastPDL is the following.

$$\Phi ::= \mathsf{EM}\,\varphi \mid \mathsf{L}_i \leq \mathsf{L}_j \mid \mathsf{L}_{i,j} \leq \mathsf{L}_k \mid \Phi \vee \Phi \mid \neg\Phi$$
$$\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle\pi\rangle$$
$$\pi ::= \leftarrow_i \mid \varphi? \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^*$$

Calling this logic *past* is justified by the fact that we allow only backward $\leftarrow_i$ edges and past-oriented constant formulas $\mathsf{L}_i \leq \mathsf{L}_j$, $\mathsf{L}_{i,j} \leq \mathsf{L}_k$, $\mathsf{Y}_i \leq \mathsf{Y}_j$, $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$, (semantics below).

Formulas of the form Φ, φ and π are called, respectively, *trace formulas* or *sentences*, *event formulas* and *path formulas*. A trace formula is evaluated on a trace (and hence it defines a trace language), an event formula is evaluated at an event of a trace, and a path formula is evaluated at a pair of events. The semantics of PastPDL is as follows. Let $t = (E, \leq, \lambda)$ be a trace. For each process $i \in \mathcal{P}$, let E_i denote the set of i-events of t. We let

$$\begin{aligned}
t &\models \mathsf{EM}\,\varphi && \text{if } t, e \models \varphi \text{ for some maximal event } e \text{ in } t,\\
t &\models \mathsf{L}_i \leq \mathsf{L}_j && \text{if } E_i \neq \emptyset,\ E_j \neq \emptyset \text{ and } \max(E_i) \leq \max(E_j),\\
t &\models \mathsf{L}_{i,j} \leq \mathsf{L}_k && \text{if } E_i \neq \emptyset,\ E_j \cap {\downarrow}E_i \neq \emptyset,\ E_k \neq \emptyset \text{ and } \max(E_j \cap {\downarrow}E_i) \leq \max(E_k).
\end{aligned}$$

In other words, a trace satisfies $\mathsf{L}_i \leq \mathsf{L}_j$ if the last event on process i is below the last event on process j, and it satisfies $\mathsf{L}_{i,j} \leq \mathsf{L}_k$ when the maximal event on process j which is below some event on process i is below the last event on process k.

Note also that $t \models \mathsf{EM}\,\varphi$ implies in particular that the trace t is nonempty, so the sentence $\neg\,\mathsf{EM}\,\top$ defines the empty trace. Also, $\mathsf{L}_i \leq \mathsf{L}_i$ simply means that $E_i \neq \emptyset$, i.e., the trace contains some i-event.

We now turn to event and path formulas. Recall that if $t = (E, \leq, \lambda)$ is a trace and $e \in E$ is an event, then $\Downarrow e$ denotes the strict past of e in t. For a process $i \in \mathcal{P}$, we denote by e_i the unique maximal event of $\Downarrow e \cap E_i$, if it exists, i.e., if $\Downarrow e \cap E_i \neq \emptyset$. If $j \in \mathcal{P}$, we write $e_{i,j}$ for $(e_i)_j$, that is, for the maximal event of $\Downarrow e_i \cap E_j$ if e_i exists and $\Downarrow e_i \cap E_j \neq \emptyset$. If e, f are events of t, we let

$$
\begin{array}{rl}
t, e \models a & \text{if } \lambda(e) = a \\
t, e \models \mathsf{Y}_i \leq \mathsf{Y}_j & \text{if } e_i, e_j \text{ exist and } e_i \leq e_j \\
t, e \models \mathsf{Y}_{i,j} \leq \mathsf{Y}_k & \text{if } e_{i,j}, e_k \text{ exist and } e_{i,j} \leq e_k \\
t, e \models \langle \pi \rangle & \text{if there exists an event } f \in E \text{ such that } t, e, f \models \pi \\
t, e, f \models \leftarrow_i & \text{if } f \text{ is the immediate predecessor of } e \text{ on process } i \\
t, e, f \models \varphi? & \text{if } e = f \text{ and } t, e \models \varphi \\
t, e, f \models \pi_1 + \pi_2 & \text{if } t, e, f \models \pi_1 \text{ or } t, e, f \models \pi_2 \\
t, e, f \models \pi_1 \cdot \pi_2 & \text{if there is an event } g, \text{ such that } t, e, g \models \pi_1 \text{ and } t, g, f \models \pi_2 \\
t, e, f \models \pi^* & \text{if there are events } e = e_0, e_1, \ldots, e_n = f \text{ with } n \geq 0 \\
 & \text{and } t, e_i, e_{i+1} \models \pi \text{ for all } 0 \leq i < n
\end{array}
$$

We sometime use $\langle \pi \rangle \varphi$ as a macro for $\langle \pi \cdot \varphi? \rangle$. For instance, we may easily express a *strict since* modality restricted to events on a specified process. More precisely, if $i \in \mathcal{P}$ is a process and φ, ψ are event formulas, then the event formula $\langle (\leftarrow_i \cdot \varphi?)^* \cdot \leftarrow_i \psi? \rangle$, denoted $\varphi \, \mathsf{S}_i \, \psi$, holds at some event e of a trace t when there is a sequence $f_n, f_{n-1}, \ldots, f_1, f_0 = e$ of *consecutive* i-events ($n > 0$) with $t, f_n \models \psi$ and $t, f_j \models \varphi$ for all $0 < j < n$.

PastPDL is no more expressive than MSO logic.

▶ **Proposition 1.** *For all* PastPDL *sentences* Φ*, event formulas* φ *and path formulas* π*, we can construct* MSO *sentences* $\overline{\Phi}$*, and formulas* $\overline{\varphi}(x)$*,* $\overline{\pi}(x, y)$ *with respectively one or two free first-order variables such that, for all traces* t *and events* e, f *in* t*, we have*

$$
\begin{array}{rcl}
t \models \Phi & \text{if and only if} & t \models \overline{\Phi} \\
t, e \models \varphi & \text{if and only if} & t, x \mapsto e \models \overline{\varphi}(x) \\
t, e, f \models \pi & \text{if and only if} & t, x \mapsto e, y \mapsto f \models \overline{\pi}(x, y)
\end{array}
$$

The proof technique is folklore and is an easy structural induction. For the base case(s), we note that the constant formulas $\mathsf{L}_i \leq \mathsf{L}_j$, $\mathsf{L}_{i,j} \leq \mathsf{L}_k$, $\mathsf{Y}_i \leq \mathsf{Y}_j$ and $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$ all have first-order definitions. For the Kleene star $\pi^\star$, the induction step relies on the well-known fact that transitive closure of an MSO-definable relation can be expressed in MSO.

It will be convenient in the sequel to use automata instead of regular expressions to specify path formulas. Define a *path automaton* to be a tuple of the form $\mathcal{A} = (Q, \Delta, I, F)$, where Q is a finite, non-empty set of states, $I, F \subseteq Q$ are, respectively, the sets of initial and final states, and Δ is a finite set of transitions of the form (q, α, q') with $q, q' \in Q$ and

- either $\alpha = \varphi?$ for some event formula φ (a *test* transition),
- or $\alpha \in \{\leftarrow_i \mid i \in \mathcal{P}\}$ (a *move* transition).

A path automaton specifies a path formula with the expected semantics: for a trace $t = (E, \leq, \lambda)$ and two events $e, f \in E$, we have $t, e, f \models \mathcal{A}$ if there exists an accepting run $q_0 \xrightarrow{\alpha_1} q_1 \cdots q_{n-1} \xrightarrow{\alpha_n} q_n$ and a sequence of events $e = e_0, e_1, \ldots, e_{n-1}, e_n = f$ such that for all $0 \leq m < n$ we have $t, e_m, e_{m+1} \models \alpha_m$. Notice that if $n = 0$ the condition is simply $e = f$.

If $\mathcal{A}$ is a path automaton, then $\langle \mathcal{A} \rangle$ is an event formula where $t, e \models \langle \mathcal{A} \rangle$ if there exists an event $f \in E$ such that $t, e, f \models \mathcal{A}$.

▶ **Proposition 2.** *Path formulas and path automata are equally expressive:*

1. *For each path formula π, we can construct a path automaton $\mathcal{A}_\pi$ such that for all traces t and events e, f we have $t, e, f \models \pi$ if and only if $t, e, f \models \mathcal{A}_\pi$.*
2. *For each path automaton $\mathcal{A}$, we can construct a path formula $\pi_\mathcal{A}$ such that for all traces t and events e, f we have $t, e, f \models \mathcal{A}$ if and only if $t, e, f \models \pi_\mathcal{A}$.*

This is a direct consequence of the equivalence between regular expressions and finite state automata. Indeed, a path formula π can be seen as a regular expression over the alphabet Γ_π consisting of the moves $\leftarrow_i$ ($i \in \mathcal{P}$) and the tests $\varphi?$ which occur at the top level of π. If $\mathcal{A}_\pi$ is an automaton accepting the language of Γ_π specified by π, then $\mathcal{A}_\pi$ is a path automaton which is equivalent to the path formula π. The converse is similarly justified.

Local past propositional dynamic logic. Proposition 2 shows that we may replace $\langle \pi \rangle$ with $\langle \mathcal{A} \rangle$ in the syntax of PastPDL event formulas without changing the expressivity of the logic. Adopting the syntax using path automata allows us to state the definition of LocPastPDL, the *local* fragment of PastPDL. More precisely, say that a path automaton $\mathcal{A}$ is *i-local* for some process $i \in \mathcal{P}$, if all its move transitions are labeled with $\leftarrow_i$. The automaton $\mathcal{A}$ is *local* if it is i-local for some $i \in \mathcal{P}$. The syntax of LocPastPDL is as follows:

$$\Phi ::= \mathsf{EM}\,\varphi \mid \mathsf{L}_i \leq \mathsf{L}_j \mid \mathsf{L}_{i,j} \leq \mathsf{L}_k \mid \Phi \vee \Phi \mid \neg\Phi$$
$$\varphi ::= a \mid \mathsf{Y}_i \leq \mathsf{Y}_j \mid \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid \varphi \vee \varphi \mid \neg\varphi \mid \langle \mathcal{A} \rangle\,.$$

where $i, j, k \in \mathcal{P}$, $a \in \Sigma$ and $\mathcal{A}$ ranges over local path automata.

The semantics of LocPastPDL is inherited from PastPDL. We show in Section 4 that both logics are expressively complete with respect to regular trace languages.

4 Expressivity

The main result in this section is the following.

▶ **Theorem 3.** PastPDL *and* LocPastPDL *are expressively complete, that is: a trace language is regular if and only if it can be defined by a* PastPDL *(resp.* LocPastPDL*) sentence.*

One direction of Theorem 3 is easily taken care of: we saw in Proposition 1 that PastPDL sentences define regular trace languages. Conversely, let L be a regular language, and let η be a morphism from $\mathrm{Tr}(\Sigma)$ to a finite monoid M, recognizing L. Since sentences of LocPastPDL are closed under disjunction, it is enough to show that every trace language of the form $\eta^{-1}(m)$ ($m \in M$) is LocPastPDL-definable.

This is established in two steps. We first deal with prime traces. Recall that a trace $t = (E, \leq, \lambda)$ is *prime* if E has a single maximal event, which we then denote by $\max(t)$. In Theorem 4, we show how to construct a LocPastPDL event formula $\varphi^{(m)}$ such that, if t is a prime trace, then $\eta(t) = m$ if and only if $t, \max(t) \models \varphi^{(m)}$.

Leveraging this partial result to handle all traces – and not just prime traces –, is done in Theorem 7.

4.1 Expressivity of event formulas in LocPastPDL

As announced, we first show that event formulas in LocPastPDL are expressive enough to describe regular sets of prime traces.

▶ **Theorem 4.** *Let $\eta\colon \mathrm{Tr}(\Sigma) \to M$ be a morphism to a finite monoid. For each $m \in M$, we can construct a* LocPastPDL *event formula $\varphi^{(m)}$ such that, if $t \in \mathrm{Tr}(\Sigma)$ is a prime trace, then $\eta(t) = m$ if and only if $t, \max(t) \models \varphi^{(m)}$.*

The proof of Theorem 4, to be completed at the end of Section 4.1, is by induction on the number of processes. We start with a high-level description of this proof. Let t be a prime trace, let $e = \max(t)$ and let k be a process such that $e \in E_k$. Let $f_1 < f_2 < \cdots < f_\ell = e$ be the sequence of events on process k. Then t is equal to the product $t_1 t_2 \cdots t_\ell$ with $t_i = {\downarrow}f_i \setminus {\downarrow}f_{i-1}$ (${\downarrow}f_0 = \emptyset$). Note that t_i is prime, with $\max(t_i) = f_i$, and that t_i has no event on process k apart from f_i: this property of t_i with respect to k opens the door to the usage of the induction hypothesis.

More precisely, we use induction to construct for each $m \in M$ an event formula $\varphi^{(m)}$ such that, for all prime traces $s = {\downarrow}g$ such that g is the only event on process k (and each t_i is of this form), we have $\eta(s) = m$ if and only if $s, g \models \varphi^{(m)}$.

The next task is to "lift" the formula $\varphi^{(m)}$, meant to be interpreted on the factors t_i, to a formula $\mathsf{lift}_k(\varphi^{(m)})$ to be interpreted on the full trace t. This is done in Lemma 6, in such a way that $t_i, f_i \models \varphi^{(m)}$ if and only if $t, f_i \models \mathsf{lift}_k(\varphi^{(m)})$. The difference is subtle: in one case, past modalities are evaluated on a scope contained in t_i, whereas in the other, their scope may span the full past of f_i in t, i.e., $t_1 \cdots t_i$. The lifted formula has to ensure that one never goes below f_{i-1}.

The particular properties of the t_i which make this possible are abstracted out, and generalized, by what we call *residues*. Somewhat informally, if P is a set of processes and g is an event, we let $\mathsf{res}_P(g)$ (the residue of the event g with respect to P) be the largest suffix of ${\downarrow}g$ which does not contain any event on the processes in P except, perhaps, g itself. Notice that $t_i = \mathsf{res}_{\{k\}}(f_i)$. Lemma 6 proves, by structural induction, that event formulas in PastPDL can be lifted with respect to residues.

Lemma 5, which plays a crucial role in the inductive proof of Lemma 6, shows how the set P of processes may increase to some set P' when one moves from an event g to some previous event g'. The determination of this set P' is possible thanks to the event formulas $\mathsf{Y}_i \le \mathsf{Y}_j$ and $\mathsf{Y}_{i,j} \le \mathsf{Y}_k$ (primary and secondary comparisons).

The last step of the proof of Theorem 4 uses a k-local path automaton to visit the sequence of events $f_1 < f_2 < \cdots < f_\ell = e$ backward, checking along the path the values of the $\eta(t_i)$ with event formulas of the form $\mathsf{lift}_k(\varphi^{(m_i)})$ and storing in its state the value of the product $\eta(t_i) \cdots \eta(t_\ell)$: when the automaton has reached f_1, the first event on process k, it has computed $\eta(t_1)\eta(t_2)\cdots\eta(t_\ell) = \eta(t)$.

The precise definition of *residuation* is as follows. For an event e of a trace $t = (E, \le, \lambda)$ and a process $i \in \mathcal{P}$, recall that $e_i = \max({\Downarrow}e \cap E_i)$, if it exists, where E_i is the set of i-events in E. By convention, we let ${\downarrow}e_i = \emptyset$ when e_i does not exist, i.e., when ${\Downarrow}e \cap E_i = \emptyset$.

If $P \subseteq \mathcal{P}$ is a set of processes, the *residue of e by P* is the trace $\mathsf{res}_P(e) = {\downarrow}e \setminus \bigcup_{i \in P} {\downarrow}e_i$. In particular, $\mathsf{res}_P(e)$ is a suffix of the trace ${\downarrow}e$, itself a prefix of t. We will use the following technical result, which makes essential use of the primary and secondary comparison formulas $\mathsf{Y}_i \le \mathsf{Y}_j$ and $\mathsf{Y}_{i,j} \le \mathsf{Y}_k$.

▶ **Lemma 5.** *Let t be a trace, $i \in \mathcal{P}$ a process and $P \subseteq \mathcal{P}$ a set of processes. Let e be an event in t such that e_i exists. Then we have*

$$ {\downarrow}e_i \cap \mathsf{res}_P(e) = \begin{cases} \varepsilon & \text{if } t, e \models \bigvee_{k \in P} \mathsf{Y}_i \le \mathsf{Y}_k, \\ \mathsf{res}_{P'}(e_i) & \text{otherwise,} \end{cases} $$

where $P' = P \cup \{j \in \mathcal{P} \mid t, e \models \mathsf{Y}_{i,j} \le \mathsf{Y}_k \text{ for some } k \in P\}$.

Proof. By definition, we have $\downarrow e_i \cap \mathsf{res}_P(e) = \downarrow e_i \setminus \bigcup_{k\in P} \downarrow e_k$. This is the empty trace if and only if e_i is in one of the $\downarrow e_k$ ($k \in P$), that is, if and only if $t, e \models \bigvee_{k\in P} \mathsf{Y}_i \leq \mathsf{Y}_k$.

Let us now assume that $t, e \not\models \bigvee_{k\in P} \mathsf{Y}_i \leq \mathsf{Y}_k$. Note that $\mathsf{res}_{P'}(e_i) = \downarrow e_i \setminus \bigcup_{k\in P'} \downarrow e_{i,k}$. Let $f \in \bigcup_{k\in P'} \downarrow e_{i,k}$. We have $f \leq e_{i,k}$ for some $k \in P'$. If $k \in P$, then $e_{i,k} \leq e_k$. If $k \notin P$, then $e_{i,k} \leq e_j$ for some $j \in P$. In both cases we have $f \in \bigcup_{k\in P} \downarrow e_k$. Therefore $\bigcup_{k\in P'} \downarrow e_{i,k} \subseteq \bigcup_{k\in P} \downarrow e_k$ and hence

$$\downarrow e_i \cap \mathsf{res}_P(e) = \downarrow e_i \setminus \bigcup_{k\in P} \downarrow e_k \subseteq \downarrow e_i \setminus \bigcup_{k\in P'} \downarrow e_{i,k} = \mathsf{res}_{P'}(e_i).$$

Conversely, let $f \in \mathsf{res}_{P'}(e_i)$. In particular, $f \in \downarrow e_i$. Assume that $f \leq e_k$ for some $k \in P$. Since $t, e \not\models Y_i \leq Y_k$, we know that $e_i \not\leq e_k$. If $e_k < e_i$, then $e_{i,k} = e_k$ and we get $f \in \bigcup_{j\in P'} \downarrow e_{i,j}$, a contradiction.

It follows that the events e_i and e_k are concurrent: $e_i \| e_k$. Let g be a maximal event in $\uparrow f \cap \downarrow e_i \cap \downarrow e_k$ (this set is not empty since it contains f). Then there exists $\ell \in \mathsf{loc}(g)$ such that $g = e_{i,\ell}$. This implies that $\ell \in P'$ and again $f \in \bigcup_{j\in P'} \downarrow e_{i,j}$, a contradiction.

This concludes the proof that $\mathsf{res}_{P'}(e_i)$ is contained in $\downarrow e_i \setminus \bigcup_{k\in P} \downarrow e_k = \downarrow e_i \cap \mathsf{res}_P(e)$. ◀

We now establish the technical core of the proof of Theorem 4, namely the following *lifting lemma*, which turns an event formula satisfied by a residue of a trace t, into another satisfied by the trace t itself.

▶ **Lemma 6** (Lifting lemma). *Let $\varphi \in \mathsf{PastPDL}$ be an event formula and $P \subseteq \mathcal{P}$ be a set of processes. We can construct an event formula $\mathsf{lift}_P(\varphi) \in \mathsf{PastPDL}$ such that, for all traces $t = (E, \leq, \lambda)$ and events $e \in E$ in t, we have $\mathsf{res}_P(e), e \models \varphi$ if and only if $t, e \models \mathsf{lift}_P(\varphi)$. Moreover, if $\varphi \in \mathsf{LocPastPDL}$ then $\mathsf{lift}_P(\varphi) \in \mathsf{LocPastPDL}$.*

Proof. The construction is by structural induction on φ. We first let

$$\begin{aligned}
\mathsf{lift}_P(a) &= a \text{ for each } a \in \Sigma \\
\mathsf{lift}_P(\mathsf{Y}_i \leq \mathsf{Y}_j) &= (\mathsf{Y}_i \leq \mathsf{Y}_j) \wedge \neg \bigvee_{\ell\in P} (\mathsf{Y}_i \leq \mathsf{Y}_\ell) \vee (\mathsf{Y}_j \leq \mathsf{Y}_\ell) \\
\mathsf{lift}_P(\mathsf{Y}_{i,j} \leq \mathsf{Y}_k) &= (\mathsf{Y}_{i,j} \leq \mathsf{Y}_k) \wedge \neg \bigvee_{\ell\in P} (\mathsf{Y}_{i,j} \leq \mathsf{Y}_\ell) \vee (\mathsf{Y}_k \leq \mathsf{Y}_\ell)
\end{aligned}$$

The announced statement is easily verified for these atomic formulas. Similarly, boolean combinations of formulas are handled by letting $\mathsf{lift}_P(\varphi \vee \psi) = \mathsf{lift}_P(\varphi) \vee \mathsf{lift}_P(\psi)$ and $\mathsf{lift}_P(\neg\varphi) = \neg\mathsf{lift}_P(\varphi)$.

The last, and more interesting case, is that where $\varphi = \langle\mathcal{A}\rangle$, for a past path automaton $\mathcal{A} = (Q, \Delta, I, F)$. We let $\mathsf{lift}_P(\langle\mathcal{A}\rangle) = \langle\mathcal{A}_P\rangle$, where $\mathcal{A}_P = (Q', \Delta', I', F')$ is the path automaton defined as follows:

- $Q' = Q \times 2^{\mathcal{P}}$, $I' = I \times \{P\}$ and $F' = F \times 2^{\mathcal{P}}$,
- for each test transition $(q_1, \varphi?, q_2) \in \Delta$ of $\mathcal{A}$ and each set $P_1 \subseteq \mathcal{P}$, we define the test transition $((q_1, P_1), \mathsf{lift}_{P_1}(\varphi)?, (q_2, P_1))$ in $\mathcal{A}_P$,
- for each move transition $(q_1, \leftarrow_k, q_2) \in \Delta$ of $\mathcal{A}$ and each sets $P_1, P_2 \subseteq \mathcal{P}$ with $P_1 \subseteq P_2$, we define a *test and move*[1] transition $((q_1, P_1), \mathsf{change}_{k,P_1,P_2}? \cdot \leftarrow_k, (q_2, P_2))$ in $\mathcal{A}_P$ where

$$\mathsf{change}_{k,P_1,P_2} = \Big(\neg \bigvee_{i\in P_1} \mathsf{Y}_k \leq \mathsf{Y}_i\Big) \wedge \Big(\bigwedge_{j\in P_2\setminus P_1} \bigvee_{i\in P_1} \mathsf{Y}_{k,j} \leq \mathsf{Y}_i\Big) \wedge \Big(\bigwedge_{j\notin P_2} \neg \bigvee_{i\in P_1} \mathsf{Y}_{k,j} \leq \mathsf{Y}_i\Big)$$

[1] Formally, in order to comply with the definition of a path automaton, we should split each test and move transition into a test transition followed by a move transition with a new intermediary state in-between.

The formula above characterises the change of context when moving from an event e on process k to the previous event e_k on process k. It is based on Lemma 5. The first conjunct says that $\mathsf{res}_{P_1}(e) \cap {\downarrow}e_k$ is nonempty. The remaining conjuncts characterises the set P_2 such that $\mathsf{res}_{P_1}(e) \cap {\downarrow}e_k = \mathsf{res}_{P_2}(e_k)$.

Observe that $\mathcal{A}_P$ is again a *past* automaton and all reachable states (q, P') in $\mathcal{A}_P$ satisfy $P \subseteq P'$. Moreover, if $\mathcal{A}$ is k-local then so is $\mathcal{A}_P$.

We claim that this construction is correct, *i.e.*, $\mathsf{res}_P(e), e \models \langle \mathcal{A} \rangle$ if and only if $t, e \models \langle \mathcal{A}_P \rangle$. The proof of this claim is in Appendix A. ◀

We can finally complete the proof of Theorem 4, which shows that, as far as prime traces are concerned, LocPastPDL event formulas can express all regular properties.

Proof of Theorem 4. We establish a more precise statement: for $m \in M$ and $P \subseteq \mathcal{P}$, we construct a LocPastPDL event formula $\varphi_P^{(m)}$ such that, if t is a prime trace satisfying $\mathsf{loc}(t \setminus \{\max(t)\}) \subseteq P$, then $\eta(t) = m$ if and only if $t, \max(t) \models \varphi_P^{(m)}$. The statement of the theorem corresponds to the case $P = \mathcal{P}$.

The proof is by induction on the cardinality of P. If $P = \emptyset$, a prime trace t satisfying the condition $\mathsf{loc}(t \setminus \{\max(t)\}) \subseteq P$ consists of the single event $\max(t)$. Therefore, we let

$$\varphi_\emptyset^{(m)} = \bigvee_{a \in \Sigma \text{ s.t. } \eta(a) = m} a \,.$$

Assume that $P \neq \emptyset$ and consider a prime trace $t = (E, \leq, \lambda)$ satisfying $\mathsf{loc}(t \setminus \{\max(t)\}) \subseteq P$. If $P \cap \mathsf{loc}(\max(t)) = \emptyset$, the primality of t implies that E is a singleton, and we let $\varphi_P^{(m)} = \varphi_\emptyset^{(m)}$.

If $P \cap \mathsf{loc}(\max(t)) \neq \emptyset$, we pick $k \in P \cap \mathsf{loc}(\max(t))$. Let $f_1 < f_2 < \cdots < f_\ell$ be the sequence of events in E_k. In particular, $\ell \geq 1$ and $\max(t) = f_\ell$. For each $1 \leq i \leq \ell$, let $t_i = \mathsf{res}_{\{k\}}(f_i)$. Then $t_i = {\downarrow}f_i \setminus {\downarrow}f_{i-1}$ (letting ${\downarrow}f_0 = \emptyset$) and hence, $t = t_1 t_2 \cdots t_\ell$ and $\eta(t) = \eta(t_1)\eta(t_2)\cdots\eta(t_\ell)$. By construction, $\mathsf{loc}(t_i \setminus \{f_i\}) \subseteq P \setminus \{k\}$ for each $1 \leq i \leq \ell$ and we can use the induction hypothesis: $\eta(t_i) = m'$ if and only if $t_i, f_i \models \varphi_{P\setminus\{k\}}^{(m')}$. Using the lifting lemma (Lemma 6), we then get that $\eta(t_i) = m'$ if and only if $t, f_i \models \mathsf{lift}_{\{k\}}(\varphi_{P\setminus\{k\}}^{(m')})$.

The membership of a trace t in $\eta^{-1}(m)$ – subject to the current assumption that t is prime, $\mathsf{loc}(t \setminus \{\max(t)\}) \subseteq P$ and $k \in P \cap \mathsf{loc}(\max(t))$ – can be computed by a k-local path automaton $\mathcal{A}_{P,k}^{(m)}$ as follows: we let $\mathcal{A}_{P,k}^{(m)} = (M \cup \{\$\}, \Delta, 1_M, \$)$, where the initial state is the unit 1_M of the monoid M, the final state is \$ and Δ consists of two types of transitions:

1. test and move transitions of the form $(m_1, \mathsf{act}_{m_1,m_2}? \cdot {\leftarrow_k}, m_2)$ where $m_1, m_2 \in M$ and

$$\mathsf{act}_{m_1,m_2} = \bigvee_{m' \mid m_2 = m' m_1} \mathsf{lift}_{\{k\}}(\varphi_{P\setminus\{k\}}^{(m')})$$

The intuition is as follows. We use the notations above. Assume that $\leftarrow_k$ moves from f_i to f_{i-1} and that, at f_i, the automaton has already computed $m_1 = \eta(t_{i+1}\cdots t_\ell) = \eta({\downarrow}f_\ell \setminus {\downarrow}f_i)$. We have seen that $t, f_i \models \mathsf{lift}_{\{k\}}(\varphi_{P\setminus\{k\}}^{(m')})$ if and only if $\eta(t_i) = m'$. Since the disjunction ranges over all m' with $m_2 = m' m_1$, we deduce that $m_2 = \eta(t_i \cdots t_\ell) = \eta({\downarrow}f_\ell \setminus {\downarrow}f_{i-1})$. Therefore, walking down the sequence $f_\ell, \ldots, f_1$, the automaton computes the values of η on the suffixes $t_i \cdots t_\ell$.

2. (accepting) test transitions of the form $(m_1, \{\mathsf{act}_{m_1,m} \wedge \neg\langle{\leftarrow_k}\rangle\}?, \$)$, where $m_1 \in M$.

To conclude, we let

$$\varphi_P^{(m)} = \Big(\bigvee_{a \in \Sigma \mid \eta(a) = m, \mathsf{loc}(a) \cap P = \emptyset} a \Big) \vee \Big(\bigvee_{k \in P, a \in \Sigma_k} a \wedge \langle \mathcal{A}_{P,k}^{(m)} \rangle \Big) \,.$$ ◀

4.2 Expressivity of sentences in LocPastPDL

We now generalize Theorem 4 from prime traces to all traces, which concludes the proof of Theorem 3. The precise statement of this generalization is as follows.

▶ **Theorem 7.** *Let $\eta\colon \mathrm{Tr}(\Sigma) \to M$ be a morphism to a finite monoid. For each $m \in M$, we can construct a* LocPastPDL *sentence $\Phi^{(m)}$ such that, for all traces $t \in \mathrm{Tr}(\Sigma)$, we have $\eta(t) = m$ iff $t \models \Phi^{(m)}$.*

As in Section 4.1, where we dealt with event formulas, we introduce a notion of *trace residuation*. If $t = (E, \leq, \lambda)$ is a trace, $i \in \mathcal{P}$ is a process and $P \subseteq \mathcal{P}$ is a set of processes, we let $\mathsf{res}_{i,P}(t)$, the *residue of process i with respect to P*, be the trace induced by the set of events $\downarrow E_i \setminus \downarrow E_P$. Here E_P denotes the set $\bigcup_{k\in P} E_k$, of all events involving a process in P. We record the following result, analogous to Lemma 5 (proof in Appendix A).

▶ **Lemma 8.** *Let t be a trace, $i \in \mathcal{P}$ a process and $P \subseteq \mathcal{P}$ a set of processes. Then we have*

$$\mathsf{res}_{i,P}(t) = \begin{cases} \varepsilon & \text{if } t \models \neg(\mathsf{L}_i \leq \mathsf{L}_i) \vee \bigvee_{j\in P}(\mathsf{L}_i \leq \mathsf{L}_j), \\ \mathsf{res}_{P'}(e) & \text{otherwise,} \end{cases}$$

where $e = \max E_i$ and $P' = \{j \in \mathcal{P} \mid t \models \bigvee_{k\in P} \mathsf{L}_{i,j} \leq \mathsf{L}_k\}$.

Proof of Theorem 7. The proof consists in identifying a particular, LocPastPDL-definable decomposition of a trace t as a product of prime traces, and using Theorem 4 to handle its factors.

Let $t = (E, \leq, \lambda)$ be a non-empty trace and let $e_1, \ldots, e_\ell$ be its maximal events. We choose a process $i_k \in \mathsf{loc}(e_k)$ for each maximal event. As maximal events are pairwise concurrent, the i_k are pairwise distinct. We let $t_1 = \downarrow e_1$ and, for $1 < k \leq \ell$, we let $Q_k = \{i_1, \ldots, i_{k-1}\}$ and $t_k = \mathsf{res}_{i_k,Q_k}(t)$. In particular, each t_k is a non-empty prime trace and $t = t_1 \cdot t_2 \cdots t_\ell$.

For each tuple $i_1, \ldots, i_\ell$ of pairwise distinct processes, the following LocPastPDL-sentence checks that a trace has ℓ maximal events located on processes $i_1, \ldots, i_\ell$ (we use i as an abbreviation for the event formula $\bigvee_{a\in\Sigma_i} a$):

$$\mathsf{MAX}_{i_1,\ldots,i_\ell} = \Big(\bigwedge_{1\leq k\leq \ell} \mathsf{EM}\, i_k\Big) \wedge \neg\,\mathsf{EM}\,\neg\Big(\bigvee_{1\leq k\leq \ell} i_k\Big) \wedge \neg\,\mathsf{EM}\,\Big(\bigvee_{1\leq k,k'\leq \ell, k\neq k'} i_k \wedge i_{k'}\Big)$$

Letting $P_1 = \emptyset$ we have $t_1 = \mathsf{res}_{P_1}(e_1)$. Using Lemma 8, we find subsets $P_k \subseteq \mathcal{P}$ such that $t_k = \mathsf{res}_{P_k}(e_k)$ for each $1 < k \leq \ell$. We note that Lemma 8 also justifies the following specification of the sets P_k: assuming that $t \models \mathsf{MAX}_{i_1,\ldots,i_\ell}$, these sets P_k are characterized by the sentence

$$\bigwedge_{1\leq k\leq \ell} \mathsf{RES}^{P_k}_{i_k,\{i_1,\ldots,i_{k-1}\}}$$

where

$$\mathsf{RES}^{P'}_{i,P} = \Big(\bigwedge_{j\in P'} \bigvee_{k\in P} \mathsf{L}_{i,j} \leq \mathsf{L}_k\Big) \wedge \Big(\bigwedge_{j\notin P'} \neg \bigvee_{k\in P} \mathsf{L}_{i,j} \leq \mathsf{L}_k\Big).$$

Finally, once the sequence $i_1, \ldots, i_\ell$ and the sets $P_1, \ldots, P_\ell$ are fixed, the equality $\eta(t) = m$ is checked by the sentence

$$\bigvee_{m=m_1\cdots m_\ell} \bigwedge_{1\leq k\leq \ell} \mathsf{EM}\left(i_k \wedge \mathsf{lift}_{P_k}(\varphi^{(m_k)})\right),$$

where the $\varphi^{(m_k)}$ $(1 \leq k \leq \ell)$ are given by Theorem 4.

To conclude the proof of the theorem, if $m \neq 1_M$, we let $\Phi^{(m)}$ be the sentence

$$\bigvee_{\substack{i_1,\dots,i_\ell \\ P_1,\dots,P_\ell}} \mathsf{MAX}_{i_1,\dots,i_\ell} \wedge \bigwedge_{1\le k\le \ell} \mathsf{RES}^{P_k}_{i_k,\{i_1,\dots,i_{k-1}\}} \wedge \bigvee_{m=m_1\cdots m_\ell} \bigwedge_{1\le k\le \ell} \mathsf{EM}\left(i_k \wedge \mathsf{lift}_{P_k}(\varphi^{(m_k)})\right). \quad (1)$$

Note that, the empty trace does not satisfy $\mathsf{MAX}_{i_1,\dots,i_\ell}$ for any tuple $(i_1,\dots,i_\ell)$ with $\ell > 0$, and hence it does not satisfy the formula in Equation (1), with $m = 1_M$. Therefore, we let $\Phi^{(1_M)}$ be the disjunction of $\neg\,\mathsf{EM}\,\top$ (which specifies the empty trace) and the formula in Equation (1), with $m = 1_M$. ◀

5 Asynchronous automata and local cascade products

In Section 6, we exploit the expressive completeness of LocPastPDL established above to give a Krohn-Rhodes style decomposition result for regular trace languages. Here, we first review the distributed model of asynchronous automata (Zielonka, [24]), seen both as acceptors of trace languages and as letter-to-letter trace transducers, and the related cascade product.

Asynchronous automata. work in a concurrent manner on traces over a distributed alphabet, say (Σ, loc). They have local states, for each process in $\mathcal{P}$, and their transitions on a letter $a \in \Sigma$ read and update only the states that are local to a process in $\mathsf{loc}(a)$. Formally, an *asynchronous automaton* A over (Σ, loc) is a tuple $(\{S_i\}_{i\in\mathcal{P}}, \{\delta_a\}_{a\in\Sigma}, s_{\mathrm{in}})$ where

- S_i is a finite non-empty set of *local i-states* for each process i;
- For $a \in \Sigma$, let $S_a = \prod_{i\in\mathsf{loc}(a)} S_i$ be called the set of *a-states*. Then $\delta_a \colon S_a \to S_a$ is a (deterministic and complete) *transition function on a-states*;
- $s_{\mathrm{in}} \in S$ (where $S = \prod_{i\in\mathcal{P}} S_i$ is called the set of *global states*) is the *initial global state*.

If s is a global state, we write s_a for its projection on S_a and s_{-a} for its projection on the remaining processes. It is convenient to write $s = (s_a, s_{-a})$.

For $a \in \Sigma$, let $\Delta_a \colon S \to S$ be the *global transition function* defined by $\Delta_a((s_a, s_{-a})) = (\delta_a(s_a), s_{-a})$. Composing these functions defines the global transition Δ_t of any trace $t \in \mathrm{Tr}(\Sigma)$: we let Δ_ε be the identity function and, if $t = t'a$, then $\Delta_t = \Delta_a \circ \Delta_{t'}$. We denote by $A(t)$ the global state reached when running A on t, that is, $A(t) = \Delta_t(s_{\mathrm{in}})$.

Zielonka's fundamental theorem [24] states that a trace language is recognizable if and only if it is accepted by some asynchronous automaton A, that is, if there exists a subset $S_{\mathrm{fin}} \subseteq S$ of *final global states* such that $L = \{t \in \mathrm{Tr}(\Sigma) \mid A(t) \in S_{\mathrm{fin}}\}$.

Asynchronous labeling functions. Asynchronous automata can be used not only as acceptors, as above, but also as devices to compute certain functions on traces: maps which, given a trace $t = (E, \le, \lambda)$, compute a trace with the same underlying poset structure $(E, \le)$, and with a richer labeling function.

That point of view, which was developed by the authors in [1, 2], generalizes the notion of sequential letter-to-letter word transducers, and is closely related to the *locally computable functions* defined in [19].

Formally, let (Σ, loc) be a distributed alphabet and Γ be a finite non-empty set. Then $\Sigma \times \Gamma$ is a distributed alphabet (over the same set $\mathcal{P}$ of processes as (Σ, loc)) for the location function given by $\mathsf{loc}(a, \gamma) = \mathsf{loc}(a)$ for every $(a, \gamma) \in \Sigma \times \Gamma$. A map $\theta \colon \mathrm{Tr}(\Sigma) \to \mathrm{Tr}(\Sigma \times \Gamma)$ is called a *Γ-labeling function* if, for each $t = (E, \le, \lambda) \in \mathrm{Tr}(\Sigma)$, we have $\theta(t) = (E, \le, (\lambda, \mu))$, *i.e.*, θ adds a new label $\mu(e) \in \Gamma$ to each event e in t.

▶ **Example 9.** Let F be a finite set of LocPastPDL event formulas and $\Gamma_F = \{0,1\}^F$. For each trace $t \in \mathrm{Tr}(\Sigma)$ and event e of t, we let $\mu_F(e)$ be the tuple of truth values of each $\varphi \in F$ at e. We then let θ^F be the Γ_F-labeling function which maps a trace $t = (E, \leq, \lambda) \in \mathrm{Tr}(\Sigma)$ to the trace $(E, \leq, (\lambda, \mu_F)) \in \mathrm{Tr}(\Sigma \times \Gamma_F)$.

An *asynchronous (letter-to-letter)* Γ*-transducer* over (Σ, loc) is a tuple $\hat{A} = (A, \{\mu_a\})$ where $A = (\{S_i\}, \{\delta_a\}, s_{\mathrm{in}})$ is an asynchronous automaton and each μ_a ($a \in \Sigma$) is a map $\mu_a \colon S_a \to \Gamma$. We associate with $\hat{A}$ the Γ-labeling function, also denoted by $\hat{A}$, from $\mathrm{Tr}(\Sigma)$ to $\mathrm{Tr}(\Sigma \times \Gamma)$, which maps $t = (E, \leq, \lambda)$ to $\hat{A}(t) = (E, \leq, (\lambda, \mu))$ in such a way that, for every event $e \in E$ with $\lambda(e) = a$ and $s = A(\Downarrow e)$, we have $\mu(e) = \mu_a(s_a)$. We say that $\hat{A}$ *computes* (or *implements*) the Γ-labeling function $\hat{A}$. We also say that an asynchronous automaton $A = (\{S_i\}, \{\delta_a\}, s_{\mathrm{in}})$ *computes* a Γ-labeling function θ if there are maps $\mu_a \colon S_a \to \Gamma$ such that $\theta = \hat{A}$, with $\hat{A} = (A, \{\mu_a\})$.

Notice that a Γ-labeling function is defined on every input trace, hence an asynchronous transducer admits a run on all traces and it does not use an acceptance condition.

▶ **Example 10.** Let $i \in \mathcal{P}$ be a process, let $\varphi_i = (\mathsf{Y}_i \leq \mathsf{Y}_i)$ be the LocPastPDL event formula which states that there is an event on process i in the strict past of the current event. With reference to Example 9, $\Gamma_F = \{0,1\}$ and the Γ_F-labeling function θ^F is computed by the following asynchronous transducer. For each process j, the set of j-states is $\{0,1\}$, and the global initial state has every process start in state 0. When the first event e occurs on process i, all processes in $\mathsf{loc}(e)$ switch to state 1: for every $a \in \Sigma_i$, δ_a is the constant map sending all states in S_a to $(1, \ldots, 1)$. This information is then propagated via synchronizing events: for every $b \in \Sigma \setminus \Sigma_i$, the map δ_b sends $(0, \ldots, 0)$ to itself and every other state to $(1, \ldots, 1)$. It is easy to add output functions $\{\mu_a\}_{a \in \Sigma}$ in order to compute θ^F.

Local cascade product. It turns out that the composition of labeling functions computed by asynchronous transducers, can also be computed by an asynchronous transducer. This asynchronous transducer is the result of the *local cascade product* operation defined below.

▶ **Definition 11.** *Let* $\hat{A} = (\{S_i\}, \{\delta_a\}, s_{\mathrm{in}}, \{\mu_a\})$ *be a* Γ*-labeling asynchronous transducer over* (Σ, loc)*, and let* $\hat{B} = (\{Q_i\}, \{\delta_{(a,\gamma)}\}, q_{\mathrm{in}}, \{\nu_{(a,\gamma)}\})$ *be a* Π*-labeling asynchronous transducer over* $(\Sigma \times \Gamma, \mathsf{loc})$*. We define the* local cascade product *of* $\hat{A}$ *and* $\hat{B}$ *to be the* $(\Gamma \times \Pi)$*-labeling asynchronous transducer* $\hat{A} \circ_\ell \hat{B} = (\{S_i \times Q_i\}, \{\nabla_a\}, (s_{\mathrm{in}}, q_{\mathrm{in}}), \{\tau_a\})$ *where* $\nabla_a((s_a, q_a)) = (\delta_a(s_a), \delta_{(a,\mu_a(s_a))}(q_a))$ *and* $\tau_a \colon S_a \times Q_a \to \Gamma \times \Pi$ *is defined by* $\tau_a((s_a, q_a)) = (\mu_a(s_a), \nu_{(a,\mu_a(s_a))}(q_a))$.

▶ Remark 12. It is directly verified that, with the notation of Definition 11, if $\hat{A}$ implements $f_A \colon \mathrm{Tr}(\Sigma) \to \mathrm{Tr}(\Sigma \times \Gamma)$ and $\hat{B}$ implements $f_B \colon \mathrm{Tr}(\Sigma \times \Gamma) \to \mathrm{Tr}(\Sigma \times \Gamma \times \Pi)$ then the local cascade product $\hat{A} \circ_\ell \hat{B}$ implements the composition $f_B \circ f_A \colon \mathrm{Tr}(\Sigma) \to \mathrm{Tr}(\Sigma \times \Gamma \times \Pi)$.

In the sequential case, that is, when $|\mathcal{P}| = 1$, the local cascade product coincides with the well-known operation of cascade product of sequential letter-to-letter transducers.

Slightly abusing language, we view a local cascade product also as an asynchronous automaton (forgetting the local labeling functions) and we can use it to accept trace languages as well. The celebrated theorem by Krohn and Rhodes [16] characterizes regular word languages as those accepted by cascade products of two simple kinds of automata:

- 2-state reset automata, where the transition function of each letter is either the identity function or constant;
- permutation automata, where each letter induces a permutation of the state set.

▶ **Theorem 13** (Krohn-Rhodes [16]). *Any regular word language is accepted by a cascade product of 2-state reset automata and permutation automata.*

In the setting of traces, we consider distributed analogues of reset and permutation automata. If $i \in \mathcal{P}$ is a process, a *2-state reset automaton localized at i* is an asynchronous automaton with two i-local states, where all other local state sets are singletons and the local transition induced by each letter is either the identity function, or a constant function. Similarly, a *permutation automaton localized at i* is an asynchronous automaton where each set of j-states ($j \neq i$) is a singleton and the local transition by any letter is a permutation.

Another important asynchronous automaton is Mukund and Sohoni's *gossip (asynchronous) automaton* $\mathcal{G}$, see [19]. Its main purpose is to compute the primary and secondary comparisons, as stated in the theorem below.

▶ **Theorem 14** (Mukund-Sohoni [19]). *Let* $\mathsf{Y} = \{\mathsf{Y}_i \leq \mathsf{Y}_j, \mathsf{Y}_{i,j} \leq \mathsf{Y}_k \mid i,j,k \in \mathcal{P}\}$ *be the set of all constant event formulas of* LocPastPDL *and let* θ^{Y} *be the corresponding labeling function. The gossip automaton* $\mathcal{G}$ *computes* θ^{Y}.

We say that a local cascade product $\hat{\mathcal{G}} \circ_\ell \hat{B}$ is *restricted* (or θ^{Y}*-restricted*) if the labeling function computed by $\hat{\mathcal{G}}$ is θ^{Y}, *i.e.*, the information passed to $\hat{B}$ by $\hat{\mathcal{G}}$ is restricted to the truth values of the event formulas in Y.

▶ Remark 15. The gossip automaton $\mathcal{G}$ also computes the truth values of the constant trace sentences $\mathsf{L} = \{\mathsf{L}_i \leq \mathsf{L}_j, \mathsf{L}_{i,j} \leq \mathsf{L}_k \mid i,j,k \in \mathcal{P}\}$, this time *globally*. More precisely, if S is the global state set of $\mathcal{G}$, then there is a map $\zeta\colon S \to \{0,1\}^{\mathsf{L}}$ such that, for every trace t, and sentence $\Phi \in \mathsf{L}$, $t \models \Phi$ if and only if the Φ-component of $\zeta(\mathcal{G}(t))$ is 1.

6 Cascade decomposition

The following is the main result of this section.

▶ **Theorem 16.** *Let φ be a* LocPastPDL *event formula and θ^φ (for $\theta^{\{\varphi\}}$) be the corresponding $\{0,1\}$-labeling function. One can construct a restricted cascade product of the gossip automaton followed by a local cascade product of localized reset and permutation automata, which computes θ^φ.*

Before we prove Theorem 16, we establish an important corollary.

▶ **Corollary 17.** *Any regular trace language is accepted by a restricted local cascade product of the gossip automaton and a local cascade product of localized reset automata and localized permutation automata.*

Proof. By Theorem 3, any regular trace language L is defined by a sentence Φ in LocPastPDL. If Φ is of the form $\mathsf{L}_i \leq \mathsf{L}_j$ or $\mathsf{L}_{i,j} \leq \mathsf{L}_k$, then L is accepted by the gossip automaton, by Remark 15. The case where Φ is a non-trivial boolean combination is easily handled, and we are left with sentences of the form $\mathsf{EM}\,\varphi$.

If $i \in \mathcal{P}$ is a process, let $\mathsf{EM}_i\,\varphi$ be the sentence which expresses that a trace has at least one i-event, and that its maximum i-event satisfies φ. Then $\mathsf{EM}\,\varphi$ is equivalent to the disjunction $\bigvee_{i\in\mathcal{P}} \Big(\mathsf{EM}_i\,\varphi \wedge \neg(\bigvee_{j\in\mathcal{P}} \mathsf{L}_i < \mathsf{L}_j)\Big)$, where $\mathsf{L}_i < \mathsf{L}_j = (\mathsf{L}_i \leq \mathsf{L}_j) \wedge \neg(\mathsf{L}_j \leq \mathsf{L}_i)$, so we only need to deal with sentences of the form $\mathsf{EM}_i\,\varphi$.

By Theorem 16, the labeling function θ^φ is computed by an asynchronous transducer A_φ of the required local cascade form. Let B be the localized reset automaton with local i-states $\{q_0, q_1\}$ (and other local state sets singletons), initial state q_0, on alphabet $\Sigma \times \{0,1\}$, with

the following transitions. The transition induced by a letter $(a, 1)$ such that $i \in \mathsf{loc}(a)$ is a constant map to q_1. Other transitions labeled $(a, 0)$ with $i \in \mathsf{loc}(a)$ are the constant map to q_0. Then $A_\varphi \circ_\ell B$ recognizes $\mathsf{EM}_i\, \varphi$ when the global final state of the B component is q_1. Notice that B only needs to check if the maximal i-event of t satisfies φ, an information which is already added to the label of this event by A_φ (as 0 or 1). ◀

We now move towards the proof of Theorem 16. We first associate with each local path automaton a regular word language over a decorated alphabet. Specifically, let $\mathcal{A}$ be an i-local path automaton and let F be the set of event formulas in its test transitions. Recall that, if $t \in \mathrm{Tr}(\Sigma)$ and e, f are events in t, we have $t, e, f \models \mathcal{A}$ if there is an accepting run $q_0 \xrightarrow{\alpha_1} q_1 \cdots q_{n-1} \xrightarrow{\alpha_n} q_n$ and a sequence of events $e = e_0, e_1, \ldots, e_{n-1}, e_n = f$ such that for all $0 \leq m < n$ we have $t, e_m, e_{m+1} \models \alpha_m$. Checking whether $t, e_m, e_{m+1} \models \varphi?$ for some $\varphi \in F$ is done by a simple inspection of the label of event $e_m = e_{m+1}$ in $\theta^F(t) \in \mathrm{Tr}(\Sigma \times \Gamma_F)$. Observe also that, since $\mathcal{A}$ is i-local, all the e_m are i-events. This leads to the definition of a word language $\mathcal{L}_F(\mathcal{A})$ over the alphabet $\Sigma_i \times \Gamma_F$. Each word w in $\mathcal{L}_F(\mathcal{A})$ is induced by a trace t and a pair of events e, f such that $t, e, f \models \mathcal{A}$, and consists of the sequence of labels in $\theta^F(t)$ of the i-events from f to e.

$$\mathcal{L}_F(\mathcal{A}) = \{\theta^F(t) \cap E_i \cap {\downarrow}e \cap {\uparrow}f \mid t, e, f \models \mathcal{A}\} \subseteq (\Sigma_i \times \Gamma_F)^* .$$

▶ **Lemma 18.** *Let $\mathcal{A}$ be an i-local path automaton and let F be the set of event formulas in its test transitions. Then $\mathcal{L}_F(\mathcal{A})$ is a regular language.*

Proof. The automaton $\mathcal{A}$ accepts a regular language over the alphabet $\{\varphi? \mid \varphi \in F\} \cup \{\leftarrow_i\}$. Processing a letter from the alphabet translates to a move to a different event in the trace (see the semantics of path automata recalled above) if that letter is $\leftarrow_i$, but not if it is of the form $\varphi?$. To smooth out this difference, we modify $\mathcal{A}$ to an automaton $\mathcal{B}$ with the same semantics (in the sense that $t, e, f \models \mathcal{A}$ if and only if $t, e, f \models \mathcal{B}$), where transitions to an accepting state have labels of the form $\psi?$ and all other transitions have a label of the form $\psi'?{\leftarrow_i}$, where ψ, ψ' are conjunctions of formulas in F (we talk of *test-and-move* transitions).

Let Q be the set of states of $\mathcal{A}$ and let $q_a \notin Q$ be a new state. The set of states of $\mathcal{B}$ is $Q' = Q \cup \{q_a\}$, $\mathcal{B}$ has the same initial states as $\mathcal{A}$, and q_a is the only accepting state of $\mathcal{B}$. The transitions of $\mathcal{B}$ are as follows.

1. Let $q, q' \in Q$ and let ψ be a conjunction of formulas in F. $\mathcal{B}$ has a test-and-move transition from q to q' labeled $\psi? \cdot \leftarrow_i$ in $\mathcal{B}$ if there is a path from q to q' in $\mathcal{A}$ starting with a sequence of test transitions using exactly all the conjuncts of ψ and ending with a move transition (labeled $\leftarrow_i$).
2. Let $q \in Q$ and let ψ be a conjunction of formulas in F. $\mathcal{B}$ has a test transition from q to q_a labeled $\psi?$ if there is a path in $\mathcal{A}$ from q to some accepting state q' of $\mathcal{A}$ consisting of test transitions using exactly all the conjuncts of ψ.

It is not difficult to see that $\mathcal{A}$ and $\mathcal{B}$ have the same semantics. For each $\varphi \in F$, let Δ_φ be the set of letters $(a, \gamma) \in \Sigma_i \times \Gamma_F$ such that the φ-component of γ is 1. We now modify $\mathcal{B}$ into a new automaton $\mathcal{B}'$ by changing the labels of transitions: for each edge labeled by $\bigwedge_{j=1}^k \varphi_j?{\leftarrow_i}$ (resp. $\bigwedge_{j=1}^k \varphi_j?$), replace the label with $\bigcap_{j=1}^k \Delta_{\varphi_j}$. The automaton $\mathcal{B}'$, over $\Sigma_i \times \Gamma_F$ is easily seen to accept the reverse language of $\mathcal{L}_F(\mathcal{A})$, so $\mathcal{L}_F(\mathcal{A})$ itself is regular. ◀

We can finally prove Theorem 16, the last missing element of this paper.

Proof of Theorem 16. The proof is by structural induction on the LocPastPDL event formula φ. If $\varphi = a$ ($a \in \Sigma$), A_a is the asynchronous transducer where each set of local states is a singleton. The labeling function is defined by $\mu_b(s) = 1$ if $b = a$ and 0 otherwise. If $\varphi = \mathsf{Y}_i \leq \mathsf{Y}_k$ or $\varphi = \mathsf{Y}_{i,j} \leq \mathsf{Y}_k$, Theorem 14 shows that we can use the gossip automaton as A_φ. Boolean combination of event formulae are easily handled.

The case where $\varphi = \langle \mathcal{A} \rangle$, for some i-local path automaton $\mathcal{A}$, is non-trivial. Let F be the set of event formulas in the test transitions of $\mathcal{A}$. By induction hypothesis, for each $\psi \in F$, we have an asynchronous transducer A_ψ in the required local cascade form which computes θ^ψ. By the usual direct product construction, which can be subsumed by a local cascade product (factorizing the gossip automaton), we then have an asynchronous transducer A_F in the required form which computes θ^F. We then construct A_φ in the form of a local cascade product $A_F \circ_\ell B$ for an appropriate asynchronous transducer B on alphabet $\Sigma \times \Gamma_F$.

Let $\mathcal{L}_F(A)$ be the language (over alphabet $\Sigma_i \times \Gamma_F$) defined above, which is regular by Lemma 18, and let C be an automaton accepting $(\Sigma_i \times \Gamma_F)^* \cdot \mathcal{L}_F(A)$. By Krohn-Rhodes's theorem (Theorem 13 above), C can be chosen to be a cascade product of 2-state reset automata and permutation automata. Let us *localize* each of these automata at process i, by adding singleton local state sets for each process $j \neq i$. The resulting local cascade product (of localized reset and permutation asynchronous transducers) allows us to check whether an i-local state is final in C or not. It is easily verified that a labeling function can then be imposed on B, by which $A_F \circ_\ell B$ computes θ^φ.

This completes the proof of Theorem 16. ◀

7 Conclusion

We have shown that LocPastPDL is expressively complete. Recall that a basic trace formula of LocPastPDL is either of the form $\mathsf{EM}\,\varphi$ or a constant comparision formula such as $\mathsf{L}_i \leq \mathsf{L}_j$ or $\mathsf{L}_{i,j} \leq \mathsf{L}_k$. We could instead use basic trace formula $\mathsf{EM}_i\,\varphi$ which asserts that the maximum i-event exists and satisfies the event formula φ. It follows from the results in [3] that boolean combinations of $\mathsf{EM}_i\,\varphi$ suffice to arrive at an expressively complete logic. Note that our expressive-completeness proof of LocPastPDL is direct and self-contained. In view of this, it would be interesting to directly express $\mathsf{L}_i \leq \mathsf{L}_j$ and $\mathsf{L}_{i,j} \leq \mathsf{L}_k$ using only basic formulas of the form $\mathsf{EM}_i\,\varphi$. Another exciting question concerns the necessity of the primary and secondary event comparision formulas $\mathsf{Y}_i \leq \mathsf{Y}_j$ and $\mathsf{Y}_{i,j} \leq \mathsf{Y}_k$ for the expressive completeness result. This is also intimately related to the necessity of the gossip automaton in our distributed Krohn-Rhodes theorem. It would be also interesting to identify a natural fragment of LocPastPDL which matches first-order logic in expressive power, and also extend the results in this work to infinite traces.

References

1 Bharat Adsul, Paul Gastin, Saptarshi Sarkar, and Pascal Weil. Wreath/cascade products and related decomposition results for the concurrent setting of Mazurkiewicz traces. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 19:1–19:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

2 Bharat Adsul, Paul Gastin, Saptarshi Sarkar, and Pascal Weil. Asynchronous wreath product and cascade decompositions for concurrent behaviours. *Log. Methods Comput. Sci.*, 18, 2022.

3 Bharat Adsul and Milind A. Sohoni. Local normal forms for logics over traces. In Manindra Agrawal and Anil Seth, editors, *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science, 22nd Conference Kanpur, India, December 12-14, 2002, Proceedings*, volume 2556 of *Lecture Notes in Computer Science*, pages 47–58. Springer, 2002.

4 Benedikt Bollig, Marie Fortin, and Paul Gastin. Communicating finite-state machines, first-order logic, and star-free propositional dynamic logic. *J. Comput. Syst. Sci.*, 115:22–53, 2021. doi:10.1016/j.jcss.2020.06.006.

5 Benedikt Bollig and Paul Gastin. Non-sequential theory of distributed systems. *CoRR*, abs/1904.06942, 2019. arXiv:1904.06942.

6 Benedikt Bollig, Dietrich Kuske, and Ingmar Meinecke. Propositional dynamic logic for message-passing systems. *Log. Methods Comput. Sci.*, 6(3), 2010. arXiv:1007.4764.

7 Giuseppe De Giacomo and Moshe Y Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

8 Volker Diekert and Paul Gastin. LTL is expressively complete for Mazurkiewicz traces. *Journal of Computer and System Sciences*, 64(2):396–418, March 2002.

9 Volker Diekert and Paul Gastin. Pure future local temporal logics are expressively complete for mazurkiewicz traces. *Information and Computation*, 204(11):1597–1619, November 2006.

10 Volker Diekert and Paul Gastin. First-order definable languages. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata: History and Perspectives*, volume 2 of *Texts in Logic and Games*, pages 261–306. Amsterdam University Press, 2008.

11 Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces*. World Scientific, 1995. doi:10.1142/2563.

12 Werner Ebinger and Anca Muscholl. Logical definability on infinite traces. *Theor. Comput. Sci.*, 154(1):67–84, 1996.

13 Michael J Fischer and Richard E Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

14 Stefan Göller, Markus Lohrey, and Carsten Lutz. PDL with intersection and converse: satisfiability and infinite-state model checking. *J. Symb. Log.*, 74(1):279–314, 2009. doi:10.2178/jsl/1231082313.

15 Giovanna Guaiana, Antonio Restivo, and Sergio Salemi. Star-free trace languages. *Theor. Comput. Sci.*, 97(2):301–311, 1992.

16 Kenneth Krohn and John Rhodes. Algebraic theory of machines I. Prime decomposition theorem for finite semigroups and machines. *Transactions of The American Mathematical Society*, 116, April 1965. doi:10.2307/1994127.

17 Manfred Kufleitner. Polynomials, fragments of temporal logic and the variety DA over traces. *Theoretical Computer Science*, 376:89–100, 2007. Special issue DLT 2006.

18 Roy Mennicke. Propositional dynamic logic with converse and repeat for message-passing systems. *Log. Methods Comput. Sci.*, 9(2), 2013. doi:10.2168/LMCS-9(2:12)2013.

19 Madhavan Mukund and Milind A. Sohoni. Keeping track of the latest gossip in a distributed system. *Distributed Comput.*, 10(3):137–148, 1997. doi:10.1007/s004460050031.

20 Howard Straubing. *Finite automata, formal logic, and circuit complexity*. Birkhaüser Verlag, Basel, Switzerland, 1994.

21 P.S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for Mazurkiewicz traces. *Information and Computation*, 179(2):230–249, December 2002.

22 Wolfgang Thomas. On logical definability of trace languages. In V. Diekert, editor, *Proceedings of a workshop of the ESPRIT Basic Research Action No 3166: Algebraic and Syntactic Methods in Computer Science (ASMICS), Kochel am See, Bavaria, FRG (1989)*, Report TUM-I9002, Technical University of Munich, pages 172–182, 1990.

23 Nicolas Troquard and Philippe Balbiani. Propositional Dynamic Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2019 edition, 2019. URL: https://plato.stanford.edu/archives/spr2019/entries/logic-dynamic/.

24 Wieslaw Zielonka. Notes on finite asynchronous automata. *RAIRO Theor. Informatics Appl.*, 21(2):99–135, 1987. doi:10.1051/ita/1987210200991.

A Appendix for Section 4

▶ **Lemma 6** (Lifting lemma). *Let $\varphi \in \mathsf{PastPDL}$ be an event formula and $P \subseteq \mathcal{P}$ be a set of processes. We can construct an event formula $\mathsf{lift}_P(\varphi) \in \mathsf{PastPDL}$ such that, for all traces $t = (E, \leq, \lambda)$ and events $e \in E$ in t, we have $\mathsf{res}_P(e), e \models \varphi$ if and only if $t, e \models \mathsf{lift}_P(\varphi)$. Moreover, if $\varphi \in \mathsf{LocPastPDL}$ then $\mathsf{lift}_P(\varphi) \in \mathsf{LocPastPDL}$.*

Proof. The construction is by structural induction on φ. We first let

$$\mathsf{lift}_P(a) = a \text{ for each } a \in \Sigma$$
$$\mathsf{lift}_P(\mathsf{Y}_i \leq \mathsf{Y}_j) = (\mathsf{Y}_i \leq \mathsf{Y}_j) \wedge \neg \bigvee_{\ell \in P} (\mathsf{Y}_i \leq \mathsf{Y}_\ell) \vee (\mathsf{Y}_j \leq \mathsf{Y}_\ell)$$
$$\mathsf{lift}_P(\mathsf{Y}_{i,j} \leq \mathsf{Y}_k) = (\mathsf{Y}_{i,j} \leq \mathsf{Y}_k) \wedge \neg \bigvee_{\ell \in P} (\mathsf{Y}_{i,j} \leq \mathsf{Y}_\ell) \vee (\mathsf{Y}_k \leq \mathsf{Y}_\ell)$$

The announced statement is easily verified for these atomic formulas. Similarly, boolean combinations of formulas are handled by letting $\mathsf{lift}_P(\varphi \vee \psi) = \mathsf{lift}_P(\varphi) \vee \mathsf{lift}_P(\psi)$ and $\mathsf{lift}_P(\neg\varphi) = \neg\mathsf{lift}_P(\varphi)$.

The last, and more interesting case, is that where $\varphi = \langle\mathcal{A}\rangle$, for a past path automaton $\mathcal{A} = (Q, \Delta, I, F)$. We let $\mathsf{lift}_P(\langle\mathcal{A}\rangle) = \langle\mathcal{A}_P\rangle$, where $\mathcal{A}_P = (Q', \Delta', I', F')$ is the path automaton defined as follows:

- $Q' = Q \times 2^{\mathcal{P}}$, $I' = I \times \{P\}$ and $F' = F \times 2^{\mathcal{P}}$,
- for each test transition $(q_1, \varphi?, q_2) \in \Delta$ of $\mathcal{A}$ and each set $P_1 \subseteq \mathcal{P}$, we define the test transition $((q_1, P_1), \mathsf{lift}_{P_1}(\varphi)?, (q_2, P_1))$ in $\mathcal{A}_P$,
- for each move transition $(q_1, \leftarrow_k, q_2) \in \Delta$ of $\mathcal{A}$ and each sets $P_1, P_2 \subseteq \mathcal{P}$ with $P_1 \subseteq P_2$, we define a *test and move*[2] transition $((q_1, P_1), \mathsf{change}_{k,P_1,P_2}? \cdot \leftarrow_k, (q_2, P_2))$ in $\mathcal{A}_P$ where

$$\mathsf{change}_{k,P_1,P_2} = \Big(\neg \bigvee_{i \in P_1} \mathsf{Y}_k \leq \mathsf{Y}_i\Big) \wedge \Big(\bigwedge_{j \in P_2 \setminus P_1} \bigvee_{i \in P_1} \mathsf{Y}_{k,j} \leq \mathsf{Y}_i\Big) \wedge \Big(\bigwedge_{j \notin P_2} \neg \bigvee_{i \in P_1} \mathsf{Y}_{k,j} \leq \mathsf{Y}_i\Big)$$

  The formula above characterises the change of context when moving from an event e on process k to the previous event e_k on process k. It is based on Lemma 5. The first conjunct says that $\mathsf{res}_{P_1}(e) \cap {\downarrow}e_k$ is nonempty. The remaining conjuncts characterises the set P_2 such that $\mathsf{res}_{P_1}(e) \cap {\downarrow}e_k = \mathsf{res}_{P_2}(e_k)$.

Observe that $\mathcal{A}_P$ is again a *past* automaton and all reachable states (q, P') in $\mathcal{A}_P$ satisfy $P \subseteq P'$. Moreover, if $\mathcal{A}$ is k-local then so is $\mathcal{A}_P$.

We claim that this construction is correct, *i.e.*, $\mathsf{res}_P(e), e \models \langle\mathcal{A}\rangle$ if and only if $t, e \models \langle\mathcal{A}_P\rangle$.

Let us first assume that $\mathsf{res}_P(e), e \models \langle\mathcal{A}\rangle$. There is an accepting run $q_0 \xrightarrow{\alpha_1} q_1 \cdots q_{n-1} \xrightarrow{\alpha_n} q_n$ of $\mathcal{A}$ and a sequence of events $e = e_0, e_1, \ldots, e_{n-1}, e_n$ such that for all $1 \leq m \leq n$ we have $\mathsf{res}_P(e), e_{m-1}, e_m \models \alpha_m$. We construct inductively

- a sequence $P_0, P_1, \ldots, P_n \subseteq \mathcal{P}$ so that ${\downarrow}e_m \cap \mathsf{res}_P(e) = \mathsf{res}_{P_m}(e_m)$ for all $0 \leq m \leq n$,
- an accepting run $(q_0, P_0) \xrightarrow{\beta_1} (q_1, P_1) \cdots (q_{n-1}, P_{n-1}) \xrightarrow{\beta_n} (q_n, P_n)$ of $\mathcal{A}_P$ such that $t, e_{m-1}, e_m \models \beta_m$ for all $1 \leq m \leq n$.

We start with $P_0 = P$ so that ${\downarrow}e_0 \cap \mathsf{res}_P(e) = \mathsf{res}_{P_0}(e_0)$ and (q_0, P_0) is initial in $\mathcal{A}_P$. Now, let $0 < m \leq n$ and assume that we have constructed the sequence of sets and the run up to $m - 1$. There are two cases.

[2] Formally, in order to comply with the definition of a path automaton, we should split each test and move transition into a test transition followed by a move transition with a new intermediary state in-between.

1. If $\alpha_m = \psi?$ is a test, we have $e_m = e_{m-1}$ and $\mathsf{res}_P(e), e_{m-1} \models \psi$. Since ψ is a past formula and $\downarrow e_{m-1} \cap \mathsf{res}_P(e) = \mathsf{res}_{P_{m-1}}(e_{m-1})$, we deduce that $\mathsf{res}_{P_{m-1}}(e_{m-1}), e_{m-1} \models \psi$. By our induction hypothesis, we then have $t, e_{m-1} \models \mathsf{lift}_{P_{m-1}}(\psi)$, and we let $\beta_m = \mathsf{lift}_{P_{m-1}}(\psi)?$ and $P_m = P_{m-1}$. With this definition, the required conditions are satisfied: $\downarrow e_m \cap \mathsf{res}_P(e) = \mathsf{res}_{P_m}(e_m)$, $((q_{m-1}, P_{m-1}), \beta_m, (q_m, P_m))$ is a transition in $\mathcal{A}_P$, and $t, e_{m-1}, e_m \models \beta_m$.
2. If $\alpha_m = \leftarrow_k$ is a left move, we have $e_{m-1}, e_m \in E_k$ and e_m is the predecessor of e_{m-1} on process k. In particular, e_m is the maximal event in $\Downarrow e_{m-1} \cap E_k$. We apply Lemma 5: first, $\downarrow e_m \cap \mathsf{res}_{P_{m-1}}(e_{m-1}) = \downarrow e_m \cap \mathsf{res}_P(e)$ is nonempty since it contains e_m; it follows that $t, e_{m-1} \models \neg \bigvee_{i \in P_{m-1}} \mathsf{Y}_k \leq \mathsf{Y}_i$. Let $P_m = P_{m-1} \cup \{j \in \mathcal{P} \mid t, e_{m-1} \models \mathsf{Y}_{k,j} \leq \mathsf{Y}_i \text{ for some } i \in P_{m-1}\}$. By Lemma 5, we have $\downarrow e_m \cap \mathsf{res}_P(e) = \downarrow e_m \cap \mathsf{res}_{P_{m-1}}(e) = \mathsf{res}_{P_m}(e_m)$. By definition of P_m, we get $t, e_{m-1} \models \mathsf{change}_{k,P_{m-1},P_m}$. We then let $\beta_m = \mathsf{change}_{k,P_{m-1},P_m}? \cdot \leftarrow_k$ so that $((q_{m-1}, P_{m-1}), \beta_m, (q_m, P_m))$ is a transition in $\mathcal{A}_P$, and $t, e_{m-1}, e_m \models \beta_m$.

Using the constructed run in $\mathcal{A}_P$ and the same sequence of events $e = e_0, e_1, \ldots, e_{n-1}, e_n$, we find that $t, e \models \langle \mathcal{A}_P \rangle$.

Conversely, assume that $t, e \models \mathsf{lift}_P(\langle \mathcal{A} \rangle) = \langle \mathcal{A}_P \rangle$. There is an accepting run $(q_0, P_0) \xrightarrow{\beta_1} (q_1, P_1) \cdots (q_{n-1}, P_{n-1}) \xrightarrow{\beta_n} (q_n, P_n)$ of $\mathcal{A}_P$ and a sequence of events $e = e_0, e_1, \ldots, e_{n-1}, e_n$ such that $t, e_{m-1}, e_m \models \beta_m$ for all $1 \leq m \leq n$. We show by induction that $\downarrow e_m \cap \mathsf{res}_P(e) = \mathsf{res}_{P_m}(e_m)$ for all $0 \leq m \leq n$. We construct simultaneously a sequence $\alpha_1, \ldots, \alpha_n$ such that $q_0 \xrightarrow{\alpha_1} q_1 \cdots q_{n-1} \xrightarrow{\alpha_n} q_n$ is an accepting run of $\mathcal{A}$ and $\mathsf{res}_P(e), e_{m-1}, e_m \models \alpha_m$ for all $1 \leq m \leq n$.

Since (q_0, P_0) is initial in $\mathcal{A}_P$, we have $P_0 = P$. Using $e_0 = e$, we get $\downarrow e_0 \cap \mathsf{res}_P(e) = \mathsf{res}_{P_0}(e_0)$. Now, assume that our properties hold up to $m - 1$. There are two cases.

1. If $\beta_m = \mathsf{lift}_{P_{m-1}}(\psi)?$ is a test, then $P_m = P_{m-1}$ and $e_m = e_{m-1}$. In particular, $\downarrow e_m \cap \mathsf{res}_P(e) = \mathsf{res}_{P_m}(e_m)$. Let $\alpha_m = \psi?$. By definition of $\mathcal{A}_P$, we know that (q_{m-1}, α_m, q_m) is a transition of $\mathcal{A}$. From $t, e_{m-1}, e_m \models \beta_m = \mathsf{lift}_{P_{m-1}}(\psi)?$, we get $t, e_{m-1} \models \mathsf{lift}_{P_{m-1}}(\psi)$ and, by the induction hypothesis, we obtain $\mathsf{res}_{P_{m-1}}(e_{m-1}), e_{m-1} \models \psi$. Since ψ is a past formula and $\mathsf{res}_{P_{m-1}}(e_{m-1}) = \downarrow e_{m-1} \cap \mathsf{res}_P(e)$, we deduce that $\mathsf{res}_P(e), e_{m-1} \models \psi$ and finally $\mathsf{res}_P(e), e_{m-1}, e_m \models \alpha_m = \psi?$.
2. If $\beta_m = \mathsf{change}_{k,P_{m-1},P_m}? \cdot \leftarrow_k$ is a test-and-move, we let $\alpha_m = \leftarrow_k$. Then $t, e_{m-1} \models \mathsf{change}_{k,P_{m-1},P_m}$ and (q_{m-1}, α_m, q_m) is a transition of $\mathcal{A}$. Moreover, $t, e_{m-1}, e_m \models \leftarrow_k$. Using the fact that $t, e_{m-1} \models \neg \bigvee_{i \in P_{m-1}} \mathsf{Y}_k \leq \mathsf{Y}_i$, Lemma 5 shows that $e_m \in \mathsf{res}_{P_{m-1}}(e_{m-1}) = \downarrow e_{m-1} \cap \mathsf{res}_P(e)$. Therefore, $\mathsf{res}_P(e), e_{m-1}, e_m \models \leftarrow_k$. Finally, using Lemma 5 again and the fact that $t, e_{m-1} \models \mathsf{change}_{k,P_{m-1},P_m}$, we get $\mathsf{res}_{P_m}(e_m) = \downarrow e_m \cap \mathsf{res}_{P_{m-1}}(e_{m-1}) = \downarrow e_m \cap \mathsf{res}_P(e)$.

Thus $\mathsf{res}_P(e), e \models \langle \mathcal{A} \rangle$, and this concludes the proof. ◀

▶ **Lemma 8.** *Let t be a trace, $i \in \mathcal{P}$ a process and $P \subseteq \mathcal{P}$ a set of processes. Then we have*

$$\mathsf{res}_{i,P}(t) = \begin{cases} \varepsilon & \text{if } t \models \neg(\mathsf{L}_i \leq \mathsf{L}_i) \vee \bigvee_{j \in P}(\mathsf{L}_i \leq \mathsf{L}_j), \\ \mathsf{res}_{P'}(e) & \text{otherwise,} \end{cases}$$

where $e = \max E_i$ and $P' = \{j \in \mathcal{P} \mid t \models \bigvee_{k \in P} \mathsf{L}_{i,j} \leq \mathsf{L}_k\}$.

Proof. Observe that $\mathsf{res}_{i,P}(t)$ is the empty trace if E_i is empty or if the maximal i-event is below a j-event for some $j \in P$. The first condition is exactly captured by $\neg(\mathsf{L}_i \leq \mathsf{L}_i)$, and the second one by $\bigvee_{j \in P}(\mathsf{L}_i \leq \mathsf{L}_j)$.

Let us now assume that $\mathsf{res}_{i,P}(t)$ is not the empty trace. Then $e = \max E_i$ exists. Moreover, by definition, $\mathsf{res}_{i,P}(t) = {\downarrow}e \setminus {\downarrow}E_P$ and $\mathsf{res}_{P'}(e) = {\downarrow}e \setminus \bigcup_{j\in P'} {\downarrow}e_j$ where e_j is the maximal event in $E_j \cap {\Downarrow}e$, if it exists.

Let f be an event in $\mathsf{res}_{i,P}(t)$. Then $f \leq e$. Suppose that $f \leq e_j$ for some $j \in P'$. By definition of P', there exists $k \in P$ such that $t \models \mathsf{L}_{i,j} \leq \mathsf{L}_k$. Then

$$f \leq e_j = \max(E_j \cap {\Downarrow}e) \leq \max(E_j \cap {\downarrow}e) \leq \max(E_k).$$

In particular, $f \in {\downarrow}E_P$, a contradiction since $f \in \mathsf{res}_{i,P}(t) = {\downarrow}e \setminus {\downarrow}E_P$. Therefore $\mathsf{res}_{i,P}(t)$ is contained in ${\downarrow}e \setminus \bigcup_{j\in P'} {\downarrow}e_j = \mathsf{res}_{P'}(e)$.

Conversely, suppose that $f \in \mathsf{res}_{P'}(e)$. Then, again, $f \leq e$. Suppose that $f \in {\downarrow}E_P$, i.e., $f \leq e' = \max(E_k)$ for some $k \in P$. If $e' < e$, then $f \leq e' = \max(E_k \cap {\Downarrow}e) = e_k$. Also, $E_k \subseteq {\downarrow}e$, so $t \models \mathsf{L}_{i,k} \leq \mathsf{L}_k$ and hence $k \in P'$, which is impossible since $f \in \mathsf{res}_{P'}(e)$.

We cannot have $e \leq e'$ either, since $t \not\models (\mathsf{L}_i \leq \mathsf{L}_k)$. Therefore e and e' are concurrent events. Let g be a maximal event in ${\uparrow}f \cap {\downarrow}e \cap {\downarrow}e'$. There exists $j \in \mathsf{loc}(g)$ such that $e_j = g = \max(E_j \cap {\downarrow}e)$. Again this implies that $j \in P'$ and $f \in \bigcup_{j\in P'} {\downarrow}e_j$, a contradiction. It follows that $\mathsf{res}_{P'}(e)$ is contained in $\mathsf{res}_{i,P}(e)$, which concludes the proof. ◀