Decentralised Runtime Verification of Timed Regular Expressions

Victor Roussanaly
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

Yliès Falcone
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

Abstract

Ensuring the correctness of distributed cyber-physical systems can be done at runtime by monitoring properties over their behaviour. In a decentralised setting, such behaviour consists of multiple local traces, each offering an incomplete view of the system events to the local monitors, as opposed to the standard centralised setting with a unique global trace. We introduce the first monitoring framework for timed properties described by timed regular expressions over a distributed network of monitors. First, we define functions to rewrite expressions according to partial knowledge for both the centralised and decentralised cases. Then, we define decentralised algorithms for monitors to evaluate properties using these functions, as well as proofs of soundness and eventual completeness of said algorithms. Finally, we implement and evaluate our framework on synthetic timed regular expressions, giving insights on the cost of the centralised and decentralised settings and when to best use each of them.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Regular languages; Theory of computation → Rewrite systems; Theory of computation → Automata over infinite objects; Computer systems organization → Real-time system specification

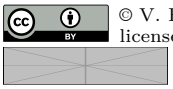
Keywords and phrases Timed expressions, Timed properties, Monitoring, Runtime verification, Decentralized systems, Asynchronous communication

Digital Object Identifier 10.4230/LIPIcs...12

Funding *Victor Roussanaly*: [H2020- ECSEL-2018-IA call – Grant Agreement number 826276 (CPS4EU)]
Yliès Falcone: [H2020- ECSEL-2018-IA call – Grant Agreement number 826276 (CPS4EU), Région Auvergne-Rhône- Alpes - "Pack Ambition Recherche" programme, French ANR project ANR-20-CE39-0009 (SEVERITAS), LabEx PERSYVAL-Lab (ANR- 11-LABX-0025-01)]

Introduction

Modern systems tend to be more distributed and interconnected. Automated verification methods are required to ensure those systems behave as they should. Moreover, their interactions with their environment are getting increasingly unpredictable, which tends to hinder static verification methods such as model checking, as they require a model of the verified system and do not scale well. On the other hand, runtime verification [12, 4, 13] requires no model. In this area, several monitoring methods detect if a system violates its given specification based on events observed at runtime. In order to express finer properties over more complex behaviour, several algorithms for monitoring properties based on real continuous time have been proposed in [16] and [15]. However, these algorithms assume a central observation point in the system, which might be less robust to an architecture change, more vulnerable to outside attacks, or less compatible with the system's architecture. For this purpose, decentralised monitoring algorithms account for the absence of a central observation point, e.g., [6] and [10]. Existing decentralised monitoring algorithms consider linear discrete



time and synchronous communication between the system components. In contrast, we address the verification of timed properties of continuous time in an asynchronous setting.

We introduce a decentralised monitoring algorithm that uses local knowledge of the global behaviour to express a verdict about a verified global timed property. In a synchronous setting, while a component lacks information on what events happened in other components, it can use a round-based approach to consider only a finite number of possibilities about global behaviour. In contrast, in an asynchronous setting, for a given time interval, there are no bounds on the number of events that can happen on another component, meaning that a local monitor should take into account an infinite number of scenarios for the events that it has not seen. Another challenge is that a monitor can be notified of past events from another component and has to update its local knowledge accordingly. Indeed, a method is proposed in [8] to account for a partial view of a global behaviour in a timed context, but it assumes that events that have not been seen cannot be seen afterwards. In our case, we assume that events that are not seen yet can still be seen in the future, and the local knowledge should be updated accordingly.

In this paper, we introduce several algorithms for monitoring timed properties in a decentralised setting, as well as an implementation to simulate and evaluate these algorithms. In Sec. 1, we present timed regular expressions, which we use to specify timed properties, and we explain how we evaluate them on a timed trace. After formally defining the decentralised monitoring problem (Sec. 2), we define a progression function that updates timed regular expression based on the local knowledge, first for the centralised setting and then for the decentralised one (Sec. 3). Afterwards (Sec. 4), we define two algorithms for decentralised monitoring of timed regular expressions, using the progression function mentioned above. Finally (Sec. 5), we present our implementation and experimental results. We compare with related work in Sec. 6 and conclude in Sec. 7.

1 Timed Words and Timed Regular Expressions

We recall the basic notions related to timed words, timed regular expressions, and regular languages in Sec. 1.1, using the same formalism as in [3]. In Sec. 1.2, we introduce reduced timed regular expressions and how to transform timed regular expressions into reduced ones, as they will serve in our monitoring framework. In Sec. 1.3, we transpose the semantics of timed regular expressions, from timed words to timed traces. We also propose new operators for timed traces.

We start by defining some basic notation: $\mathcal{I}$ denotes the set of intervals in $\mathbb{R}^+$ and for S a set, $\mathcal{P}(S)$ denotes the set of subsets of S. We also use $\cdot$ to denote the concatenation between two words.

1.1 Timed Regular Expressions and Timed Regular Languages [3]

We recall the syntax and semantics of timed words and timed regular expressions. Let Σ be an alphabet. A timed word (called *time-event sequence* in [3]) over the alphabet Σ is a word of $\mathbb{R}^+ \cup \Sigma$, composed of events in Σ and numerical values that represent delays, that is, the time between two consecutive events. As such, two consecutive delays can be added, which means that for two timed words u and v and for two delays x and y, $u \cdot x \cdot y \cdot v = u \cdot (x+y) \cdot v$. For example, $0.4 \cdot 1.1 \cdot b \cdot a \cdot 0.1 \cdot 0.2 \cdot c \cdot 2.1$ and $1.5 \cdot b \cdot a \cdot 0.3 \cdot c \cdot 2.1$ represent the same timed word. We denote by $\mathcal{T}(\Sigma)$ the set of timed words over Σ and by ϵ the empty word. A subset of $\mathcal{T}(\Sigma)$ is called *timed language*. A function $\theta : \Sigma_1 \to \Sigma_2 \cup \{\epsilon\}$ is called a *renaming*. We consider its natural extensions $\Sigma_1^* \to \Sigma_2^*$ and $\mathcal{T}(\Sigma_1) \to \mathcal{T}(\Sigma_2)$, and we use the same symbol

θ to denote them.

We use the classical word concatenation denoted by $\cdot$, but we also need an *absorbing concatenation* denoted by the operator $\circ$. Let us denote by $\delta : \mathcal{T}(\Sigma) \to \mathbb{R}^+$ the function that returns the sum of delays in a timed word. For two timed words u and v, if there exists a word w such that $\delta(u) \cdot w = v$, then we can define $u \circ v = u \cdot w$. This means that $u \circ v$ is defined if and only if v starts with a delay greater than the sum of delays in u, and if that is the case, then we remove this delay from the front of v before concatenating it to u. For example, $(a \cdot 2 \cdot b) \circ (3 \cdot c) = a \cdot 2 \cdot b \cdot 1 \cdot c$ while $(a \cdot 2 \cdot b) \circ (1 \cdot c)$ is not defined. Concatenation operators are extended to timed languages in the classical way. Moreover, for $n \in \mathbb{N}$ and $n \geq 2$, L^n and $L^{\circ n}$ respectively denote the language obtained by concatenating L with itself using operators $\cdot$ and $\circ$, respectively; while $L^0 = L^{\circ 0} = \{\epsilon\}$.

▶ **Definition 1** (Syntax of timed regular expressions). *Timed regular expressions over Σ are defined inductively by the following grammar where $I \subseteq \mathcal{I}$, $a \in \Sigma$, L' a timed regular expression over Σ' and $\theta : \Sigma' \to \Sigma \cup \{\epsilon\}$ a renaming:*

$$L := \epsilon \mid \underline{a} \mid \langle L \rangle_I \mid L \wedge L \mid L \vee L \mid L \cdot L \mid L \circ L \mid \theta(L') \mid L^\star \mid L^\circledast$$

Intuitively, a timed regular expression can be, respectively, the empty word, the letter a at any time, a language limited to time interval I, the conjunction ($\wedge$), disjunction ($\vee$), concatenation ($\cdot$), and absorbing concatenation ($\circ$) of two timed regular expressions, the renaming obtained through function θ, as well as the Kleene star ($\star$) and the Kleene star using the absorbing concatenation ($\circledast$) applied to a timed regular expression. The set of timed regular expressions obtained as above is denoted by $\mathcal{E}(\Sigma)$.

▶ **Definition 2** (Semantics of timed regular expressions). *The semantics of a timed regular expression is the timed language defined inductively by function $[\![\cdot]\!] : \mathcal{E}(\Sigma) \to \mathcal{P}(\mathcal{T}(\Sigma))$:*

- $[\![\epsilon]\!] = \{\epsilon\}$,
- $[\![\underline{a}]\!] = \{r \cdot a \mid r \in \mathbb{R}^+\}$,
- $[\![L_1 \wedge L_2]\!] = [\![L_1]\!] \cap [\![L_2]\!]$,
- $[\![L_1 \vee L_2]\!] = [\![L_1]\!] \cup [\![L_2]\!]$,
- $[\![L_1 \cdot L_2]\!] = [\![L_1]\!] \cdot [\![L_2]\!]$,
- $[\![L_1 \circ L_2]\!] = [\![L_1]\!] \circ [\![L_2]\!]$,
- $[\![\langle L \rangle_I]\!] = \{u \in [\![L_1]\!] \mid \delta(u) \in I\}$,
- $[\![L^\star]\!] = \bigcup_{i=0}^{\infty} [\![L^i]\!]$,
- $[\![L^\circledast]\!] = \bigcup_{i=0}^{\infty} [\![L^{\circ i}]\!]$,
- $[\![\theta(L)]\!] = \{\theta(u) \mid u \in [\![L]\!]\}$.

We call *timed regular languages* the languages defined by timed regular expressions.

▶ **Example 3** (Timed regular expressions). Let us consider some alphabet $\{a, b\}$:

- $(\underline{a} \vee \underline{b})^\star \cdot (\underline{a} \circ \langle \underline{b} \rangle_{[0;1]}) \cdot (\underline{a} \vee \underline{b})^\star$ denotes the language of timed words where at some point b occurs within one time unit after some a;
- $\langle \underline{a} \rangle^\star_{[0;1]}$ denotes the language of timed words composed of a's where each event occurs within one time unit from the previous one;
- $\langle \underline{a} \rangle^\circledast_{[0;1]}$ denotes the language of timed words where all events occur within the first time unit. Note that this is semantically equivalent to $\langle \underline{a}^\circledast \rangle_{[0;1]}$ and $\langle \underline{a}^\star \rangle_{[0;1]}$.

Two timed regular expressions L and L' are *equivalent* if their language is the same, that is, $[\![L]\!] = [\![L']\!]$.

1.2 Reduced Timed Regular Expressions

First, let us introduce reduced timed regular expressions.

▶ **Definition 4** (Reduced timed regular expression). *A* reduced timed regular expression *is a timed regular expression where the time constraining operator $\langle \cdot \rangle_I$ is applied only to singular events.*

Any timed regular expression can be rewritten into an equivalent reduced timed regular expression that defines the same language.

▶ **Proposition 5.** *Let L, L_1, L_2 be some timed regular expressions over Σ. We have:*

- $[\![\langle L_1 \vee L_2 \rangle_I]\!] = [\![\langle L_1 \rangle_I \vee \langle L_2 \rangle_I]\!]$,
- $[\![\langle L_1 \circ L_2 \rangle_I]\!] = [\![L_1 \circ \langle L_2 \rangle_I]\!]$,
- $[\![\theta(L) \rangle_I]\!] = [\![\theta(\langle L \rangle_I)]\!]$,
- $[\![\langle L^{\circledast} \rangle_I]\!] = [\![\langle \epsilon \rangle_I \vee \langle L^{\circledast} \circ L \rangle_I]\!] = [\![\langle \epsilon \rangle_I \vee L^{\circledast} \circ \langle L \rangle_I]\!]$,
- $[\![\langle L^{\star} \rangle_I]\!] = [\![\langle \epsilon \rangle_I \vee (\langle L^{\star} \cdot L \rangle_I]\!] = [\![\langle \epsilon \rangle_I \vee (L^{\star} \cdot L \wedge \langle \Sigma^{\circledast} \rangle_I)]\!]$.
- $[\![\langle L_1 \wedge L_2 \rangle_I]\!] = [\![\langle L_1 \rangle_I \wedge \langle L_2 \rangle_I]\!]$,
- $[\![\langle L_1 \cdot L_2 \rangle_I]\!] = [\![L_1 \cdot L_2 \wedge \langle \Sigma^{\circledast} \rangle_I]\!]$,
- $[\![\langle \epsilon \rangle_I]\!] = \{\epsilon\}$, *if* $0 \in I$, $\emptyset$ *otherwise,*

In the remainder, we only consider reduced timed regular expressions.

1.3 Semantics of Timed Regular Expressions over Timed Traces

In the context of decentralised monitoring, the monitor uses a trace as a sequence of time-stamped events. Formally, a *timed trace* over the alphabet Σ is a finite word over $\Sigma \times \mathbb{R}^+$ such that for two consecutive letters (α_1, t_1) and (α_2, t_2), we have $t_1 \leq t_2$. A timed trace is a sequence of events from Σ where each event is paired with the time at which it occurs. For example, $(a, 1.5) \cdot (b, 3.1) \cdot (a, 3.1) \cdot (c, 3.4)$ is a timed trace. Additionally, we also consider (ϵ, t) where ϵ is the empty word. Although this does not represent an observed event, it can be used to represent the absence of such an event. It can be simplified if there is an element following it with $(\epsilon, t) \cdot (\alpha, t') = (\alpha, t')$ for $\alpha \in \Sigma$. We denote by $\mathcal{R}(\Sigma)$ the set of timed traces over Σ.

For $\pi = (\alpha_1, t_1) \cdots (\alpha_n, t_n)$ a timed trace, let us denote by $\tau_{\text{first}}(\pi) = t_1$ (resp. $\tau_{\text{last}}(\pi) = t_n$) the time at which the first (resp. last) event of π occurs. We denote by $L{\downarrow_t}$ the language represented by $\theta_\epsilon(\langle \underline{x} \rangle_{[t,t]})$ where θ_ϵ is the renaming that maps everything to ϵ. Intuitively, $L{\downarrow_t}$ is the language of timed words $\{t \cdot u \mid u \in [\![L]\!]\}$. Similarly, we define $L{\uparrow^t}$ by shifting all the time constraints that appear before the first concatenation $(\cdot)$ in L by subtracting t. It can be defined inductively as such:

- $\underline{a}{\uparrow^t} = \underline{a}$,
- $(L_1 \wedge L_2){\uparrow^t} = L_1{\uparrow^t} \wedge L_2{\uparrow^t}$,
- $(L_1 \vee L_2){\uparrow^t} = L_1{\uparrow^t} \vee L_2{\uparrow^t}$,
- $(L_1 \cdot L_2){\uparrow^t} = L_1{\uparrow^t} \cdot L_2$,
- $(L_1 \circ L_2){\uparrow^t} = L_1{\uparrow^t} \circ L_2{\uparrow^t}$,
- $(\langle L \rangle_I){\uparrow^t} = \langle L{\uparrow^t} \rangle_{I-t}$
- $L^{\star}{\uparrow^t} = L{\uparrow^t} \vee L^{\star}$,
- $L^{\circledast}{\uparrow^t} = (L{\uparrow^t})^{\circledast}$,
- $\theta(L){\uparrow^t} = \theta(L{\uparrow^t})$.

▶ **Definition 6** (Semantics of timed regular expressions over timed traces)**.** *The semantics of a timed regular expression is the set of timed traces defined inductively by function $[\![\cdot]\!]_{\text{tr}}$:*

- $[\![\epsilon]\!]_{\text{tr}} = \{\epsilon\}$
- $[\![\underline{a}]\!]_{\text{tr}} = \{(a, t) \mid t \in \mathbb{R}^+\}$
- $[\![L_1 \wedge L_2]\!]_{\text{tr}} = [\![L_1]\!]_{\text{tr}} \cap [\![L_2]\!]_{\text{tr}}$
- $[\![L_1 \vee L_2]\!]_{\text{tr}} = [\![L_1]\!]_{\text{tr}} \cup [\![L_2]\!]_{\text{tr}}$
- $[\![\theta(L)]\!]_{\text{tr}} = \{\theta(u) \mid u \in [\![(L)]\!]_{\text{tr}}\}$
- $[\![L_1 \circ L_2]\!]_{\text{tr}} = \{u \cdot v \mid u \in [\![L_1]\!]_{\text{tr}}, v \in [\![L_2]\!]_{\text{tr}}\}$
- $[\![L_1 \cdot L_2]\!]_{\text{tr}} = \{u \cdot v \mid u \in [\![L_1]\!]_{\text{tr}}, v \in [\![L_2 {\downarrow_{\tau_{\text{last}}(u)}}]\!]_{\text{tr}}\}$
- $[\![\langle L \rangle_I]\!]_{\text{tr}} = \{\pi \in [\![L_1]\!]_{\text{tr}} \mid \tau_{\text{last}}(u) \in I\}$
- $[\![L^{\star}]\!]_{\text{tr}} = \bigcup_{i=0}^{\infty} [\![L^i]\!]_{\text{tr}}$
- $[\![L^{\circledast}]\!]_{\text{tr}} = \bigcup_{i=0}^{\infty} [\![L^{\circ i}]\!]_{\text{tr}}$

Let us denote by $\omega : \mathcal{R}(\Sigma)\mathcal{T}(\Sigma)$ the function that takes a timed trace $(\alpha_1, t_1) \cdot (\alpha_2, t_2) \cdots (\alpha_n, t_n)$ and returns the time word $t_1 \cdot \alpha_1 \cdot (t_2 - t_1) \cdot \alpha_2 \cdot (t_3 - t_2) \cdots (t_n - t_{n-1}) \cdot \alpha_n$. This function converts a timed trace into a timed word that represents the same sequence of events over physical time.

▶ **Proposition 7.** *For $u \in \mathcal{R}(\Sigma)$ and $L \in \mathcal{E}(\Sigma)$, $u \in [\![L]\!]_{\text{tr}}$ if and only if $\omega(u) \in [\![L]\!]$.*

Note that the timed words produced by function ω do not have a delay at the end. That is why we denote by $\sim$ the relation defined by $u \sim v$ if and only if there exists $x \in \mathbb{R}$ such that $v = u \cdot x$ or $u = v \cdot x$. We can show that every equivalence class in $\mathcal{T}(\Sigma)/\sim$ has only one representative that is an image of a timed trace by ω and this equivalence class has one unique reverse image by ω^{-1}.

▶ **Corollary 8.** *For $u \in \mathcal{T}(\Sigma)$ and $L \in \mathcal{E}(\Sigma)$, $u \in [\![L]\!]$ if and only if there exists $v \in \mathcal{T}(\Sigma)$ such that $v \sim u$ and $\omega^{-1}(v) \in [\![L]\!]_{\mathrm{tr}}$.*

This means that the language obtained by applying ω^{-1} to the language of timed words of an expression is exactly the language of timed traces of that expression.

1.4 Global operators

Using the timed traces semantics, we introduce two new operators $\lfloor . \rfloor_I$ and $\lceil \cdot \rceil^I$. $\lceil \cdot \rceil^I$ is similar to the $\langle \cdot \rangle_I$ operators, except that it refers to global time. In this case, global does not refer to the distributed nature of the systems we study, it means that it is based on the absolute time of the events, and not the relative time between two events. Of course, this operator would make no sense in the timed words definition of the semantics, as the concatenation of two words change the value of this global time for the word on the right side of the concatenation. This is why we express the semantics of these operators by extending $[\![\cdot]\!]_{\mathrm{tr}}$.

- $[\![\lceil L \rceil^I]\!] = \{u \in [\![L]\!]_{\mathrm{tr}} \mid \tau_{\mathrm{last}}(u) \in I\}$
- $[\![\lfloor L \rfloor_I]\!] = \{u \in [\![L]\!]_{\mathrm{tr}} \mid \tau_{\mathrm{first}}(u) \in I\}$

We call *global timed regular expressions* an expression that contains these operators and denote the set of expressions over Σ by $\mathcal{GE}(\Sigma)$.

▶ **Proposition 9.** *For all $L \in \mathcal{GE}(\Sigma)$, there exists $L' \in \mathcal{E}(\Sigma)$ such that $[\![L]\!]_{\mathrm{tr}} = [\![L']\!]_{\mathrm{tr}}$.*

This can be proven through direct construction or by using the fact that timed regular expressions are semantically equivalent to timed automata, and as such adding global constraints does not add to the expressiveness of timed automata.

▶ **Example 10.** The expression $\langle \underline{a} \rangle_{[0;3]} \cdot (\langle \underline{b} \rangle_{[0;2]})^{\circledast} \cdot \lfloor (\underline{a} \cdot \underline{b})^{+} \rfloor_{[4;5]}$ is semantically equivalent to $\langle \langle \underline{a} \rangle_{[0;3]} \cdot (\langle \underline{b} \rangle_{[0;2]})^{\circledast} \cdot \underline{a} \rangle_{[4;5]} \cdot \underline{b} \cdot (\underline{a} \cdot \underline{b})^{\star}$.

2 The Decentralised Timed Monitoring Problem

We formally state the decentralised timed monitoring problem as well as its objectives.

Context and notations. Let us suppose that the system at hand consists of n independent components denoted by C_i, for $0 < i \leq n$. Each component C_i emits local events over a local alphabet Σ_i. Let $\Sigma = \bigcup_{i \in [1,n]} \Sigma_i$ be the global alphabet of events. We assume that the local alphabets form a partition of Σ, which means that if $i \neq j$ then $\Sigma_i \cap \Sigma_j = \emptyset$. Let p_i be the projection of a timed trace of $\mathcal{R}(\Sigma)$ onto $\mathcal{R}(\Sigma_i)$ and denote by p the function defined by $p(\sigma) = (p_1(\sigma), p_2(\sigma), ..., p_n(\sigma))$. As such, after observing our local traces $\sigma_1, \sigma_2, ..., \sigma_n$, we can apply p^{-1} to obtain the possible global traces. We also note $\sigma|_t$ as the prefix of σ, which contains all events that occur before t. To each component C_i is attached a monitor M_i, which can observe a trace over Σ_i. We denote the verdict of a monitor by $\mathrm{Verdict}_i : \mathbb{R} \to \{\mathtt{bad}, \mathtt{inconclusive}\}$. Monitors are purposed to detect violations.

Assumptions. Our assumptions on the system are as follows.

- Messages can be exchanged between any pair of monitors.
- The monitors only observe their local trace and the messages they receive.
- If a message is sent, it is eventually received, which means that there is no loss of messages.
- The communication is done through FIFO channels, meaning that the messages sent from one monitor to another are received in the order they were sent.
- All monitors share the same global clock, meaning that there are no clock drifts.

Objectives. Given some timed regular expression L, and a global trace σ, our goal is to find an algorithm for the monitors such that the following properties hold.

▶ **Definition 11** (Definitive Verdict). *If there is a time t for which there is a verdict* $\text{Verdict}_i(t) = \texttt{bad}$*, then for all* $t' \geq t$, $\text{Verdict}_i(t') = \texttt{bad}$.

▶ **Definition 12** (Soundness). *If there is a time t for which there is a verdict* $\text{Verdict}_i(t) = \texttt{bad}$ *then $\sigma|_t$ is a bad prefix of L, i.e. for all $\sigma' \in \mathcal{R}(\Sigma)$, if $\sigma|_t$ is a prefix of σ' then $\sigma' \notin [\![L]\!]_{\text{tr}}$.*

▶ **Definition 13** (Eventual completeness). *If there is a time t for which $\sigma|_t$ is a bad prefix of L, then there is a delay $t' \geq 0$ such that* $\text{Verdict}_i(t + t') = \texttt{bad}$.

The goal is for one monitor, using its partial knowledge, to deduce whether or not its partial vision of the trace can be completed to be $[\![L]\!]_{\text{tr}}$.[1] Moreover, we also aim to minimise the time it takes to detect such a violation of the expression. Finally, to limit the communication overhead, we also aim to limit the number of messages sent, as well as the total size of the messages exchanged.

3 Progression for Timed Regular Expressions

We consider decentralised monitors that observe only their local events, indexed with the time at which they happen. For a monitor of index i, we define a *decentralised progression function* $\text{Pr}_i : \mathcal{GE}(\Sigma) \to (\Sigma \times \mathbb{R}^+) \to \mathcal{GE}(\Sigma)$, meaning a function that takes an event locally observed (α, t) and a specification as a timed regular expression L, and returns an expression that represents the new specification now that the monitor knows that (α, t) happened. In a centralised case, where there is only one monitor observing the events in the same order they occur, we denote such function Pr_0 and it should satisfy the following property:

▶ **Definition 14** (Centralised progression function). $\text{Pr}_0 : \mathcal{GE}(\Sigma) \to (\Sigma \times \mathbb{R}^+) \to \mathcal{GE}(\Sigma)$ *is a centralised progression function iff for any expression $L \in \mathcal{GE}(\Sigma)$ and any trace event (α, t),* $\text{Pr}_0(L)(\alpha, t) = \{u \mid (t \cdot \alpha) \circ u \in L\}$.

In other words, it is a left derivative. If a centralised monitor applies this function successively as it observes the events, and the resulting expression represents the empty language, then the monitor can produce a verdict. Whereas in the decentralised case where a monitor observes a local event (α, t), there might have been other events happening in another component at a time before t. In other words, for the decentralised case, Pr_i must satisfy the following property:

▶ **Definition 15** (Decentralised progression function). $\text{Pr}_i : \mathcal{GE}(\Sigma) \to (\Sigma \times \mathbb{R}^+) \to \mathcal{GE}(\Sigma)$ *is a decentralised progression function iff for any expression $L \in \mathcal{GE}(\Sigma)$ and any trace event (α, t), we have:* $[\![\text{Pr}_i(L)(\alpha, t)]\!]_{\text{tr}} = \{u \cdot v \in \mathcal{R}(\Sigma) \mid u \cdot (\alpha, t) \cdot v \in [\![L]\!]_{\text{tr}} \wedge u \in \mathcal{R}(\Sigma \setminus \Sigma_i)\}$.

[1] Note that we do not consider the alternative problem of determining that the trace will always be in $[\![L]\!]_{\text{tr}}$ for any possible future completion. This problem is, however, harder, as it requires testing whether or not an expression denotes the universal language, which is undecidable.

The above function is not a left derivative, as it allows events that are not from Σ_i to occur before the observed event. We now propose a decentralised progression function and detail its inductive definition. To define such a function, we define a filter function $\Phi : \mathcal{GE}(\Sigma) \to \mathcal{P}(\Sigma) \to \mathcal{GE}(\Sigma)$. The proper definition is given in the appendix. This function is defined such that it removes events of a given alphabet from an expression.

▶ **Proposition 16.** *For $L \in \mathcal{GE}(\Sigma)$ and $\Sigma' \subseteq \Sigma$, $[\![\Phi(L)(\Sigma')]\!] = [\![L \wedge (\Sigma \setminus \Sigma')^\star]\!]$.*

In other words, function Φ takes an expression $L \in \mathcal{GE}(\Sigma)$ and an alphabet $\Sigma' \subseteq \Sigma$ and returns an expression that represents the restriction of L to $\Sigma \setminus \Sigma'$. Let us define a progression function Pr_i for component C_i by looking at each case. First, let us look at the base cases:

- $\mathrm{Pr}_i(\underline{a})(\alpha, t) = \epsilon_t$, if $\alpha = a$
- $\mathrm{Pr}_i(\langle \underline{a} \rangle_I)(\alpha, t) = \emptyset$, if $t \notin I$
- $\mathrm{Pr}_i(\underline{a})(\alpha, t) = \emptyset$, if $\alpha \neq a$
- $\mathrm{Pr}_i(\langle \underline{a} \rangle_I)(\alpha, t) = \mathrm{Pr}_i(\underline{a})(\alpha, t)$, if $t \in I$

- $\mathrm{Pr}_i(L_1 \vee L_2)(\alpha, t) = \mathrm{Pr}_i(L_1)(\alpha, t) \vee \mathrm{Pr}_i(L_2)(\alpha, t)$
- $\mathrm{Pr}_i(L_1 \wedge L_2)(\alpha, t) = \mathrm{Pr}_i(L_1)(\alpha, t) \wedge \mathrm{Pr}_i(L_2)(\alpha, t)$

For those cases, the decentralised progression is straightforward, and if the event has been observed, it is removed from the formula. Then, let us look at the absorbing concatenation:

- $\mathrm{Pr}_i(L_1 \circ L_2)(\alpha, t) = (\mathrm{Pr}_i(L_1)(\alpha, t) \circ \lfloor L_2 \rfloor_{[t;\infty[}) \vee \lceil \Phi(L_1)(\Sigma_i) \rceil^{[0;t]} \circ \mathrm{Pr}_i(L_2)(\alpha, t)$
- $\mathrm{Pr}_i(L^{\circledast})(\alpha, t) = \lceil \Phi(L)(\Sigma_i)^{\circledast} \rceil^{[0;t]} \circ \mathrm{Pr}_i(L)(\alpha, t) \circ \lfloor L^{\circledast} \rfloor_{[t;\infty[}$

For the absorbing concatenation $\circ$, we examine whether (α, t) corresponds to an event on each side of the concatenation. If (α, t) is an event on the left side, then events on the right side must have occurred later. Otherwise, if (α, t) is an event on the right side, then the events on the left side must have happened earlier. Since they happened before the current time, it means that the current monitor can not have seen them, so they cannot be events from the monitor of index i.

Let us consider the progression for the global operators. For these cases, we use additional notation. First, for I an interval and $t \in \mathbb{R}$, we write $t < I$ (resp. $t > I$) if and only if for all $t' \in I, t < t'$ (resp. $t > t'$). We also denote by $I{\downarrow}$ the interval obtained by changing the lower bound of I to 0 inclusive, and by $I{\uparrow}$ the interval obtained by changing the upper bound of I to ∞. Hence, we have:

- $\mathrm{Pr}_i(\lceil L \rceil^I)(\alpha, t) = \lceil \mathrm{Pr}_i(L)(\alpha, t) \rceil^{I\downarrow}$, if $t \in I$
- $\mathrm{Pr}_i(\lceil L \rceil^I)(\alpha, t) = \lceil \mathrm{Pr}_i(L)(\alpha, t) \rceil^{I}$, if $t < I$
- $\mathrm{Pr}_i(\lceil L \rceil^I)(\alpha, t) = \emptyset$, otherwise
- $\mathrm{Pr}_i(\lfloor L \rfloor_I)(\alpha, t) = \lfloor \mathrm{Pr}_i(L)(\alpha, t) \rfloor_{I\uparrow}$, if $t \in I$
- $\mathrm{Pr}_i(\lfloor L \rfloor_I)(\alpha, t) = \lfloor \mathrm{Pr}_i(L)(\alpha, t) \rfloor_{I}$, if $t > I$
- $\mathrm{Pr}_i(\lfloor L \rfloor_I)(\alpha, t) = \emptyset$, otherwise

For $\lceil L \rceil^I$, we note that if (α, t) happened and $t \in I$, then it means that we can apply the progression to L provided that all other events occur before t or between t and the upper bound of I, which means that they occur in $I{\downarrow}$. On the other hand, if $t < I$, then it cannot be the last event seen, and we are still waiting for an event in I. We have a similar approach to $\lfloor L \rfloor_I$.

Finally, let us consider the non-absorbing concatenation. We want something similar to $\circ$ for $\cdot$ where we consider whether the event occurred on the right or left side. The problem here is that if the event happened on the right-hand side, then the result of our progression function depends on the time of the last event on the left-hand side since the concatenation "resets" the time at which we interpret the expression. Here we use the fact that the function $t' \to \mathrm{Pr}_i(L{\downarrow_{t'}})(\alpha, t)$ is a piece-wise constant function. This means that we can look at a finite number of possible intervals at which the last element of the left-hand side occurs. By building those intervals and computing the expression associated with each one, we can build

a function $\Delta_i : (\mathcal{GE}(\Sigma) \times \mathcal{I}) \to (\Sigma \times \mathbb{R}^+) \to \mathcal{P}(\mathcal{I} \times \mathcal{GE}(\Sigma))$. A more detailed definition of such a function is provided in the appendix.

- $\Pr_i(L_1 \cdot L_2)(\alpha, t) = \bigvee_{(I,L') \in \Delta_i(L_2, \mathbb{R}^+)(\alpha,t)} \lceil \Phi(L_1)(\Sigma_i) \rceil^I \cdot L' \vee \Pr_i(L_1)(\alpha, t) \cdot \lfloor L_2 \rfloor_{[t;\infty[}$
- $\Pr_i(L^\star)(\alpha, t) = \bigvee_{(I,L') \in \Delta_i(L, \mathbb{R}^+)(\alpha,t)} \lceil \Phi(L)(\Sigma_i)^\star \rceil^I \cdot L' \cdot \lfloor L^\star \rfloor_{[t;\infty[}$
- $\Pr_i(\theta(L))(\alpha, t) = \bigvee_{\alpha' \in u^{-1}(\alpha)} \theta(\Pr_i(L)(\alpha', t))$

▶ **Example 17.** Let us consider the language $L = (\underline{a} \vee \underline{b})^\circledast \cdot \langle \underline{a} \rangle_{[0;1]}$ with $\Sigma_1 = \{a\}, \Sigma_2 = \{b\}$. Then $\Pr_1(L)(a, 2) = (\lceil \underline{b}^\circledast \rceil^{[0;2]} \circ \lfloor (\underline{a} \vee \underline{b})^\circledast \cdot \langle \underline{a} \rangle_{[0;1]} \rfloor_{[2;\infty[}) \vee \lceil \underline{b}^\circledast \rceil^{[1;2]}$.
This means that, either the a observed was on the left of the concatenation, and in that case we have the language defined by L where before the global time $t = 2$, there can only be b events. Or the a observed was the one on the right side, so we can only observe events b that preceded it, with the last b between the global time $t = 1$ and $t = 2$.

We prove in appendix that this is a decentralised progression function $\Pr_i$ following Definition 15. As a consequence, we can prove the following theorem.

▶ **Theorem 18.** *For all i, j, $i \neq j$, for all $a \in \Sigma_i$ and $b \in \Sigma_j$, for all $t_a, t_b \in \mathbb{R}^+$ for all $L \in \mathcal{GE}(\Sigma)$, $[\![\Pr_j((\Pr_i(L)(a, t_a))(b, t_b)]\!]_{\mathrm{tr}} = [\![\Pr_i((\Pr_j(L)(b, t_b))(a, t_a)]\!]_{\mathrm{tr}}$.*

This means that the order at which we observe two events from two different monitors does not matter, and we always obtain an equivalent expression. Note that in the case $\Sigma_i = \Sigma$, then $\Pr_i$ also satisfies Definition 14 and we denote it $\Pr_0$, which means that we also have a centralised progression function, defined as a specific case of our decentralised progression function.

Let us inductively define function $\Pr^*$ such that for (α, t) a trace event of Σ_i and π a sequence of timed trace events, $\Pr^*(L)((\alpha, t) \cdot \pi) = \Pr^*(\Pr_i(L)(\alpha, t))(\pi)$ and $\Pr^*(L)(\epsilon) = L$. Note that we do not require π to be a timed trace, meaning that the events are not necessarily ordered by ascending time.

▶ **Corollary 19.** *For all $L \in \mathcal{GE}(\Sigma)$, for all π, π' such that for all i, $p_i(\pi) = p_i(\pi')$, $[\![\Pr^*(L)(\pi)]\!]_{\mathrm{tr}} = [\![\Pr^*(L)(\pi')]\!]_{\mathrm{tr}}$*

Corollary 19 implies that the order at which the events are observed does not change the language we obtain, provided that the local order on each component is preserved.

4 Decentralised Monitoring

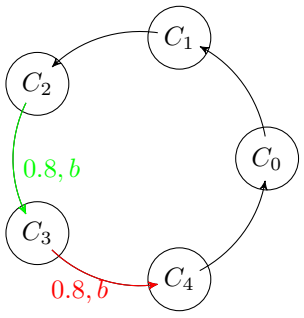
We define two algorithms to achieve decentralised monitoring for timed regular expressions. Both algorithms allow detecting a violation of the specification at runtime. The first algorithm simulates centralised monitoring, while the second algorithm leverages the decentralised progression function.

4.1 Simulating Centralised Monitoring

We place ourselves under the assumptions we described in Sec. 2. That is to say that when a message is sent, it is eventually received with no loss, and we assume *sequential consistency*, meaning that when one component sends multiple messages to another component, then they are received in the same order as they were sent. We also discard the possibility of two events happening at the exact same time in two different components. Since we consider real time, this is not something that would likely occur, but if it happened, we would not know in

```
1  L_i := specification for the system
2  when an internal event (t, α) is observed do
3      Send (C_i, α, t) to C_{i+1}
4  when a message (C, α, t) is received do
5      if k ≠ i then
6          add (α, t) to the memory
7          Send (C, α, t) to C_{i+1}
8      else
9          foreach (α', t') in the memory s.t.
             t' ≤ t (ordered by ascending t') do
10             L_i := Pr_0(L)(α', t')
11             Remove (α', t') from memory
12         L_i := Pr_0(L)(α, t)
13         if [[L_i]]_tr = ∅ then
14             return bad verdict
```

(a) Centralised progression algorithm for C_i.



(b) C_3 sends its message when it observes b at $t = 0.8$. When it receives it later (green message), then it knows for sure that it has seen all events up to $t = 0.8$.

Figure 1 Algorithm for decentralised monitoring using centralised progression

which order to consider them. But it can also be said that if the order of two simultaneous events on two different components mattered, then the specification would not be adapted to the system either. Finally, we also consider that the components are connected in a ring, as depicted in Figure 1b.

With these assumptions, let us consider that all components apply the algorithm shown in Figure 1a. It means that a component sends events that it sees to its successor. When it receives a message that it did not originate, it saves it into its memory and forwards it to its successor. Using sequential consistency, the following property holds:

▶ **Proposition 20.** *If C_i observes (α, t) and $C_{i'}$ observes (α', t') with $t < t'$, then $C_{i'}$ receives the message (C_i, α, t) before $(C_{i'}, \alpha', t')$.*

From this, we can deduce that when a monitor receives a message it originated, then it saw all the events that happened in the system up to the time when this message was first created. So it can simulate centralised monitoring on its memory up to that point. Using Proposition 20 and Definition 14 we can show the following.

▶ **Theorem 21.** *Let $\sigma \in \mathcal{R}(\Sigma)$ be the global timed trace, L the initial expression, and let (α, t) be the last event of this trace, with $\alpha \in \Sigma_i$. After C_i receives its last message (C_i, α, t), we have $[\![L_i]\!]_{\text{tr}} = \{\sigma' \in \mathcal{R}(\Sigma) \mid \sigma \cdot \sigma' \in L\}$.*

Emptiness checking

We know that, if at some point $[\![L_i]\!]_{\text{tr}} = \emptyset$ for some i, then the specification is violated. This entails testing the emptiness of an expression. One way to do so is to build a timed automaton that recognise the same language, using the construction shown in [3], as we know that the problem of knowing whether or not the language of a timed automaton is empty is PSPACE [1] and there are several tools that solve that problem. While this construction ensures an emptiness check, it is costly, and it is desirable to avoid it at runtime. Instead, we simplify the expression between each progression to detect when the denoted language is

Algorithm 1 Algorithm for C_i

```
1  if i = 1 then
2  |  L_i := specification
3  else
4  |  L_i := waiting
5  when an internal event (α, t) is observed do
6  |  if L_i = waiting then
7  |  |  add (α, t) to the memory
8  |  else
9  |  |  L_i := Pr_i(L)(e, t)
10 |  |  if [[L_i]]_tr = ∅ then
11 |  |  |  return bad verdict
12 when an expression L is received do
13 |  L_i = L foreach (α, t) in the memory do
14 |  |  L_i := Pr_i(L)(α, t)
15 |  |  Remove (α, t) from the memory
16 |  |  if [[L_i]]_tr = ∅ then
17 |  |  |  return bad verdict
18 when urgent(L_i) do
19 |  send L_i to target(L_i)
20 |  L_i := waiting
```

empty. The simplification used in this work is very simple. First we simplify every ϵ, ϵ_t or $\emptyset$ that appear in the expression. Then we find nested time constraints in order to merge them and simplify them as much as possible. This could be improved by simplifying conjunctions and disjunctions, but simplification of timed regular expressions is not well documented, and that is why we limit ourselves to these naive simplifications.

One could also notice that if $[\![L_i]\!]_{\text{tr}} = [\![\Sigma^\star]\!]_{\text{tr}}$ then it means that from this point onward the specification will always be satisfied. This means that we could possibly detect whether or not the specification is sure to be satisfied if we can test whether or not the language associated with the expression is universal. Unfortunately, this problem is not decidable in the general case [2], and that is why we only propose verdict `bad`.

4.2 Using Decentralised Progression

We now consider another approach where monitors do not exchange observed events, but instead there is one running expression passed along the monitors and that is updated with their progression function. In that case, at any given time, there is at most one monitor holding the expression and being marked as active. Monitors that are not active are only observing local events and recording them in their local memory. The active monitor updates a running expression using decentralised progression, based on its local observations. At some point, it passes this expression to another monitor and becomes not active. The monitor that receives that expression updates it with its own memory of observed events and becomes active. This is shown in Algorithm 1. Hence, in that case, we do not need sequential consistency, nor do we need the assumption of the components communicating in a ring. This algorithm uses two functions, urgent(L_i) and target(L_i). These functions decide when we want to pass the expression and to whom we want to pass it. There are several possible ways to implement them, as long as the expression is eventually passed to every component. In that case, the verdict reached by this algorithm is eventually the same as the centralised monitoring as a direct consequence of Corollary 19. We propose the following implementation of these functions. Function urgent(L) replaces every occurrence of $\lceil L' \rceil^I$ by $\emptyset$ and checks whether the resulting expression represents the empty language. If it is not empty, then it means that the specification can still be satisfied, even if nothing has been seen by the other monitors, and the active monitor returns `false`. Otherwise it means that we want to know what the other monitors observed and it returns `true`. The function target(L_i) can be implemented in multiple ways. First, we consider choosing target(L_i) $= i + 1 \bmod n$, which gives us something similar to the centralised approach, where the messages are sent to the

successor along the ring. We also propose another implementation where we try to detect which monitor is the most relevant for the past constraints $\lceil L \rceil^I$ present in the formula and choose it as the target.

Note that this approach has several advantages. The main one being that a monitor can decide that the specification has been violated even if it has only a partial view and has not communicated with all the other monitors. This also means that less messages can be exchanged, so the impact of communication delays is reduced. The next section describes experiments with these approaches.

5 Simulation and Benchmarks

In this section, we validate the effectiveness of our decentralised monitoring approaches by showing the benefits over an approach that simulates centralised monitoring. For this, we first briefly describe the implementation of our monitoring algorithms (Sec. 5.1). Then, we detail our experimental setup (Sec. 5.2) and obtained results, as well as some of the conclusions drawn (Sec. 5.3).

5.1 Implementation as an OCaml Benchmark

We implemented a simulation environment for the methods described in the previous sections. For this, we extended DecentMon [6, 7], an OCaml benchmark for decentralised monitoring in the discrete-time setting. We extended it to our timed setting and added support for regular timed expressions and timed traces. We implemented progression-based monitoring for the two methods. The extension for the timed setting consists of 1,400 LLOC. For the approach based on decentralised progression, we consider the following two alternatives for the target component chosen by a monitor when sending a message: (i) a dynamic approach where the target component is chosen based on the current expression, and (ii) a static approach where the target is chosen as the successor in a directed ring, as explained in the previous section.

5.2 Experimental Setup

We test and compare the three approaches for decentralised monitoring of timed regular expressions: (i) simulating centralised, (ii) decentralised with a dynamic target, and (iii) decentralised with a static ring target. We consider a network of 10 monitors, each of them capable of observing a different event. We consider the communication delay when a message is sent as a random value between 0 and 40 units of time. Timed regular expressions are not commonly used, since people prefer timed automata that are as expressive as timed regular expressions. This means that one approach could have been to take a benchmark of timed automata, compute a set of equivalent expressions and use them as a benchmark. But this poses a problem as the performance would be affected by our choice of equivalent expression. Another difficulty is the absence of reference benchmark for timed decentralised monitoring. That is why in this context, we choose to generate random timed regular expressions. We generate 100 expressions of a fixed size, for each size between 2 and 8. For the time constraints that may appear, we randomly chose the lower bound between 0 and 30, and the upper bound is ∞ or a sum between the lower bound and an integer chosen randomly between 0 and 30. For each of these expressions, we generate 100 timed traces of length 200, with a delay between two consecutive events chosen randomly between 0 and 10. Note that a randomly generated trace will most likely not respect a random specification, and the progression will find an empty language after a couple of steps. In order to avoid

Table 1 Summary of the experimental results.

Algorithm	Target	messages		# progressions	decision time
		# sent	total size		
Centralised	-	278.50	10305	278.50	257.99
Decentralised	dynamic	5.91	539426	10.46	142.71
	static	6.70	376944	11.78	143.78

those cases, we ensure that for an expression of size k, the progressed expression is not empty after observing the first $5 \times k$ events. To do so, we discard expressions where we cannot find such a trace in a reasonable time. For each trace, we perform monitoring according to the three aforementioned approaches.

During each experiment, we record the number of messages sent, the total size of the messages sent, the number of progressions performed, and the time taken to reach a verdict. This results in 2,900,000 tests for each of the three algorithms.

We evaluate the algorithms along three dimensions that are relevant for decentralised monitoring.

- Communication (messages). We measure the total number and size of the messages exchanged by the monitor. Since there are between 8 and 16 operators, they are encoded over 4 bits. The time values in timed constraints are encoded over 32 bits. There are 10 possible events encoded over 4 bits
- Computation (progressions). We measure the total number of calls to the progression functions defined in the previous sections, including the recursive calls.
- Delay (decision time). We measure the time it takes for one of the monitors to detect that the current global trace violates the monitored timed regular expression.

Note that here we do not consider the computation time associated with each progression computation. We consider that this time should be negligible compared to the communication delays.

5.3 Results and Discussion

Tab. 1 reports the results of our experiments, reporting the average values for each of the metrics. In the following, we discuss the results and conclude.

First, let us compare the two decentralised approaches. Choosing a dynamic target gives us slightly better results on all metrics, except for the size of the messages. The difference remains marginal, and it seems that in our experiments, dynamically choosing the target does not give a significantly shorter decision time. Of course, this stems from the choice function, which can be improved, as it showed poor results in some specific cases. Although these are seldom, their impact on the average values is noticeable. We believe that a more in-depth analysis of these cases can give us a better choice of target.

However, the performance of the decentralised approach is significantly better than that of the centralised approach, where the number of messages and progressions is higher. Indeed, in the centralised approach, every monitor records each event and sends a message for each one, meaning that each monitor applies many progressions. This is not the case in the decentralised approach, where only a few messages are seen by each monitor. It is also shown that the time required to pass observations along all monitors in the centralised approach is much longer. Indeed, in order to apply the progression function on a local event it has observed, a monitor has to wait for it to circulate along all the other monitors; with 10

monitors and an average communication delay of 20 time units, it means that the monitor has to wait on average for 200 time units.

Of course, the trade-off can be seen in the size of the sent messages. This is because the centralised approach sends timed events, while the decentralised approach sends timed expressions. Those expressions are written with timed constraints, that tend to pile up after several progressions, and that increase the size of the expression. However, we can imagine two ways to alleviate this issue. The first would be to improve the simplification of expressions to reduce the size of expressions, possibly by simplifying some global time constraints or some conjunctions.

Another idea would be to send either a history of the timed events or an expression depending on which one is smaller. Indeed, each monitor could compute the expression if it received some history of the timed events. This approach is hard to implement, as it would mean that monitors should remember what other monitors have seen so far.

Another data that we did not show in the table is the impact of the communication delay relative to the delay between consecutive events. In fact, with higher communication delays, the simulated centralised approach has a decision time that increases much more than the decentralised approach. This is because the centralised method requires the message to travel along the entirety of the ring before taking any decision, which slows down the decision time proportionality to both the number of monitors and the delay of communication. This is less of a problem for the decentralised approach, which can decide by exchanging fewer messages between a few monitors. In our tests, we have even seen that in many cases, the decentralised approach decided with fewer messages as we increased the communication delay. This happens because when a message is received after being in transit for a long time, the receiving monitor has had enough time to build a long history that would violate the property. So we can deduce that when the communication delay is high, the decentralised approach should be the main option.

6 Related Work

As this paper introduces decentralised runtime verification for timed properties described by timed regular expressions [3], the related approaches consist of those for monitoring timed properties and those decentralising the monitoring process. In monitoring timed specifications, research efforts have mainly focused on synthesising decision procedures (monitors) for timed properties. Bauer et al. [5] introduce a variant of Timed Linear-time Temporal Logic (TLTL), a timed extension of Linear-time Temporal Logic, with a semantics tailored for runtime verification defined on finite traces. They additionally synthesise finite-trace monitors from TLTL formulas. Metric Temporal Logic (MTL) is another extension of LTL, with dense time. Nickovic et al. [16] translate MTL formulas into timed automata and Thati et al. [18] use a tailored progression function to evaluate formulas at runtime. More recently, Grez et al. [15] consider the monitoring problem for timed automata by introducing a data structure that allows the monitoring of a non-deterministic 1-clock automaton. Pinisetty et al. [17] introduce a predictive setting for runtime verification of timed properties leveraging reachability analysis to anticipate the detection of verdicts. Specific to timed regular expressions [3], Montre [21] is a tool monitoring using timed pattern matching. The mentioned approaches consider that the monitored system is centralised, and the decision procedure is fed with a unique trace containing complete observations.

Decentralised runtime verification has been introduced in [6], see [10] for a recent overview. Approaches in decentralised runtime verification take as input Linear-time Temporal Logic formulas such as [6, 7, 14] or finite-state automata such as in [11, 9]. All these approaches

monitor specifications of discrete time, which is much simpler and does not account for the physical time that impacts the evaluation of the specification as well as the moment at which monitors perform their evaluation.

Finally, we note that decentralised runtime verification resembles diagnosis [22, 23], which tries to detect the occurrence of a fault after a finite number of discrete steps and the component responsible for the fault. Our approach differs from diagnosis, as we assume that monitors' (combined) local information suffices to detect violations. However, in diagnosis, the model of the system is taken as input, and a central decision-making point is assumed. Similar to diagnosis, there is decentralised observability [19, 20] that combines the state of local observers with locally or globally bounded or unbounded memory to get a truthful verdict. While [19] requires a central decision-making point, the recent approach in [20] introduces the "at least one can tell" condition, which characterises when local agents can evaluate the global behaviour. While this approach, like ours, does not require a central observation point, it does not allow monitoring of the membership to an arbitrary timed regular expression.

7 Conclusion and Perspectives

We have introduced centralised and decentralised progression for timed regular expressions and have shown how it can be used to implement several algorithms to achieve decentralised monitoring of timed properties described by timed regular expressions. While several approaches exist for decentralised monitoring of untimed properties and centralised monitoring of timed properties, this is the first realisation of decentralised monitoring of timed properties. We have implemented the decentralised monitoring algorithms and evaluated their runtime behaviour costs in metrics relevant to decentralised monitoring.

These results give insights and research directions to improve these methods, such as using better simplification rules or having a better choice of targets when the expression is passed between monitors. Alternatively, decentralised monitoring approaches can be designed for properties described by timed automata, as they are well adopted in the community. Since there is an equivalence between timed regular expressions and timed automata, one could simply implement the transformation from automata to expressions. However, we believe that finding an analogue of progression for timed automata seems promising, as it could outperform the methods shown in this paper using knowledge of the states and transitions, as in [11] with finite-state automata. Finally, another perspective is to define progression for Metric Temporal Logic (MTL). Indeed, if we find a progression that satisfies the same properties as those shown in this paper, the same algorithms can be applied.

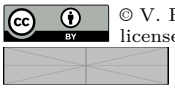
References

1 R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990.

2 Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

3 Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.

4 Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018.



5 Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.

6 Andreas Klaus Bauer and Yliès Falcone. Decentralised LTL monitoring. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2012.

7 Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods Syst. Des.*, 49(1-2):109–158, 2016.

8 Daniel de Leng and Fredrik Heintz. Approximate stream reasoning with metric temporal logic under uncertainty. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019.

9 Antoine El-Hokayem and Yliès Falcone. On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. *ACM Trans. Softw. Eng. Methodol.*, 29(1):1:1–1:57, 2020.

10 Yliès Falcone. On decentralized monitoring. In Ayoub Nouri, Weimin Wu, Kamel Barkaoui, and ZhiWu Li, editors, *Verification and Evaluation of Computer and Communication Systems - 15th International Conference, VECoS 2021, Virtual Event, November 22-23, 2021, Revised Selected Papers*, volume 13187 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2021.

11 Yliès Falcone, Tom Cornebize, and Jean-Claude Fernandez. Efficient and generalized decentralized monitoring of regular languages. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings*, volume 8461 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2014.

12 Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred Broy, Doron A. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

13 Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 23(2):255–284, 2021.

14 Florian Gallay and Yliès Falcone. Decentralized LTL enforcement. In Pierre Ganty and Davide Bresolin, editors, *Proceedings 12th International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2021, Padua, Italy, 20-22 September 2021*, volume 346 of *EPTCS*, pages 135–151, 2021.

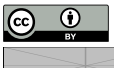
15 Alejandro Grez, Filip Mazowiecki, Michal Pilipczuk, Gabriele Puppis, and Cristian Riveros. The monitoring problem for timed automata. *CoRR*, abs/2002.07049, 2020.

16 Dejan Nickovic and Oded Maler. AMT: a property-based monitoring tool for analog systems. In Jean-François Raskin and P. S. Thiagarajan, editors, *Proceedings of the 5th International Conference on Formal modeling and analysis of timed systems (FORMATS 2007)*, volume 4763 of *Lecture Notes in Computer Science*, pages 304–319. Springer-Verlag, 2007.

17 Srinivas Pinisetty, Thierry Jéron, Stavros Tripakis, Yliès Falcone, Hervé Marchand, and Viorel Preoteasa. Predictive runtime verification of timed properties. *J. Syst. Softw.*, 132:353–365, 2017.

18 Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

19 Stavros Tripakis. Decentralized observation problems. In *44th IEEE IEEE Conference on Decision and Control and 8th European Control Conference Control, CDC/ECC 2005, Seville, Spain, 12-15 December, 2005*, pages 6–11. IEEE, 2005.



20 Stavros Tripakis and Karen Rudie. Decentralized observation of discrete-event systems: At least one can tell. *IEEE Control. Syst. Lett.*, 6:1652–1657, 2022.

21 Dogan Ulus. Montre: A tool for monitoring timed regular expressions. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 329–335. Springer, 2017.

22 Yin Wang, Tae-Sic Yoo, and Stéphane Lafortune. Diagnosis of discrete event systems using decentralized architectures. *Discret. Event Dyn. Syst.*, 17(2):233–263, 2007.

23 Xiang Yin and Stéphane Lafortune. Codiagnosability and coobservability under dynamic observations: Transformation and verification. *Autom.*, 61:241–252, 2015.

A Definition of Φ

- $\Phi(\underline{a})(\Sigma') = \underline{a}$ if $a \notin \Sigma'$
- $\Phi(\langle \underline{a} \rangle_I)(\Sigma') = \langle \underline{a} \rangle_I$ if $a \notin \Sigma'$
- $\Phi(L^{\circledast})(\Sigma') = \Phi(L)(\Sigma')^{\circledast}$
- $\Phi(\lceil L \rceil^I)(\Sigma') = \lceil \Phi(L)(\Sigma') \rceil^I$
- $\Phi(\theta(L))(\Sigma') = \theta(\Phi(L)(\theta^{-1}(\Sigma')))$
- $\Phi(\underline{a})(\Sigma') = \emptyset$ if $a \in \Sigma'$
- $\Phi(\langle \underline{a} \rangle_I)(\Sigma') = \emptyset$ if $a \in \Sigma'$
- $\Phi(L^{\star})(\Sigma') = \Phi(L)(\Sigma')^{\star}$
- $\Phi(\lfloor L \rfloor_I)(\Sigma') = \lfloor \Phi(L)(\Sigma') \rfloor_I$

- $\Phi(L_1 \vee L_2)(\Sigma') = \Phi(L_1)(\Sigma') \vee \Phi(L_2)(\Sigma')$
- $\Phi(L_1 \wedge L_2)(\Sigma') = \Phi(L_1)(\Sigma') \wedge \Phi(L_2)(\Sigma')$
- $\Phi(L_1 \circ L_2)(\Sigma') = \Phi(L_1)(\Sigma') \circ \Phi(L_2)(\Sigma')$
- $\Phi(L_1 \cdot L_2)(\Sigma') = \Phi(L_1)(\Sigma') \cdot \Phi(L_2)(\Sigma')$

B Definition of Δ_i

For I, I' two intervals, we denote $I \lhd I' = \{x \in I \mid \forall t \in I', x < t\}$.
And $I \rhd I' = \{x \in I \mid \forall t \in I', x > t\}$ such that $I \lhd I'$, $I \rhd I'$ and $I \cap I'$ are always a partition of I. We define Δ_i as follows:

- $\Delta_i(\underline{a}, I)(\alpha, t) = \{(I, \epsilon)\}$ if $\alpha = a$.
- $\Delta_i(\underline{a}, I)(\alpha, t) = \{(I, \emptyset)\}$ otherwise.
- $\Delta_i(\langle \underline{a} \rangle_{I'}, I)(\alpha, t) = \{(I \rhd I', \emptyset); (I \lhd I', \emptyset); (t - I' \cap I, \epsilon)\}$ if $\alpha = a$.
- $\Delta_i(\langle \underline{a} \rangle_{I'}, I)(\alpha, t) = \{(\emptyset, I)\}$ otherwise.
- $\Delta_i(L_1 \vee L_2, I)(\alpha, t) =$
  $\{(I_1 \cap I_2, L_1' \vee L_2') \mid (I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t), (I_2, L_2') \in \Delta_i(L_2, I)(\alpha, t)\}$
- $\Delta_i(L_1 \wedge L_2, I)(\alpha, t) =$
  $\{(I_1 \cap I_2, L_1' \wedge L_2') \mid (I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t), (I_2, L_2') \in \Delta_i(L_2, I)(\alpha, t)\}$
- $\Delta_i(L_1 \circ L_2, I)(\alpha, t) = \{(I_1 \cap I_2, L_1' \circ \lfloor L_2 \rfloor_{[t;\infty[} \vee \lceil \Phi(L_1)(\Sigma_i) \rceil^{[0;t]} \circ L'2) \mid (I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t), (I_2, L_2') \in \Delta_i(L_2, I)(\alpha, t)\}$
- $\Delta_i(L_1^{\circledast}, I)(\alpha, t) =$
  $\{(I_1, \lceil Past(L_1)(\Sigma_i)^{\circledast} \rceil^{[0;t]} \circ L'1 \circ \lfloor L_1^{\circledast} \rfloor_{[t;\infty[}) \mid (I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t)\}$
- $\Delta_i(L_1 \cdot L_2, I)(\alpha, t) =$
  $\{(I_1, (\bigvee_{(I_2, L_2') \in \Delta_i(L_2, I)(\alpha, t)} \lceil \Phi(L_1)(\Sigma_i)) \cdot L_2' \rceil^{I_2} \vee L_1' \circ \lfloor L_2 \rfloor_{[t;\infty[}) \mid (I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t)\}$
- $\Delta_i(L_1^{\star}, I)(\alpha, t) = \{(I_1, \bigvee_{(I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t)} \lceil \Phi(L_1)(\Sigma_i)^{\star} \rceil^{I_1} \cdot L_1' \cdot \lfloor L_1^{\star} \rfloor_{[t;\infty[})\}$

This function satisfies the following properties.

$$\forall t, I, L, \forall (I', L') \in \Delta_i(L, I)(\alpha, t), \forall t' \in I', L' = \mathrm{Pr}_i(L \downarrow_{t'})(\alpha, t)$$

$$\forall t, I, L, \forall (I_1, L_1), (I_2, L_2) \in \Delta_i(L, I)(\alpha, t), (I_1, L_1) \neq (I_2, L_2) \Leftrightarrow I_1 \cap I_2 = \emptyset$$

$$\forall t, I, L, \bigcup_{(I', L') \in \Delta_i(L, I)(\alpha, t)} I' = I$$

C Proof that Pr_i is a decentralised progression function

Proof. Let us prove this by induction. We will consider the cases that are not immediately apparent :

- Let $L = L_1 \circ L_2$. Let us prove the double inclusion.
  1. Assume $u \in [\![\mathrm{Pr}_i(L)(\alpha, t)]\!]_{\mathrm{tr}}$. There are then two possible cases.
     - First case: $u \in [\![\mathrm{Pr}_i(L_1)(\alpha, t) \circ \lfloor L_2 \rfloor_{[t;\infty[}]\!]_{\mathrm{tr}}$ which means that $u = u_1 \cdot u_2$ with $u_1 \in [\![\mathrm{Pr}_i(L_1)(\alpha, t)]\!]_{\mathrm{tr}}$ and $u_2 \in [\![L_2]\!]_{\mathrm{tr}}$ and $\tau_{\mathrm{first}}(u_2) \geq t$. Using our induction hypothesis $u_1 = v_1 \cdot v_2$ with $v_1 \cdot (\alpha, t) \cdot v_2 \in L_1$ and $v_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$. Therefore, we proved $v_1 \cdot (\alpha, t) \cdot v_2 \cdot u_2 \in [\![L]\!]_{\mathrm{tr}}$.
     - Second case $u \in [\![\lceil \Phi(L_1)(\Sigma_i) \rceil^{[0;t]} \circ \mathrm{Pr}_i(L_2)(\alpha, t)]\!]_{\mathrm{tr}}$, then it means that $u = u_1 \cdot u_2$ with $u_1 \in [\![\Phi(L_1)(\Sigma_i)]\!]_{\mathrm{tr}}$ and $u_2 \in [\![\mathrm{Pr}_i(L_2)(\alpha, t)]\!]_{\mathrm{tr}}$. This means that $u_1 \in [\![L_1]\!]_{\mathrm{tr}} \cap \mathcal{R}(\Sigma \setminus \Sigma_i)$. Using our induction hypothesis, we have $u_2 = v_1 \cdot v_2$ with $v_1 \cdot (\alpha, t) \cdot v_2 \in [\![L_2]\!]_{\mathrm{tr}}$ and $v_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$. We can then deduce $u_1 \cdot v_1 \cdot (\alpha, t) \cdot v_2 \in [\![L]\!]_{\mathrm{tr}}$ and $(u_1 \cdot v_1) \in \mathcal{R}(\Sigma \setminus \Sigma_i)$

     This means that in both cases $u \in \{u' \cdot v' \mid u' \cdot (\alpha, t) \cdot v' \in [\![L]\!]_{\mathrm{tr}} \text{ and } u' \in \mathcal{R}(\Sigma \setminus \Sigma_i)\}$
  2. Let us prove the other side of the inclusion. Let us assume u and v such that $u \cdot (\alpha,) \cdot v \in [\![L]\!]_{\mathrm{tr}} \wedge u \in \mathcal{R}(\Sigma \setminus \Sigma_i)\}$. This means that $u \cdot (\alpha, t) \cdot v = u' \cdot v'$ with $u' \in [\![L_1]\!]_{\mathrm{tr}}$ and $v' \in [\![L_2]\!]_{\mathrm{tr}}$. We can then deduce that we either have $u' = u \cdot (\alpha, t) \cdot v_1$ or have $v' = u_1 \cdot (\alpha, t) \cdot v$. In other words, we have either $u \cdot v_1 \in [\![\mathrm{Pr}_i(L_1)(\alpha, t)]\!]_{\mathrm{tr}}$ with $v' \in [\![\lfloor L_2 \rfloor_{[t,\infty[}]\!]_{\mathrm{tr}}$ or $u_1 \cdot v \in [\![\mathrm{Pr}_i(L_2)(\alpha, t)]\!]_{\mathrm{tr}}$ with $u' \in L_1$ and in the last case, every element before (α, t) cannot contain elements of Σ_i, therefore it must be true for u' which means $u' \in [\![\lceil \Phi_i(L_1)(\Sigma_i) \rceil^{[0;t]}]\!]_{\mathrm{tr}}$. This means that in both cases $u \cdot v \in [\![\mathrm{Pr}_i(L)(\alpha, t)]\!]_{\mathrm{tr}}$
- If $L = L_1^{\circledast}$.
  1. Let us assume $u \in [\![\mathrm{Pr}_i(L)(\alpha, t)]\!]_{\mathrm{tr}}$. This means that $u = u_1 \cdot u_2 \cdot u_3$ with $u_1 \in [\![\lceil \Phi(L_1)(\Sigma_i)^{\circledast} \rceil^{[0;t]}]\!]_{\mathrm{tr}}$, $u_3 \in [\![\lfloor L_1^{\circledast} \rfloor_{[t;\infty[}]\!]_{\mathrm{tr}}$ and $u_2 \in [\![\mathrm{Pr}_1(L_1)(\alpha, t)]\!]_{\mathrm{tr}}$. Therefore, we know that $u_1 \in [\![L_1^{\circledast}]\!]_{\mathrm{tr}}$, $u_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$, and $u_2 = v_1 \cdot v_2$ with $v_1 \cdot (\alpha, t) \cdot v_2 \in [\![L_1]\!]_{\mathrm{tr}}$. Hence $u_1 \cdot v_1 \cdot (\alpha, t) \cdot v_2 \cdot v_3 \in [\![L_1^{\circledast}]\!]_{\mathrm{tr}}$ with $u_1 \cdot v_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$
  2. Now to prove the other inclusion. Let us assume $u \cdot v$ such that $w = u \cdot (\alpha, t) \cdot v \in [\![L]\!]_{\mathrm{tr}}$. Since w is not empty, it is equal to $w_1 \cdot w_2 \cdot .... \cdot w_n$, with $w_i \in [\![L_1]\!]_{\mathrm{tr}}$. Because we know that w contains (α, t), we can denote by k the index of the w_i containing this event. In other words, we have $w_k = u' \cdot (\alpha, t) \cdot v' \in [\![L_1]\!]_{\mathrm{tr}}$, which means that $u' \cdot v' \in [\![Prog_i(L_1)(\alpha, t)]\!]_{\mathrm{tr}}$. This proves that $u \cdot v = w_0 \cdot ... \cdot w_{k-1} \cdot u' \cdot v' \cdot w_{k+1} \cdot ...w_n \in [\![\lceil \Phi(L_1)(\Sigma_i)^{\circledast} \rceil^{[0;t]} \circ \mathrm{Pr}_i(L_1)(\alpha, t) \circ \lfloor L_1^{\circledast} \rfloor_{[t;\infty[}]\!]_{\mathrm{tr}}$
- Let $L = L_1 \cdot L_2$. Let us prove the double inclusion.
  1. Assume $u \in [\![\mathrm{Pr}_i(L)(\alpha, t)]\!]_{\mathrm{tr}}$. There are then two possible cases.
     - First case: $u \in [\![\mathrm{Pr}_i(L_1)(\alpha, t) \circ \lfloor L_2 \rfloor_{[t;\infty[}]\!]_{\mathrm{tr}}$ then means that $u = u_1 \cdot u_2$ with $u_1 \in [\![\mathrm{Pr}_i(L_1)(\alpha, t)]\!]_{\mathrm{tr}}$ and $u_2 \in [\![L_2 \downarrow_{\tau_{\mathrm{last}}(u_1)}]\!]_{\mathrm{tr}}$ and $\tau_{\mathrm{first}}(u_2) \geq t$. Using our induction hypothesis $u_1 = v_1 \cdot v_2$ with $v_1 \cdot (\alpha, t) \cdot v_2 \in L_1$ and $v_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$. This allows us to conclude that $v_1 \cdot (\alpha, t) \cdot v_2 \cdot u_2 \in [\![L]\!]_{\mathrm{tr}}$.

- Second case: There is $(I, L') \in \Delta_i(L_2, \mathbb{R}^+)(\alpha, t)$ such that $u \in [\![\lceil \Phi(L_1)(\Sigma_i) \rceil^I \cdot L'(\alpha, t)]\!]_{\text{tr}}$. It means that $u = u_1 \cdot u_2$ with $u_1 \in [\![\Phi(L_1)(\Sigma_i)]\!]_{\text{tr}}$ and $u_2 \in [\![L'(\alpha, t)]\!]_{\text{tr}}$. This means that $u_1 \in [\![L_1]\!]_{\text{tr}} \cap \mathcal{R}(\Sigma \setminus \Sigma_i)$. Taking into account the first property of Δ_i, we can also see that $u_2 \in [\![\text{Pr}_i(L_2 \downarrow_{\tau_{\text{last}}(u_1)})(\alpha, t)]\!]_{\text{tr}}$. Using our induction hypothesis, we have $u_2 = v_1 \cdot v_2$ with $v_1 \cdot (\alpha, t) \cdot v_2 \in [\![L_2 \downarrow_{\tau_{\text{last}}(u_1)}]\!]_{\text{tr}}$ and $v_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$. Therefore, we have $u_1 \cdot v_1 \cdot (\alpha, t) \cdot v_2 \in [\![L]\!]_{\text{tr}}$ and $(u_1 \cdot v_1) \in \mathcal{R}(\Sigma \setminus \Sigma_i)$

2. Let us prove the other side of the inclusion. Assume u and v such that $u \cdot (\alpha,) \cdot v \in [\![L]\!]_{\text{tr}} \wedge u \in \mathcal{R}(\Sigma \setminus \Sigma_i)\}$. This means that $u \cdot (\alpha, t) \cdot v = u' \cdot v'$ with $u' \in [\![L_1]\!]_{\text{tr}}$ and $v' \in [\![L_2 \downarrow_{\tau_{\text{last}}(u')}]\!]_{\text{tr}}$. We can deduce that we have $u' = u \cdot (\alpha, t) \cdot v_1$ or we have $v' = u_1 \cdot (\alpha, t) \cdot v$.
   In other words, we have $u \cdot v_1 \in [\![\text{Pr}_i(L_1)(\alpha, t)]\!]_{\text{tr}}$ with $v' \in [\![\lfloor L_2 \rfloor_{[t,\infty[}]\!]_{\text{tr}}$ or we have $u_1 \cdot v \in [\![\text{Pr}_i(L_2 \downarrow_{\tau_{\text{last}}(u')})(\alpha, t)]\!]_{\text{tr}}$ with $u' \in L_1$. In the first case, we have $u \cdot v \in [\![\text{Pr}_i(L)(\alpha, t)]\!]_{\text{tr}}$.
   Let us look at the other case. Every element before (α, t) cannot contain elements of Σ_i, therefore it must be true of u', which means $u' \in [\![\Phi_i(L_1)(\Sigma_i)]\!]_{\text{tr}}$. Using the second and third properties of Δ_i, we can see that there is one and only one $(I, L') \in \Delta_i(L_2, \mathbb{R}^+)(\alpha, t)$ such that $\tau_{\text{last}}(u') \in I$. That means we have $u' \in [\![\lceil \Phi_i(L_1)(\Sigma_i) \rceil^I]\!]_{\text{tr}}$ and $u_1 \cdot v \in [\![L']\!]_{\text{tr}}$. We then proved in both cases that $u \cdot v \in [\![\text{Pr}_i(L)(\alpha, t)]\!]_{\text{tr}}$.

- If $L = L_1^\star$.

1. Assume $u \in [\![\text{Pr}_i(L)(\alpha, t)]\!]_{\text{tr}}$. This means that there is $(I, L') \in \Delta_i(L_1, \mathbb{R}^+)(\alpha, t)$ such that $u = u_1 \cdot u_2 \cdot u_3$ with $u_1 \in [\![\lceil \Phi(L_1)(\Sigma_i)^\star \rceil^I]\!]_{\text{tr}}$ and $u_3 \in [\![\lfloor L_1^\star \rfloor_{[t;\infty[} \downarrow_{\tau_{\text{last}}(u_2)}]\!]_{\text{tr}}$ as well as $u_2 \in [\![L']\!]_{\text{tr}}$. Using the first property of Δ_i, this means $u_2 \in [\![\text{Pr}_i(L_1 \downarrow_{\tau_{\text{last}}(u_1)})(\alpha, t)]\!]_{\text{tr}}$. In other words, we know that $u_1 \in [\![L_1^\star]\!]_{\text{tr}}$, $u_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$ and $u_2 = v_1 \cdot v_2$ with $v_1 \cdot (\alpha, t) \cdot v_2 \in [\![L_1 \downarrow_{\tau_{\text{last}}(u_1)}]\!]_{\text{tr}}$. We can then deduce that $u_1 \cdot v_1 \cdot (\alpha, t) \cdot v_2 \cdot v_3 \in [\![L_1^\star]\!]_{\text{tr}}$ with $u_1 \cdot v_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$
2. Now to prove the other inclusion. Assume $u \cdot v$ such that $w = u \cdot (\alpha, t) \cdot v \in [\![L]\!]_{\text{tr}}$. Since w is not empty, $w = w_1 \cdot w_2 \cdot .... \cdot w_n$, with $w_i \in [\![L_1 \downarrow_{\tau_{\text{last}}(w_{i-1})}]\!]_{\text{tr}}$. Because we know that w contains (α, t), we can denote by k the index of w_i that contains this event. In other words, we have $w_k = u' \cdot (\alpha, t) \cdot v' \in [\![L_1 \downarrow_{\tau_{\text{last}}(w_{k-1})}]\!]_{\text{tr}}$, which means that $u' \cdot v' \in [\![Prog_i(L_1 \downarrow_{\tau_{\text{last}}(w_{i-1})})(\alpha, t)]\!]_{\text{tr}}$. Using the second and third properties of Δ_i we deduce that there is one and only one $(I, L') \in \Delta_i(L_1, \mathbb{R}^+)(\alpha, t)$ such that $\tau_{\text{last}}(w_{k-1}) \in I$ and in that case we have $w_k \in [\![L']\!]_{\text{tr}}$. This shows that $u \cdot v = w_0 \cdot ... \cdot w_{k-1} \cdot u' \cdot v' \cdot w_{k+1} \cdot ...w_n \in [\![\lceil \Phi(L_1)(\Sigma_i)^\star \rceil^I \cdot L' \cdot \lfloor L_1^\star \rfloor_{[t;\infty[}]\!]_{\text{tr}}$

◀