



HAL
open science

Decentralised Runtime Verification of Timed Regular Expressions

Victor Roussanaly, Yliès Falcone

► **To cite this version:**

Victor Roussanaly, Yliès Falcone. Decentralised Runtime Verification of Timed Regular Expressions. TIME 2022 - 29th International Symposium on Temporal Representation and Reasoning, Nov 2022, Online, France. pp.1-18, 10.4230/LIPIcs..12 . hal-03911668

HAL Id: hal-03911668

<https://inria.hal.science/hal-03911668>

Submitted on 23 Dec 2022

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

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

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

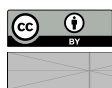
Digital Object Identifier 10.4230/LIPIcs...12

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



© V. Roussanaly and Y. Falcone;

licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

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

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

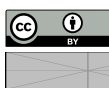
70 **1 Timed Words and Timed Regular Expressions**

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

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

80 **1.1 Timed Regular Expressions and Timed Regular Languages [3]**

81 We recall the syntax and semantics of timed words and timed regular expressions. Let Σ be
82 an alphabet. A timed word (called *time-event sequence* in [3]) over the alphabet Σ is a word
83 of $\mathbb{R}^+ \cup \Sigma$, composed of events in Σ and numerical values that represent delays, that is, the
84 time between two consecutive events. As such, two consecutive delays can be added, which
85 means that for two timed words u and v and for two delays x and y , $u \cdot x \cdot y \cdot v = u \cdot (x + y) \cdot v$.
86 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
87 word. We denote by $\mathcal{T}(\Sigma)$ the set of timed words over Σ and by ϵ the empty word. A subset
88 of $\mathcal{T}(\Sigma)$ is called *timed language*. A function $\theta : \Sigma_1 \rightarrow \Sigma_2 \cup \{\epsilon\}$ is called a *renaming*. We
89 consider its natural extensions $\Sigma_1^* \rightarrow \Sigma_2^*$ and $\mathcal{T}(\Sigma_1) \rightarrow \mathcal{T}(\Sigma_2)$, and we use the same symbol



90 θ to denote them.

91 We use the classical word concatenation denoted by \cdot , but we also need an *absorbing*
 92 *concatenation* denoted by the operator \circ . Let us denote by $\delta : \mathcal{T}(\Sigma) \rightarrow \mathbb{R}^+$ the function that
 93 returns the sum of delays in a timed word. For two timed words u and v , if there exists a
 94 word w such that $\delta(u) \cdot w = v$, then we can define $u \circ v = u \cdot w$. This means that $u \circ v$
 95 is defined if and only if v starts with a delay greater than the sum of delays in u , and if that is
 96 the case, then we remove this delay from the front of v before concatenating it to u . For
 97 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
 98 operators are extended to timed languages in the classical way. Moreover, for $n \in \mathbb{N}$ and
 99 $n \geq 2$, L^n and $L^{\circ n}$ respectively denote the language obtained by concatenating L with itself
 100 using operators \cdot and \circ , respectively; while $L^0 = L^{\circ 0} = \{\epsilon\}$.

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

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

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

110 **► Definition 2** (Semantics of timed regular expressions). *The semantics of a timed regular*
 111 *expression is the timed language defined inductively by function* $\llbracket \cdot \rrbracket : \mathcal{E}(\Sigma) \rightarrow \mathcal{P}(\mathcal{T}(\Sigma))$:

112	■ $\llbracket \epsilon \rrbracket = \{\epsilon\}$,	118	■ $\llbracket \langle L \rangle_I \rrbracket = \{u \in \llbracket L_1 \rrbracket \mid \delta(u) \in I\}$,
113	■ $\llbracket \underline{a} \rrbracket = \{r \cdot a \mid r \in \mathbb{R}^+\}$,	119	■ $\llbracket L^* \rrbracket = \bigcup_{i=0}^{\infty} \llbracket L^i \rrbracket$,
114	■ $\llbracket L_1 \wedge L_2 \rrbracket = \llbracket L_1 \rrbracket \cap \llbracket L_2 \rrbracket$,	120	■ $\llbracket L^{\otimes} \rrbracket = \bigcup_{i=0}^{\infty} \llbracket L^{\circ i} \rrbracket$,
115	■ $\llbracket L_1 \vee L_2 \rrbracket = \llbracket L_1 \rrbracket \cup \llbracket L_2 \rrbracket$,	121	■ $\llbracket \theta(L) \rrbracket = \{\theta(u) \mid u \in \llbracket L \rrbracket\}$.
116	■ $\llbracket L_1 \cdot L_2 \rrbracket = \llbracket L_1 \rrbracket \cdot \llbracket L_2 \rrbracket$,		
117	■ $\llbracket L_1 \circ L_2 \rrbracket = \llbracket L_1 \rrbracket \circ \llbracket L_2 \rrbracket$,		

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

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

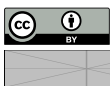
- 124 ■ $(\underline{a} \vee \underline{b})^* \cdot (\underline{a} \circ \langle b \rangle_{[0;1]}) \cdot (\underline{a} \vee \underline{b})^*$ denotes the language of timed words where at some point b
 125 occurs within one time unit after some a ;
- 126 ■ $\langle \underline{a} \rangle_{[0;1]}^*$ denotes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

130 Two timed regular expressions L and L' are *equivalent* if their language is the same, that is,
 131 $\llbracket L \rrbracket = \llbracket L' \rrbracket$.

132 1.2 Reduced Timed Regular Expressions

133 First, let us introduce reduced timed regular expressions.

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



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

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

- 140 ■ $\llbracket \langle L_1 \vee L_2 \rangle_I \rrbracket = \llbracket \langle L_1 \rangle_I \vee \langle L_2 \rangle_I \rrbracket$, 143 ■ $\llbracket \langle L_1 \wedge L_2 \rangle_I \rrbracket = \llbracket \langle L_1 \rangle_I \wedge \langle L_2 \rangle_I \rrbracket$,
- 141 ■ $\llbracket \langle L_1 \circ L_2 \rangle_I \rrbracket = \llbracket L_1 \circ \langle L_2 \rangle_I \rrbracket$, 144 ■ $\llbracket \langle L_1 \cdot L_2 \rangle_I \rrbracket = \llbracket L_1 \cdot L_2 \wedge \langle \Sigma^\otimes \rangle_I \rrbracket$,
- 142 ■ $\llbracket \theta(L) \rrbracket_I = \llbracket \theta(\langle L \rangle_I) \rrbracket$, 145 ■ $\llbracket \langle \epsilon \rangle_I \rrbracket = \{\epsilon\}$, if $0 \in I$, \emptyset otherwise,
- 146 ■ $\llbracket \langle L^\otimes \rangle_I \rrbracket = \llbracket \langle \epsilon \rangle_I \vee \langle L^\otimes \circ L \rangle_I \rrbracket = \llbracket \langle \epsilon \rangle_I \vee L^\otimes \circ \langle L \rangle_I \rrbracket$,
- 147 ■ $\llbracket \langle L^* \rangle_I \rrbracket = \llbracket \langle \epsilon \rangle_I \vee \langle L^* \cdot L \rangle_I \rrbracket = \llbracket \langle \epsilon \rangle_I \vee (L^* \cdot L \wedge \langle \Sigma^\otimes \rangle_I) \rrbracket$.

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

149 1.3 Semantics of Timed Regular Expressions over Timed Traces

150 In the context of decentralised monitoring, the monitor uses a trace as a sequence of time-
 151 stamped events. Formally, a *timed trace* over the alphabet Σ is a finite word over $\Sigma \times \mathbb{R}^+$
 152 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
 153 a sequence of events from Σ where each event is paired with the time at which it occurs. For
 154 example, $(a, 1.5) \cdot (b, 3.1) \cdot (a, 3.1) \cdot (c, 3.4)$ is a timed trace. Additionally, we also consider
 155 (ϵ, t) where ϵ is the empty word. Although this does not represent an observed event, it can
 156 be used to represent the absence of such an event. It can be simplified if there is an element
 157 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
 158 over Σ .

159 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. $\tau_{\text{last}}(\pi) =$
 160 t_n) the time at which the first (resp. last) event of π occurs. We denote by $L \downarrow_t$ the language
 161 represented by $\theta_\epsilon(\langle \underline{x} \rangle_{[t, t]})$ where θ_ϵ is the renaming that maps everything to ϵ . Intuitively,
 162 $L \downarrow_t$ is the language of timed words $\{t \cdot u \mid u \in \llbracket L \rrbracket\}$. Similarly, we define $L \uparrow^t$ by shifting all
 163 the time constraints that appear before the first concatenation (\cdot) in L by subtracting t . It
 164 can be defined inductively as such:

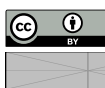
- 165 ■ $\underline{a} \uparrow^t = \underline{a}$, 170 ■ $(\langle L \rangle_I) \uparrow^t = \langle L \uparrow^t \rangle_{I-t}$
- 166 ■ $(L_1 \wedge L_2) \uparrow^t = L_1 \uparrow^t \wedge L_2 \uparrow^t$, 171 ■ $L^* \uparrow^t = L \uparrow^t \vee L^*$,
- 167 ■ $(L_1 \vee L_2) \uparrow^t = L_1 \uparrow^t \vee L_2 \uparrow^t$, 172 ■ $L^\otimes \uparrow^t = (L \uparrow^t)^\otimes$,
- 168 ■ $(L_1 \cdot L_2) \uparrow^t = L_1 \uparrow^t \cdot L_2$, 173 ■ $\theta(L) \uparrow^t = \theta(L \uparrow^t)$.
- 169 ■ $(L_1 \circ L_2) \uparrow^t = L_1 \uparrow^t \circ L_2 \uparrow^t$,

174 ► **Definition 6** (Semantics of timed regular expressions over timed traces). *The semantics of a*
 175 *timed regular expression is the set of timed traces defined inductively by function $\llbracket \cdot \rrbracket_{\text{tr}}$:*

- 176 ■ $\llbracket \langle \epsilon \rangle \rrbracket_{\text{tr}} = \{\epsilon\}$ 181 ■ $\llbracket \langle L \rangle_I \rrbracket_{\text{tr}} = \{\pi \in \llbracket L_1 \rrbracket_{\text{tr}} \mid \tau_{\text{last}}(u) \in I\}$
- 177 ■ $\llbracket \underline{a} \rrbracket_{\text{tr}} = \{(a, t) \mid t \in \mathbb{R}^+\}$ 182 ■ $\llbracket L^* \rrbracket_{\text{tr}} = \bigcup_{i=0}^{\infty} \llbracket L^i \rrbracket_{\text{tr}}$
- 178 ■ $\llbracket L_1 \wedge L_2 \rrbracket_{\text{tr}} = \llbracket L_1 \rrbracket_{\text{tr}} \cap \llbracket L_2 \rrbracket_{\text{tr}}$ 183 ■ $\llbracket L^\otimes \rrbracket_{\text{tr}} = \bigcup_{i=0}^{\infty} \llbracket L^{\circ i} \rrbracket_{\text{tr}}$
- 179 ■ $\llbracket L_1 \vee L_2 \rrbracket_{\text{tr}} = \llbracket L_1 \rrbracket_{\text{tr}} \cup \llbracket L_2 \rrbracket_{\text{tr}}$
- 180 ■ $\llbracket \theta(L) \rrbracket_{\text{tr}} = \{\theta(u) \mid u \in \llbracket (L) \rrbracket_{\text{tr}}\}$
- 184 ■ $\llbracket L_1 \circ L_2 \rrbracket_{\text{tr}} = \{u \cdot v \mid u \in \llbracket L_1 \rrbracket_{\text{tr}}, v \in \llbracket L_2 \rrbracket_{\text{tr}}\}$
- 185 ■ $\llbracket L_1 \cdot L_2 \rrbracket_{\text{tr}} = \{u \cdot v \mid u \in \llbracket L_1 \rrbracket_{\text{tr}}, v \in \llbracket L_2 \downarrow_{\tau_{\text{last}}(u)} \rrbracket_{\text{tr}}\}$

186 Let us denote by $\omega : \mathcal{R}(\Sigma) \mathcal{T}(\Sigma)$ the function that takes a timed trace $(\alpha_1, t_1) \cdot (\alpha_2, t_2) \cdots (\alpha_n, t_n)$
 187 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
 188 converts a timed trace into a timed word that represents the same sequence of events over
 189 physical time.

190 ► **Proposition 7.** *For $u \in \mathcal{R}(\Sigma)$ and $L \in \mathcal{E}(\Sigma)$, $u \in \llbracket L \rrbracket_{\text{tr}}$ if and only if $\omega(u) \in \llbracket L \rrbracket$.*



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

196 ► **Corollary 8.** For $u \in \mathcal{T}(\Sigma)$ and $L \in \mathcal{E}(\Sigma)$, $u \in \llbracket L \rrbracket$ if and only if there exists $v \in \mathcal{T}(\Sigma)$
 197 such that $v \sim u$ and $\omega^{-1}(v) \in \llbracket L \rrbracket_{\text{tr}}$.

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

200 1.4 Global operators

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

$$208 \quad \blacksquare \quad \llbracket [L]^I \rrbracket = \{u \in \llbracket L \rrbracket_{\text{tr}} \mid \tau_{\text{last}}(u) \in I\} \quad 209 \quad \blacksquare \quad \llbracket [L]_I \rrbracket = \{u \in \llbracket L \rrbracket_{\text{tr}} \mid \tau_{\text{first}}(u) \in I\}$$

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

212 ► **Proposition 9.** For all $L \in \mathcal{GE}(\Sigma)$, there exists $L' \in \mathcal{E}(\Sigma)$ such that $\llbracket L \rrbracket_{\text{tr}} = \llbracket L' \rrbracket_{\text{tr}}$.

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

216 ► **Example 10.** The expression $\langle a \rangle_{[0;3]} \cdot (\langle b \rangle_{[0;2]})^{\otimes} \cdot [(a \cdot b)^+]_{[4;5]}$ is semantically equivalent
 217 to $\langle \langle a \rangle_{[0;3]} \cdot (\langle b \rangle_{[0;2]})^{\otimes} \cdot a \cdot b \cdot (a \cdot b)^* \rangle_{[4;5]}$.

218 2 The Decentralised Timed Monitoring Problem

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

220 **Context and notations.** Let us suppose that the system at hand consists of n
 221 independent components denoted by C_i , for $0 < i \leq n$. Each component C_i emits local
 222 events over a local alphabet Σ_i . Let $\Sigma = \bigcup_{i \in [1, n]} \Sigma_i$ be the global alphabet of events. We
 223 assume that the local alphabets form a partition of Σ , which means that if $i \neq j$ then
 224 $\Sigma_i \cap \Sigma_j = \emptyset$. Let p_i be the projection of a timed trace of $\mathcal{R}(\Sigma)$ onto $\mathcal{R}(\Sigma_i)$ and denote by p
 225 the function defined by $p(\sigma) = (p_1(\sigma), p_2(\sigma), \dots, p_n(\sigma))$. As such, after observing our local
 226 traces $\sigma_1, \sigma_2, \dots, \sigma_n$, we can apply p^{-1} to obtain the possible global traces. We also note
 227 $\sigma|_t$ as the prefix of σ , which contains all events that occur before t . To each component C_i
 228 is attached a monitor M_i , which can observe a trace over Σ_i . We denote the verdict of a
 229 monitor by $\text{Verdict}_i : \mathbb{R} \rightarrow \{\text{bad}, \text{inconclusive}\}$. Monitors are purposed to detect violations.

230 **Assumptions.** Our assumptions on the system are as follows.



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

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

239 ► **Definition 11 (Definitive Verdict).** *If there is a time t for which there is a verdict*
 240 $\text{Verdict}_i(t) = \mathbf{bad}$, *then for all $t' \geq t$, $\text{Verdict}_i(t') = \mathbf{bad}$.*

241 ► **Definition 12 (Soundness).** *If there is a time t for which there is a verdict $\text{Verdict}_i(t) = \mathbf{bad}$*
 242 *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 $\sigma' \notin \llbracket L \rrbracket_{\text{tr}}$.*

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

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

250 **3 Progression for Timed Regular Expressions**

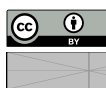
251 We consider decentralised monitors that observe only their local events, indexed with the
 252 time at which they happen. For a monitor of index i , we define a *decentralised progression*
 253 *function* $\text{Pr}_i : \mathcal{GE}(\Sigma) \rightarrow (\Sigma \times \mathbb{R}^+) \rightarrow \mathcal{GE}(\Sigma)$,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

258 ► **Definition 14 (Centralised progression function).** $\text{Pr}_0 : \mathcal{GE}(\Sigma) \rightarrow (\Sigma \times \mathbb{R}^+) \rightarrow \mathcal{GE}(\Sigma)$ *is a*
 259 *centralised progression function iff for any expression $L \in \mathcal{GE}(\Sigma)$ and any trace event (α, t) ,*
 260 $\text{Pr}_0(L)(\alpha, t) = \{u \mid (t \cdot \alpha) \circ u \in L\}$.

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

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

¹ Note that we do not consider the alternative problem of determining that the trace will always be
 in $\llbracket L \rrbracket_{\text{tr}}$ 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.



270 The above function is not a left derivative, as it allows events that are not from Σ_i
 271 to occur before the observed event. We now propose a decentralised progression function
 272 and detail its inductive definition. To define such a function, we define a filter function
 273 $\Phi : \mathcal{GE}(\Sigma) \rightarrow \mathcal{P}(\Sigma) \rightarrow \mathcal{GE}(\Sigma)$. The proper definition is given in the appendix. This function
 274 is defined such that it removes events of a given alphabet from an expression.

275 **► Proposition 16.** For $L \in \mathcal{GE}(\Sigma)$ and $\Sigma' \subseteq \Sigma$, $\llbracket \Phi(L)(\Sigma') \rrbracket = \llbracket L \wedge (\Sigma \setminus \Sigma')^* \rrbracket$.

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

- | | |
|---|--|
| 279 ■ $\text{Pr}_i(\underline{a})(\alpha, t) = \epsilon_t$, if $\alpha = a$
280 ■ $\text{Pr}_i(\langle \underline{a} \rangle_I)(\alpha, t) = \emptyset$, if $t \notin I$ | 281 ■ $\text{Pr}_i(\underline{a})(\alpha, t) = \emptyset$, if $\alpha \neq a$
282 ■ $\text{Pr}_i(\langle \underline{a} \rangle_I)(\alpha, t) = \text{Pr}_i(\underline{a})(\alpha, t)$, if $t \in I$ |
| 283 ■ $\text{Pr}_i(L_1 \vee L_2)(\alpha, t) = \text{Pr}_i(L_1)(\alpha, t) \vee \text{Pr}_i(L_2)(\alpha, t)$ | |
| 284 ■ $\text{Pr}_i(L_1 \wedge L_2)(\alpha, t) = \text{Pr}_i(L_1)(\alpha, t) \wedge \text{Pr}_i(L_2)(\alpha, t)$ | |

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

- | | |
|---|--|
| 287 ■ $\text{Pr}_i(L_1 \circ L_2)(\alpha, t) = (\text{Pr}_i(L_1)(\alpha, t) \circ [L_2]_{[t; \infty[}) \vee [\Phi(L_1)(\Sigma_i)]^{[0; t[} \circ \text{Pr}_i(L_2)(\alpha, t)$ | 288 ■ $\text{Pr}_i(L^{\otimes})(\alpha, t) = [\Phi(L)(\Sigma_i)^{\otimes}]^{[0; t[} \circ \text{Pr}_i(L)(\alpha, t) \circ [L^{\otimes}]_{[t; \infty[}$ |
|---|--|

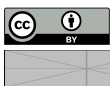
289 For the absorbing concatenation \circ , we examine whether (α, t) corresponds to an event on
 290 each side of the concatenation. If (α, t) is an event on the left side, then events on the right
 291 side must have occurred later. Otherwise, if (α, t) is an event on the right side, then the
 292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

295 Let us consider the progression for the global operators. For these cases, we use additional
 296 notation. First, for I an interval and $t \in \mathbb{R}$, we write $t < I$ (resp. $t > I$) if and only if for all
 297 $t' \in I$, $t < t'$ (resp. $t > t'$). We also denote by $I \downarrow$ the interval obtained by changing the lower
 298 bound of I to 0 inclusive, and by $I \uparrow$ the interval obtained by changing the upper bound of I
 299 to ∞ . Hence, we have:

- | | |
|---|---|
| 300 ■ $\text{Pr}_i([L]^I)(\alpha, t) = [\text{Pr}_i(L)(\alpha, t)]^{I \downarrow}$, if $t \in I$ | 301 ■ $\text{Pr}_i([L]_I)(\alpha, t) = [\text{Pr}_i(L)(\alpha, t)]_{I \uparrow}$, if $t \in I$ |
| 302 ■ $\text{Pr}_i([L]^I)(\alpha, t) = \emptyset$, otherwise | 303 ■ $\text{Pr}_i([L]_I)(\alpha, t) = \emptyset$, otherwise |

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

311 Finally, let us consider the non-absorbing concatenation. We want something similar to \circ
 312 for \cdot where we consider whether the event occurred on the right or left side. The problem
 313 here is that if the event happened on the right-hand side, then the result of our progression
 314 function depends on the time of the last event on the left-hand side since the concatenation
 315 “resets” the time at which we interpret the expression. Here we use the fact that the function
 316 $t' \rightarrow \text{Pr}_i(L \downarrow_{t'}) (\alpha, t)$ is a piece-wise constant function. This means that we can look at a
 317 finite number of possible intervals at which the last element of the left-hand side occurs. By
 318 building those intervals and computing the expression associated with each one, we can build



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

$$321 \blacksquare \Pr_i(L_1 \cdot L_2)(\alpha, t) = \bigvee_{(I, L') \in \Delta_i(L_2, \mathbb{R}^+)(\alpha, t)} [\Phi(L_1)(\Sigma_i)]^I \cdot L' \vee \Pr_i(L_1)(\alpha, t) \cdot \lfloor L_2 \rfloor_{[t; \infty[}$$

$$322 \blacksquare \Pr_i(L^*)(\alpha, t) = \bigvee_{(I, L') \in \Delta_i(L, \mathbb{R}^+)(\alpha, t)} [\Phi(L)(\Sigma_i)^*]^I \cdot L' \cdot \lfloor L^* \rfloor_{[t; \infty[}$$

$$323 \blacksquare \Pr_i(\theta(L))(\alpha, t) = \bigvee_{\alpha' \in u^{-1}(\alpha)} \theta(\Pr_i(L)(\alpha', t))$$

324 **► Example 17.** Let us consider the language $L = (\underline{a} \vee \underline{b})^{\otimes} \cdot \langle \underline{a} \rangle_{[0;1]}$ with $\Sigma_1 = \{a\}$, $\Sigma_2 = \{b\}$.
 325 Then $\Pr_1(L)(a, 2) = (\lfloor \underline{b}^{\otimes} \rfloor_{[0;2]} \circ \lfloor (\underline{a} \vee \underline{b})^{\otimes} \cdot \langle \underline{a} \rangle_{[0;1]} \rfloor_{[2; \infty[}) \vee \lfloor \underline{b}^{\otimes} \rfloor_{[1;2]}$.

326 This means that, either the a observed was on the left of the concatenation, and in that case
 327 we have the language defined by L where before the global time $t = 2$, there can only be b
 328 events. Or the a observed was the one on the right side, so we can only observe events b that
 329 preceded it, with the last b between the global time $t = 1$ and $t = 2$.

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

332 **► 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
 333 $L \in \mathcal{GE}(\Sigma)$, $\llbracket \Pr_j((\Pr_i(L)(a, t_a))(b, t_b)) \rrbracket_{\text{tr}} = \llbracket \Pr_i((\Pr_j(L)(b, t_b))(a, t_a)) \rrbracket_{\text{tr}}$.

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

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

343 **► Corollary 19.** For all $L \in \mathcal{GE}(\Sigma)$, for all π, π' such that for all i , $p_i(\pi) = p_i(\pi')$,
 344 $\llbracket \Pr^*(L)(\pi) \rrbracket_{\text{tr}} = \llbracket \Pr^*(L)(\pi') \rrbracket_{\text{tr}}$

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

347 4 Decentralised Monitoring

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

352 4.1 Simulating Centralised Monitoring

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

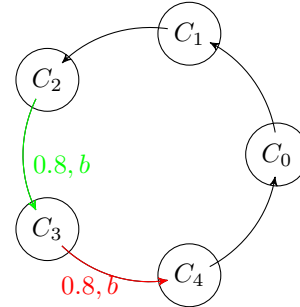


```

1  $L_i$  := specification for the system
2 when an internal event  $(t, \alpha)$  is observed do
3   | Send  $(C_i, \alpha, t)$  to  $C_{i+1}$ 
4 when a message  $(C, \alpha, t)$  is received do
5   | if  $k \neq i$  then
6     |   add  $(\alpha, t)$  to the memory
7     |   Send  $(C, \alpha, t)$  to  $C_{i+1}$ 
8   | else
9     |   foreach  $(\alpha', t')$  in the memory s.t.
10    |      $t' \leq t$  (ordered by ascending  $t'$ ) do
11    |     |  $L_i := \text{Pr}_0(L)(\alpha', t')$ 
12    |     | Remove  $(\alpha', t')$  from memory
13    |     |  $L_i := \text{Pr}_0(L)(\alpha, t)$ 
14    |     | if  $\llbracket L_i \rrbracket_{\text{tr}} = \emptyset$  then
15    |     |   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

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

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

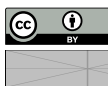
367 ► **Proposition 20.** *If C_i observes (α, t) and $C_{i'}$ observes (α', t') with $t < t'$, then $C_{i'}$ receives*
368 *the message (C_i, α, t) before $(C_{i'}, \alpha', t')$.*

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

373 ► **Theorem 21.** *Let $\sigma \in \mathcal{R}(\Sigma)$ be the global timed trace, L the initial expression, and let*
374 *(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) ,*
375 *we have $\llbracket L_i \rrbracket_{\text{tr}} = \{\sigma' \in \mathcal{R}(\Sigma) \mid \sigma \cdot \sigma' \in L\}$.*

376 Emptiness checking

377 We know that, if at some point $\llbracket L_i \rrbracket_{\text{tr}} = \emptyset$ for some i , then the specification is violated.
378 This entails testing the emptiness of an expression. One way to do so is to build a timed
379 automaton that recognise the same language, using the construction shown in [3], as we know
380 that the problem of knowing whether or not the language of a timed automaton is empty is
381 PSPACE [1] and there are several tools that solve that problem. While this construction
382 ensures an emptiness check, it is costly, and it is desirable to avoid it at runtime. Instead,
383 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   |  $L_i := \text{specification}$ 
3 else
4   |  $L_i := \text{waiting}$ 
5 when an internal event  $(\alpha, t)$  is observed do
6   | if  $L_i = \text{waiting}$  then
7     | add  $(\alpha, t)$  to the memory
8   | else
9     |  $L_i := \text{Pr}_i(L)(e, t)$ 
10    | if  $\llbracket L_i \rrbracket_{\text{tr}} = \emptyset$  then
11    | return bad verdict
12 when an expression  $L$  is received do
13   |  $L_i = L$  foreach  $(\alpha, t)$  in the memory do
14     |  $L_i := \text{Pr}_i(L)(\alpha, t)$ 
15     | Remove  $(\alpha, t)$  from the memory
16     | if  $\llbracket L_i \rrbracket_{\text{tr}} = \emptyset$  then
17     | return bad verdict
18 when  $\text{urgent}(L_i)$  do
19   | send  $L_i$  to  $\text{target}(L_i)$ 
20   |  $L_i := \text{waiting}$ 

```

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

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

394 4.2 Using Decentralised Progression

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



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

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

425 **5 Simulation and Benchmarks**

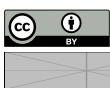
426 In this section, we validate the effectiveness of our decentralised monitoring approaches by
427 showing the benefits over an approach that simulates centralised monitoring. For this, we
428 first briefly describe the implementation of our monitoring algorithms (Sec. 5.1). Then,
429 we detail our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

431 **5.1 Implementation as an OCaml Benchmark**

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

442 **5.2 Experimental Setup**

443 We test and compare the three approaches for decentralised monitoring of timed regular
444 expressions: (i) simulating centralised, (ii) decentralised with a dynamic target, and (iii)
445 decentralised with a static ring target. We consider a network of 10 monitors, each of them
446 capable of observing a different event. We consider the communication delay when a message
447 is sent as a random value between 0 and 40 units of time. Timed regular expressions are
448 not commonly used, since people prefer timed automata that are as expressive as timed
449 regular expressions. This means that one approach could have been to take a benchmark
450 of timed automata, compute a set of equivalent expressions and use them as a benchmark.
451 But this poses a problem as the performance would be affected by our choice of equivalent
452 expression. Another difficulty is the absence of reference benchmark for timed decentralised
453 monitoring. That is why in this context, we choose to generate random timed regular
454 expressions. We generate 100 expressions of a fixed size, for each size between 2 and 8. For
455 the time constraints that may appear, we randomly chose the lower bound between 0 and
456 30, and the upper bound is ∞ or a sum between the lower bound and an integer chosen
457 randomly between 0 and 30. For each of these expressions, we generate 100 timed traces of
458 length 200, with a delay between two consecutive events chosen randomly between 0 and 10.
459 Note that a randomly generated trace will most likely not respect a random specification,
460 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

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

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

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

470 ■ Communication (messages). We measure the total number and size of the messages
 471 exchanged by the monitor. Since there are between 8 and 16 operators, they are encoded
 472 over 4 bits. The time values in timed constraints are encoded over 32 bits. There are 10
 473 possible events encoded over 4 bits

474 ■ Computation (progressions). We measure the total number of calls to the progression
 475 functions defined in the previous sections, including the recursive calls.

476 ■ Delay (decision time). We measure the time it takes for one of the monitors to detect
 477 that the current global trace violates the monitored timed regular expression.

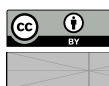
478 Note that here we do not consider the computation time associated with each progression
 479 computation. We consider that this time should be negligible compared to the communication
 480 delays.

481 5.3 Results and Discussion

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

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

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



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

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

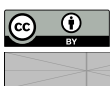
508 Another idea would be to send either a history of the timed events or an expression
509 depending on which one is smaller. Indeed, each monitor could compute the expression if it
510 received some history of the timed events. This approach is hard to implement, as it would
511 mean that monitors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

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

525 **6 Related Work**

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

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



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

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

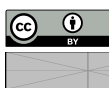
561 **7 Conclusion and Perspectives**

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

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

580 **References**

- 581 1 R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *[1990]*
582 *Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425,
583 1990.
- 584 2 Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*,
585 126(2):183–235, 1994.
- 586 3 Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–
587 206, 2002.
- 588 4 Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime
589 verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification -*
590 *Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages
591 1–33. Springer, 2018.



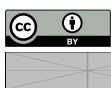
© V. Roussalany and Y. Falcone;

licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- 592 **5** Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and
593 TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.
- 594 **6** Andreas Klaus Bauer and Yliès Falcone. Decentralised LTL monitoring. In Dimitra
595 Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International
596 Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in
597 Computer Science*, pages 85–100. Springer, 2012.
- 598 **7** Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with
599 a global clock. *Formal Methods Syst. Des.*, 49(1-2):109–158, 2016.
- 600 **8** Daniel de Leng and Fredrik Heintz. Approximate stream reasoning with metric temporal
601 logic under uncertainty. In *Proceedings of the Thirty-Third AAAI Conference on
602 Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence
603 Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence,
604 AAAI’19/IAAI’19/EAAI’19*. AAAI Press, 2019.
- 605 **9** Antoine El-Hokayem and Yliès Falcone. On the monitoring of decentralized specifications:
606 Semantics, properties, analysis, and simulation. *ACM Trans. Softw. Eng. Methodol.*, 29(1):1:1–
607 1:57, 2020.
- 608 **10** Yliès Falcone. On decentralized monitoring. In Ayoub Nouri, Weimin Wu, Kamel Barkaoui,
609 and ZhiWu Li, editors, *Verification and Evaluation of Computer and Communication Systems
610 - 15th International Conference, VECoS 2021, Virtual Event, November 22-23, 2021, Revised
611 Selected Papers*, volume 13187 of *Lecture Notes in Computer Science*, pages 1–16. Springer,
612 2021.
- 613 **11** Yliès Falcone, Tom Cornebize, and Jean-Claude Fernandez. Efficient and generalized
614 decentralized monitoring of regular languages. In Erika Ábrahám and Catuscia Palamidessi,
615 editors, *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG
616 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated
617 Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5,
618 2014. Proceedings*, volume 8461 of *Lecture Notes in Computer Science*, pages 66–83. Springer,
619 2014.
- 620 **12** Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred
621 Broy, Doron A. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*,
622 volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication
623 Security*, pages 141–175. IOS Press, 2013.
- 624 **13** Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying
625 runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 23(2):255–284, 2021.
- 626 **14** Florian Gallay and Yliès Falcone. Decentralized LTL enforcement. In Pierre Ganty and Davide
627 Bresolin, editors, *Proceedings 12th International Symposium on Games, Automata, Logics,
628 and Formal Verification, GandALF 2021, Padua, Italy, 20-22 September 2021*, volume 346 of
629 *EPTCS*, pages 135–151, 2021.
- 630 **15** Alejandro Grez, Filip Mazowiecki, Michal Pilipczuk, Gabriele Puppis, and Cristian Riveros.
631 The monitoring problem for timed automata. *CoRR*, abs/2002.07049, 2020.
- 632 **16** Dejan Nickovic and Oded Maler. AMT: a property-based monitoring tool for analog systems.
633 In Jean-François Raskin and P. S. Thiagarajan, editors, *Proceedings of the 5th International
634 Conference on Formal modeling and analysis of timed systems (FORMATS 2007)*, volume
635 4763 of *Lecture Notes in Computer Science*, pages 304–319. Springer-Verlag, 2007.
- 636 **17** Srinivas Pinisetty, Thierry Jéron, Stavros Tripakis, Yliès Falcone, Hervé Marchand, and Viorel
637 Preoteasa. Predictive runtime verification of timed properties. *J. Syst. Softw.*, 132:353–365,
638 2017.
- 639 **18** Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic
640 specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.
- 641 **19** Stavros Tripakis. Decentralized observation problems. In *44th IEEE IEEE Conference on
642 Decision and Control and 8th European Control Conference Control, CDC/ECC 2005, Seville,
643 Spain, 12-15 December, 2005*, pages 6–11. IEEE, 2005.



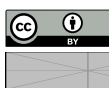
- 644 20 Stavros Tripakis and Karen Rudie. Decentralized observation of discrete-event systems: At
645 least one can tell. *IEEE Control. Syst. Lett.*, 6:1652–1657, 2022.
- 646 21 Dogan Ulus. Montre: A tool for monitoring timed regular expressions. In Rupak Majumdar
647 and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference,*
648 *CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of
649 *Lecture Notes in Computer Science*, pages 329–335. Springer, 2017.
- 650 22 Yin Wang, Tae-Sic Yoo, and Stéphane Lafortune. Diagnosis of discrete event systems using
651 decentralized architectures. *Discret. Event Dyn. Syst.*, 17(2):233–263, 2007.
- 652 23 Xiang Yin and Stéphane Lafortune. Codiagnosability and coobservability under dynamic
653 observations: Transformation and verification. *Autom.*, 61:241–252, 2015.

654 A Definition of Φ

- 655 ■ $\Phi(\underline{a})(\Sigma') = \underline{a}$ if $a \notin \Sigma'$ 660 ■ $\Phi(\underline{a})(\Sigma') = \emptyset$ if $a \in \Sigma'$
- 656 ■ $\Phi(\langle \underline{a} \rangle_I)(\Sigma') = \langle \underline{a} \rangle_I$ if $a \notin \Sigma'$ 661 ■ $\Phi(\langle \underline{a} \rangle_I)(\Sigma') = \emptyset$ if $a \in \Sigma'$
- 657 ■ $\Phi(L^\otimes)(\Sigma') = \Phi(L)(\Sigma')^\otimes$ 662 ■ $\Phi(L^*)(\Sigma') = \Phi(L)(\Sigma')^*$
- 658 ■ $\Phi(\lceil L \rceil^I)(\Sigma') = \lceil \Phi(L)(\Sigma') \rceil^I$ 663 ■ $\Phi(\lfloor L \rfloor_I)(\Sigma') = \lfloor \Phi(L)(\Sigma') \rfloor_I$
- 659 ■ $\Phi(\theta(L))(\Sigma') = \theta(\Phi(L)(\theta^{-1}(\Sigma')))$
- 664 ■ $\Phi(L_1 \vee L_2)(\Sigma') = \Phi(L_1)(\Sigma') \vee \Phi(L_2)(\Sigma')$
- 665 ■ $\Phi(L_1 \wedge L_2)(\Sigma') = \Phi(L_1)(\Sigma') \wedge \Phi(L_2)(\Sigma')$
- 666 ■ $\Phi(L_1 \circ L_2)(\Sigma') = \Phi(L_1)(\Sigma') \circ \Phi(L_2)(\Sigma')$
- 667 ■ $\Phi(L_1 \cdot L_2)(\Sigma') = \Phi(L_1)(\Sigma') \cdot \Phi(L_2)(\Sigma')$

668 B Definition of Δ_i

- 669 For I, I' two intervals, we denote $I \triangleleft I' = \{x \in I \mid \forall t \in I', x < t\}$.
- 670 And $I \triangleright I' = \{x \in I \mid \forall t \in I', x > t\}$ such that $I \triangleleft I', I \triangleright I'$ and $I \cap I'$ are always a
671 partition of I . We define Δ_i as follows:
- 672 ■ $\Delta_i(\underline{a}, I)(\alpha, t) = \{(I, \epsilon)\}$ if $\alpha = a$.
- 673 ■ $\Delta_i(\underline{a}, I)(\alpha, t) = \{(I, \emptyset)\}$ otherwise.
- 674 ■ $\Delta_i(\langle \underline{a} \rangle_{I'}, I)(\alpha, t) = \{(I \triangleright I', \emptyset); (I \triangleleft I', \emptyset); (t - I' \cap I, \epsilon)\}$ if $\alpha = a$.
- 675 ■ $\Delta_i(\langle \underline{a} \rangle_{I'}, I)(\alpha, t) = \{(\emptyset, I)\}$ otherwise.
- 676 ■ $\Delta_i(L_1 \vee L_2, I)(\alpha, t) =$
677 $\{(I_1 \cap I_2, L_1' \vee L_2') \mid (I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t), (I_2, L_2') \in \Delta_i(L_2, I)(\alpha, t)\}$
- 678 ■ $\Delta_i(L_1 \wedge L_2, I)(\alpha, t) =$
679 $\{(I_1 \cap I_2, L_1' \wedge L_2') \mid (I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t), (I_2, L_2') \in \Delta_i(L_2, I)(\alpha, t)\}$
- 680 ■ $\Delta_i(L_1 \circ L_2, I)(\alpha, t) = \{(I_1 \cap I_2, L_1' \circ \lfloor L_2' \rfloor_{[t; \infty[} \vee \lceil \Phi(L_1)(\Sigma_i) \rceil^{[0; t]}) \mid (I_1, L_1') \in$
681 $\Delta_i(L_1, I)(\alpha, t), (I_2, L_2') \in \Delta_i(L_2, I)(\alpha, t)\}$
- 682 ■ $\Delta_i(L_1^\otimes, I)(\alpha, t) =$
683 $\{(I_1, \lceil Past(L_1)(\Sigma_i)^\otimes \rceil^{[0; t]} \circ L_1' \circ \lfloor L_1' \rfloor_{[t; \infty[} \mid (I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t)\}$
- 684 ■ $\Delta_i(L_1 \cdot L_2, I)(\alpha, t) =$
685 $\{(I_1, (\bigvee_{(I_2, L_2') \in \Delta_i(L_2, I)(\alpha, t)} \lceil \Phi(L_1)(\Sigma_i) \rceil \cdot L_2')^{I_2} \vee L_1' \circ \lfloor L_2' \rfloor_{[t; \infty[} \mid (I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t)\}$
- 686 ■ $\Delta_i(L_1^*, I)(\alpha, t) = \{(I_1, \bigvee_{(I_1, L_1') \in \Delta_i(L_1, I)(\alpha, t)} \lceil \Phi(L_1)(\Sigma_i) \rceil^{I_1} \cdot L_1' \cdot \lfloor L_1' \rfloor_{[t; \infty[})\}$



687 This function satisfies the following properties.

$$688 \quad \forall t, I, L, \forall (I', L') \in \Delta_i(L, I)(\alpha, t), \forall t' \in I', L' = \text{Pr}_i(L \downarrow_{t'})(\alpha, t)$$

$$689 \quad \forall t, I, L, \forall (I_1, L_1), (I_2, L_2) \in \Delta_i(L, I)(\alpha, t), (I_1, L_1) \neq (I_2, L_2) \Leftrightarrow I_1 \cap I_2 = \emptyset$$

$$690 \quad \forall t, I, L, \bigcup_{(I', L') \in \Delta_i(L, I)(\alpha, t)} I' = I$$

692 **C Proof that Pr_i is a decentralised progression function**

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

695 ■ Let $L = L_1 \circ L_2$. Let us prove the double inclusion.

696 1. Assume $u \in \llbracket \text{Pr}_i(L)(\alpha, t) \rrbracket_{\text{tr}}$. There are then two possible cases.

697 ■ First case: $u \in \llbracket \text{Pr}_i(L_1)(\alpha, t) \circ [L_2]_{[t; \infty]} \rrbracket_{\text{tr}}$ which means that $u = u_1 \cdot u_2$ with
698 $u_1 \in \llbracket \text{Pr}_i(L_1)(\alpha, t) \rrbracket_{\text{tr}}$ and $u_2 \in \llbracket L_2 \rrbracket_{\text{tr}}$ and $\tau_{\text{first}}(u_2) \geq t$. Using our induction
699 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
700 proved $v_1 \cdot (\alpha, t) \cdot v_2 \cdot u_2 \in \llbracket L \rrbracket_{\text{tr}}$.

701 ■ Second case $u \in \llbracket [\Phi(L_1)(\Sigma_i)]^{[0; t]} \circ \text{Pr}_i(L_2)(\alpha, t) \rrbracket_{\text{tr}}$, then it means that $u = u_1 \cdot u_2$
702 with $u_1 \in \llbracket \Phi(L_1)(\Sigma_i) \rrbracket_{\text{tr}}$ and $u_2 \in \llbracket \text{Pr}_i(L_2)(\alpha, t) \rrbracket_{\text{tr}}$. This means that $u_1 \in \llbracket L_1 \rrbracket_{\text{tr}} \cap$
703 $\mathcal{R}(\Sigma \setminus \Sigma_i)$. Using our induction hypothesis, we have $u_2 = v_1 \cdot v_2$ with $v_1 \cdot (\alpha, t) \cdot v_2 \in$
704 $\llbracket L_2 \rrbracket_{\text{tr}}$ and $v_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$. We can then deduce $u_1 \cdot v_1 \cdot (\alpha, t) \cdot v_2 \in \llbracket L \rrbracket_{\text{tr}}$ and
705 $(u_1 \cdot v_1) \in \mathcal{R}(\Sigma \setminus \Sigma_i)$

706 This means that in both cases $u \in \{u' \cdot v' \mid u' \cdot (\alpha, t) \cdot v' \in \llbracket L \rrbracket_{\text{tr}} \text{ and } u' \in \mathcal{R}(\Sigma \setminus \Sigma_i)\}$

707 2. Let us prove the other side of the inclusion. Let us assume u and v such that
708 $u \cdot (\alpha, t) \cdot v \in \llbracket L \rrbracket_{\text{tr}} \wedge u \in \mathcal{R}(\Sigma \setminus \Sigma_i)$. This means that $u \cdot (\alpha, t) \cdot v = u' \cdot v'$ with
709 $u' \in \llbracket L_1 \rrbracket_{\text{tr}}$ and $v' \in \llbracket L_2 \rrbracket_{\text{tr}}$. We can then deduce that we either have $u' = u \cdot (\alpha, t) \cdot v_1$
710 or have $v' = u_1 \cdot (\alpha, t) \cdot v$. In other words, we have either $u \cdot v_1 \in \llbracket \text{Pr}_i(L_1)(\alpha, t) \rrbracket_{\text{tr}}$
711 with $v' \in \llbracket [L_2]_{[t; \infty]} \rrbracket_{\text{tr}}$ or $u_1 \cdot v \in \llbracket \text{Pr}_i(L_2)(\alpha, t) \rrbracket_{\text{tr}}$ with $u' \in L_1$ and in the last
712 case, every element before (α, t) cannot contain elements of Σ_i , therefore it must be
713 true for u' which means $u' \in \llbracket [\Phi_i(L_1)(\Sigma_i)]^{[0; t]} \rrbracket_{\text{tr}}$. This means that in both cases
714 $u \cdot v \in \llbracket \text{Pr}_i(L)(\alpha, t) \rrbracket_{\text{tr}}$

715 ■ If $L = L_1^{\otimes}$.

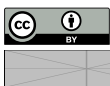
716 1. Let us assume $u \in \llbracket \text{Pr}_i(L)(\alpha, t) \rrbracket_{\text{tr}}$. This means that $u = u_1 \cdot u_2 \cdot u_3$ with $u_1 \in$
717 $\llbracket [\Phi(L_1)(\Sigma_i)^{\otimes}]^{[0; t]} \rrbracket_{\text{tr}}$, $u_3 \in \llbracket [L_1^{\otimes}]_{[t; \infty]} \rrbracket_{\text{tr}}$ and $u_2 \in \llbracket \text{Pr}_1(L_1)(\alpha, t) \rrbracket_{\text{tr}}$. Therefore, we
718 know that $u_1 \in \llbracket L_1^{\otimes} \rrbracket_{\text{tr}}$, $u_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$, and $u_2 = v_1 \cdot v_2$ with $v_1 \cdot (\alpha, t) \cdot v_2 \in \llbracket L_1 \rrbracket_{\text{tr}}$.
719 Hence $u_1 \cdot v_1 \cdot (\alpha, t) \cdot v_2 \cdot v_3 \in \llbracket L_1^{\otimes} \rrbracket_{\text{tr}}$ with $u_1 \cdot v_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$

720 2. Now to prove the other inclusion. Let us assume $u \cdot v$ such that $w = u \cdot (\alpha, t) \cdot v \in \llbracket L \rrbracket_{\text{tr}}$.
721 Since w is not empty, it is equal to $w_1 \cdot w_2 \cdot \dots \cdot w_n$, with $w_i \in \llbracket L_1 \rrbracket_{\text{tr}}$. Because
722 we know that w contains (α, t) , we can denote by k the index of the w_i containing
723 this event. In other words, we have $w_k = u' \cdot (\alpha, t) \cdot v' \in \llbracket L_1 \rrbracket_{\text{tr}}$, which means that
724 $u' \cdot v' \in \llbracket \text{Prog}_i(L_1)(\alpha, t) \rrbracket_{\text{tr}}$. This proves that $u \cdot v = w_0 \cdot \dots \cdot w_{k-1} \cdot u' \cdot v' \cdot w_{k+1} \cdot \dots \cdot w_n \in$
725 $\llbracket [\Phi(L_1)(\Sigma_i)^{\otimes}]^{[0; t]} \circ \text{Pr}_i(L_1)(\alpha, t) \circ [L_1^{\otimes}]_{[t; \infty]} \rrbracket_{\text{tr}}$

726 ■ Let $L = L_1 \cdot L_2$. Let us prove the double inclusion.

727 1. Assume $u \in \llbracket \text{Pr}_i(L)(\alpha, t) \rrbracket_{\text{tr}}$. There are then two possible cases.

728 ■ First case: $u \in \llbracket \text{Pr}_i(L_1)(\alpha, t) \circ [L_2]_{[t; \infty]} \rrbracket_{\text{tr}}$ then means that $u = u_1 \cdot u_2$ with
729 $u_1 \in \llbracket \text{Pr}_i(L_1)(\alpha, t) \rrbracket_{\text{tr}}$ and $u_2 \in \llbracket L_2 \downarrow_{\tau_{\text{last}}(u_1)} \rrbracket_{\text{tr}}$ and $\tau_{\text{first}}(u_2) \geq t$. Using our
730 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
731 allows us to conclude that $v_1 \cdot (\alpha, t) \cdot v_2 \cdot u_2 \in \llbracket L \rrbracket_{\text{tr}}$.



732 **Second case:** There is $(I, L') \in \Delta_i(L_2, \mathbb{R}^+)(\alpha, t)$ such that $u \in \llbracket [\Phi(L_1)(\Sigma_i)]^I \cdot$
733 $L'(\alpha, t) \rrbracket_{\text{tr}}$. It means that $u = u_1 \cdot u_2$ with $u_1 \in \llbracket \Phi(L_1)(\Sigma_i) \rrbracket_{\text{tr}}$ and $u_2 \in \llbracket L'(\alpha, t) \rrbracket_{\text{tr}}$.
734 This means that $u_1 \in \llbracket L_1 \rrbracket_{\text{tr}} \cap \mathcal{R}(\Sigma \setminus \Sigma_i)$. Taking into account the first property of Δ_i ,
735 we can also see that $u_2 \in \llbracket \text{Pr}_i(L_2 \downarrow_{\tau_{\text{last}}(u_1)})(\alpha, t) \rrbracket_{\text{tr}}$. Using our induction hypothesis,
736 we have $u_2 = v_1 \cdot v_2$ with $v_1 \cdot (\alpha, t) \cdot v_2 \in \llbracket L_2 \downarrow_{\tau_{\text{last}}(u_1)} \rrbracket_{\text{tr}}$ and $v_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$.
737 Therefore, we have $u_1 \cdot v_1 \cdot (\alpha, t) \cdot v_2 \in \llbracket L \rrbracket_{\text{tr}}$ and $(u_1 \cdot v_1) \in \mathcal{R}(\Sigma \setminus \Sigma_i)$.

738 **2.** Let us prove the other side of the inclusion. Assume u and v such that $u \cdot (\alpha, t) \cdot v \in$
739 $\llbracket L \rrbracket_{\text{tr}} \wedge u \in \mathcal{R}(\Sigma \setminus \Sigma_i)$. This means that $u \cdot (\alpha, t) \cdot v = u' \cdot v'$ with $u' \in \llbracket L_1 \rrbracket_{\text{tr}}$ and
740 $v' \in \llbracket L_2 \downarrow_{\tau_{\text{last}}(u')} \rrbracket_{\text{tr}}$. We can deduce that we have $u' = u \cdot (\alpha, t) \cdot v_1$ or we have
741 $v' = u_1 \cdot (\alpha, t) \cdot v$.

742 In other words, we have $u \cdot v_1 \in \llbracket \text{Pr}_i(L_1)(\alpha, t) \rrbracket_{\text{tr}}$ with $v' \in \llbracket [L_2]_{[t, \infty]} \rrbracket_{\text{tr}}$ or we have
743 $u_1 \cdot v \in \llbracket \text{Pr}_i(L_2 \downarrow_{\tau_{\text{last}}(u')})(\alpha, t) \rrbracket_{\text{tr}}$ with $u' \in L_1$. In the first case, we have $u \cdot v \in$
744 $\llbracket \text{Pr}_i(L)(\alpha, t) \rrbracket_{\text{tr}}$.

745 Let us look at the other case. Every element before (α, t) cannot contain elements
746 of Σ_i , therefore it must be true of u' , which means $u' \in \llbracket \Phi_i(L_1)(\Sigma_i) \rrbracket_{\text{tr}}$. Using the
747 second and third properties of Δ_i , we can see that there is one and only one $(I, L') \in$
748 $\Delta_i(L_2, \mathbb{R}^+)(\alpha, t)$ such that $\tau_{\text{last}}(u') \in I$. That means we have $u' \in \llbracket [\Phi_i(L_1)(\Sigma_i)]^I \rrbracket_{\text{tr}}$
749 and $u_1 \cdot v \in \llbracket L' \rrbracket_{\text{tr}}$. We then proved in both cases that $u \cdot v \in \llbracket \text{Pr}_i(L)(\alpha, t) \rrbracket_{\text{tr}}$.

750 **■ If $L = L_1^*$.**

751 **1.** Assume $u \in \llbracket \text{Pr}_i(L)(\alpha, t) \rrbracket_{\text{tr}}$. This means that there is $(I, L') \in \Delta_i(L_1, \mathbb{R}^+)(\alpha, t)$ such
752 that $u = u_1 \cdot u_2 \cdot u_3$ with $u_1 \in \llbracket [\Phi(L_1)(\Sigma_i)^*]^I \rrbracket_{\text{tr}}$ and $u_3 \in \llbracket [L_1^*]_{[t, \infty]} \downarrow_{\tau_{\text{last}}(u_2)} \rrbracket_{\text{tr}}$ as well
753 as $u_2 \in \llbracket L' \rrbracket_{\text{tr}}$. Using the first property of Δ_i , this means $u_2 \in \llbracket \text{Pr}_i(L_1 \downarrow_{\tau_{\text{last}}(u_1)})(\alpha, t) \rrbracket_{\text{tr}}$.
754 In other words, we know that $u_1 \in \llbracket L_1^* \rrbracket_{\text{tr}}$, $u_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$ and $u_2 = v_1 \cdot v_2$ with
755 $v_1 \cdot (\alpha, t) \cdot v_2 \in \llbracket L_1 \downarrow_{\tau_{\text{last}}(u_1)} \rrbracket_{\text{tr}}$. We can then deduce that $u_1 \cdot v_1 \cdot (\alpha, t) \cdot v_2 \cdot v_3 \in \llbracket L_1^* \rrbracket_{\text{tr}}$
756 with $u_1 \cdot v_1 \in \mathcal{R}(\Sigma \setminus \Sigma_i)$.

757 **2.** Now to prove the other inclusion. Assume $u \cdot v$ such that $w = u \cdot (\alpha, t) \cdot v \in \llbracket L \rrbracket_{\text{tr}}$.
758 Since w is not empty, $w = w_1 \cdot w_2 \cdot \dots \cdot w_n$, with $w_i \in \llbracket L_1 \downarrow_{\tau_{\text{last}}(w_{i-1})} \rrbracket_{\text{tr}}$. Because
759 we know that w contains (α, t) , we can denote by k the index of w_i that contains
760 this event. In other words, we have $w_k = u' \cdot (\alpha, t) \cdot v' \in \llbracket L_1 \downarrow_{\tau_{\text{last}}(w_{k-1})} \rrbracket_{\text{tr}}$, which
761 means that $u' \cdot v' \in \llbracket \text{Prog}_i(L_1 \downarrow_{\tau_{\text{last}}(w_{i-1})})(\alpha, t) \rrbracket_{\text{tr}}$. Using the second and third
762 properties of Δ_i we deduce that there is one and only one $(I, L') \in \Delta_i(L_1, \mathbb{R}^+)(\alpha, t)$
763 such that $\tau_{\text{last}}(w_{k-1}) \in I$ and in that case we have $w_k \in \llbracket L' \rrbracket_{\text{tr}}$. This shows that
764 $u \cdot v = w_0 \cdot \dots \cdot w_{k-1} \cdot u' \cdot v' \cdot w_{k+1} \cdot \dots \cdot w_n \in \llbracket [\Phi(L_1)(\Sigma_i)^*]^I \cdot L' \cdot [L_1^*]_{[t, \infty]} \rrbracket_{\text{tr}}$
765 ◀

