Organising LTL Monitors over Distributed Systems with a Global Clock

Christian Colombo, Yliès Falcone

Formal Methods in System Design (FMSD) manuscript No.
(will be inserted by the editor)

Organising LTL Monitors over Distributed Systems with a Global Clock

Christian Colombo · **Yliès Falcone**

Received: date / Accepted: date

Abstract Users wanting to monitor distributed systems often prefer to abstract away the architecture of the system by directly specifying correctness properties on the global system behaviour. To support this abstraction, a compilation of the properties would not only involve the typical choice of monitoring algorithm, but also the organisation of submonitors across the component network. Existing approaches, considered in the context of LTL properties over distributed systems with a global clock, include the so-called orchestration and migration approaches. In the orchestration approach, a central monitor receives the events from all subsystems. In the migration approach, LTL formulae transfer themselves across subsystems to gather local information.

We propose a third way of organising submonitors: choreography, where monitors are organised as a tree across the distributed system, and each child feeds intermediate results to its parent. We formalise choreography-based decentralised monitoring by showing how to synthesise a network from an LTL formula, and give a decentralised monitoring algorithm working on top of an LTL network. We prove the algorithm correct and implement it in a benchmark tool. We also report on an empirical investigation comparing these three approaches on several concerns of decentralised monitoring: the delay in reaching a verdict due to communication latency, the number and size of the messages exchanged, and the number of execution steps required to reach the verdict.

1 Introduction

Since processor speed has stabilised over the past years, more systems are being designed to be decentralised to benefit from the multiple cores typically present in contemporary processors. This shift in system design poses a number of challenges in the domain of runtime

C. Colombo
Department of Computer Science, University of Malta
E-mail: christian.colombo@um.edu.mt

Y. Falcone
Univ. Grenoble Alpes, Inria, LIG, F-38000 Grenoble, France
University of Illinois at Urbana-Champaign
E-mail: ylies.falcone@imag.fr

verification, namely, how to exploit the distributed resources to keep the processing, memory, and communication overheads as low as possible without imposing substantial changes in the monitored distributed architecture.

In runtime verification (cf. [12,17,20,32]) one is interested in synthesising a monitor to evaluate a stream of events (reflecting the behaviour of a system) according to some correctness properties. When the system consists of several computing units (referred to as components in the sequel), it is desirable to decentralise the monitoring process for several reasons, including reducing the number and size of required messages as well as the computation needed to reach a verdict (as seen in [3,6,10]). One way of exploiting the availability of multiple system components is by decentralising the monitoring process, i.e., designing decentralised monitors that are as independent as possible and share the computation required for monitoring. Moreover, decentralised monitoring avoids introducing a central observation point in the system (which typically requires a modification of the system architecture), and it also generally reduces the communication overhead.

In this paper, we study the problem of decentralised monitoring of distributed systems with a global clock. In this setting, the system is composed of several (black box) components $C_1, C_2, \ldots, C_n$ for some $n \in \mathbb{N} \setminus \{0\}$ ($\mathbb{N}$ being the set of natural numbers) running on the same clock where each component C_i has a local set of atomic propositions of interest AP_i. A local monitor can be attached to each component so that with instrumentation each component can produce a local trace of events, which at time $t \in \mathbb{N}$ is $u_i(0)\cdots u_i(t)$ where $u_i(t') \in 2^{AP_i}$ is the event emitted at time $t' \leq t$. Furthermore, the specification of the system is given by a Linear Temporal Logic (LTL, cf. [25]) formula φ defined over $\cup_{i\in[1,n]} AP_i$. Note that in general $2^{\cup_{i\in[1,n]} AP_i} \neq \cup_{i\in[1,n]} 2^{AP_i}$, implying in particular that the evaluation of the specification (i) cannot be performed on components in isolation, and (ii) imposes communication. The decentralised monitoring problem then consists in checking the (virtual) global trace of the system (that can be reconstructed from the local traces) $u_1(0) \cup \ldots \cup u_n(0) \cdot u_1(1) \cup \ldots \cup u_n(1) \cdots u_1(t) \cup \ldots \cup u_n(t)$ against φ at any time $t' \leq t$. Several communication protocols may be used by local monitors to communicate and reach a global verdict. For instance, communication can be considered as instantaneous and the monitors are allowed to send an arbitrary number of messages to each other between two system steps[1], as is the case in [7]. Communication can be considered as having a fixed latency depending on the global clock. For instance, [6] assumes that the communication between monitors occurs between ticks of the global clock and that any message sent at time t is received at time $t+1$. Another possibility is to design a decentralised monitoring algorithm and protocol so that no assumption is made wrt. the latency of messages, as in [10].

According to the computation performed by local monitors and their communication protocol, a decentralised monitoring solution can be categorised under the setting of either orchestration, migration, or choreography (using terminology from [13]), using LTL as a reference instantiation:

Orchestration: Orchestration is the setting where a single monitor (either one of the monitors attached to local components or an additional monitor introduced in the system) carries out all the monitoring processing while receiving local events from the other local monitors. Orchestration forces most of the computation payload to be placed on the local monitor receiving the events, while reducing the computation of other monitors to sending their local events.

[1] We abstract away from clock and communication cycles and take a "step" to signify each time a fresh set of events becomes available to the monitor.

Migration: Migration is the setting where the monitoring entity transports itself across the network, evolving as it goes along — abstracting away the need to transfer lower level (finer-grained) information.

Choreography: Choreography is the setting where monitors are organised into a network and a protocol is used to enable cooperation between monitors. The synthesis of the communication network uses the structure of the monitored formula to place specialised monitors in charge of monitoring only a "projection" of the formula. Hence, compared to migration, choreography leverages the syntactic structure of the monitored formula and uses static information gathered from an offline analysis of the formula.

Assumptions. At this point, we would like to emphasise the important assumptions in our work. First, we assume the existence of a *global clock* in the system of which the local monitors are aware (as in [6,10]). This assumption is realistic for several critical industrial systems (e.g., [15,24,28]), including those where the system at hand is composed of several applications executing on the same operating system, or where a synchronous bus (e.g., FlexRay) provides the global clock. Furthermore, we direct the reader to [3] for an application of our earlier work [6] on networked embedded systems. Second, we assume that the monitors can communicate directly with each other in a reliable fashion: any message sent is supposed to eventually arrive unaltered to its destination.[2] However, we do not make assumptions on the preservation of the order of messages sent by each monitor.

Contributions. In the sequel, first, we present at an abstract level the three settings for decentralised monitoring of distributed systems with a global clock (namely orchestration, migration, and choreography). Second, we introduce choreography-based decentralised monitoring with full formal details. We present some algorithms that are applied before monitoring and use the syntax of the LTL formula at hand to synthesise a monitoring network where local monitors can be placed to form a choreography at runtime. Third, we present the choreography-based monitoring algorithm. Fourth, we empirically compare orchestration, migration (from [6]), and choreography using a benchmark implementation.

Note that this paper extends [7] with the following additional contributions, mainly revolving around an overhaul of the design of the previous framework, a more efficient decentralised monitoring algorithm, complete proofs, and a more extensive evaluation:

- We dropped the assumption of instantaneous communication of monitors: our approach in [7] assumes that local monitors can send an arbitrary number of communication messages to each other between any two ticks of the global clock. This is shown in Fig. 1(left) where monitoring nodes M_i and M_{i+1} continue to progress (denoted by loops) and communicate (denoted by dotted arrows) after each system cycle (denoted by t, $t+1$, $t+2$) until no further progressions occur and consequently there is nothing to be communicated. The framework presented in this paper only uses the information provided by the global clock to keep track of event ordering and does not make any assumptions on the latency of messages. Our new framework is thus compatible with more communication schemes, ranging from instantaneous communication as in [7], to communication with a fixed or bounded delay (cf. [19]), to communication without any guarantee on the arrival time of messages. This is shown in Fig. 1(right) where monitor communication cycles can span multiple system cycles or vice versa. Thus, the algorithm in this paper is more flexible and practical because monitoring messages can in particular be piggy-backed on

[2] Many algorithms can be used for guaranteeing the absence of message loss in distributed systems, see [21] for instance.

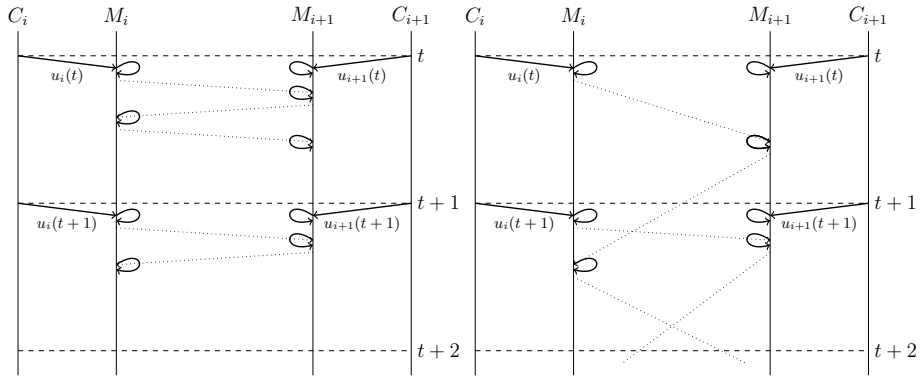


Fig. 1: The instantaneous communication timing model (left) vs. decoupled timing (right).

existing system messages using time-stamps provided by the global clock. Thus, if used in this setting, our algorithm does not impose the sending of extra messages in the system for monitoring purposes.

- Drawing inspiration from git terminology, the underlying communication network works by "pushing" instead of "pulling" information (as in [7]): monitors now know what messages they have to send (push) to which components, instead of having monitors requesting information from other monitors and then receiving it (pulling information).
- We provide a more efficient algorithm where the communication and the hierarchical organisation of monitors is fixed, thus avoiding the reconfigurations needed at every system step in [7]. Such reconfigurations are avoided by analysing the formula prior to monitoring, computing a communication network where "projections" of the monitored formula are locally monitored on each component. Hence, the messages exchanged between monitors are verdicts for the "projected" formulae.
- We study more properties of choreography-based decentralised monitoring by introducing Lemma 3 and Theorem 1 (in Section 4), proving the correctness of the verdict reached even when the delay of message arrival is unbounded.
- We provide more details and examples at several places in this paper, and include the proofs of all theorems.
- We provide a new implementation of choreography-based monitoring and include it in DecentMon[3], an existing benchmark for evaluating decentralised monitoring.
- We evaluate our new implementation against the metrics studied in [7] (such as the delay, size and number of messages exchanged by monitors) but provide larger experiments and also study and compare the performance of choreography-based decentralised monitoring when monitoring realistic specifications given by specification patterns [8]. Note that since the decentralised monitoring algorithm introduced in this paper is more general than the one (implemented) in [6] in terms of communication delay, we compare the implementation in the unified setting where the communication between monitors takes one unit of time.

Paper organisation. The rest of the paper is organised as follows. Section 2 introduces some background. Section 3 recalls the orchestration, migration, and choreography approaches

[3] `http://decentmonitor.forge.imag.fr`

for LTL monitoring. In Section 4, we introduce the mathematics of generating a network for choreography-based decentralised monitoring, while Section 5 outlines how the network evolves during monitoring. Then, Section 6 gives the guarantees of our decentralised monitoring algorithm. Section 7 reports on our empirical evaluation and comparison of the three approaches using a benchmark implementation. Section 8 compares this paper with related work. Finally, Section 9 concludes and proposes some avenues for future work. Appendix A contains the proofs of the propositions and theorems. Appendix B contains plots for the complementary visualisation of the experiment results presented in Section 7.

2 Background

In this section, we formally define the mathematical notation which will be used throughout the paper focusing on the notions of a distributed system and alphabet, followed by an introduction to the syntax and semantics of LTL.

Notation on functions. For a function f, we note $f : X \rightharpoonup Y$ (resp. $f : X \to Y$) when f is a partial (resp. total) function from set X to set Y; $X \rightharpoonup Y$ (resp. $X \to Y$) denotes the set of partial (resp. total) functions from X to Y. The domain (resp. co-domain) of f is the subset of X (resp. Y) for which there is a corresponding Y (resp. X) element: $\mathrm{dom}(f) \stackrel{\text{def}}{=} \{x \in X \mid \exists y \in Y \cdot f(x) = y\}$, $\mathrm{codom}(f) \stackrel{\text{def}}{=} \{y \in Y \mid \exists x \in X \cdot f(x) = y\}$. Moreover, we denote by $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ a partial function defined on some elements $x_1, \ldots, x_n$ with values $y_1, \ldots, y_n$, respectively. Finally, we define the function-override operator $\dagger$: $f \dagger g$ denotes the function where the values of function f are overridden by the values of function g. Function $f \dagger g$ is defined as follows:

$$(f \dagger g)(x) \stackrel{\text{def}}{=} \begin{cases} g(x) & \text{if } x \in \mathrm{dom}(g), \\ f(x) & \text{if } x \notin \mathrm{dom}(g) \text{ and } x \in \mathrm{dom}(f), \\ \mathit{undef} & \text{otherwise.} \end{cases}$$

Distributed systems and alphabet. Let a distributed system be represented by a set of components: $\mathscr{C} = \{C_0, C_1, \ldots, C_n\}$ for some $n \in \mathbb{N}$, and the alphabet Σ be the set of all events of the components: $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_n$, where Σ_i is the alphabet of C_i built over a set of local atomic propositions AP_i: $\Sigma_i = 2^{AP_i}$. Note that in general $\Sigma \neq 2^{\cup_i AP_i}$. For the sake of a simpler presentation, we assume that the alphabets and sets of local atomic propositions are pair-wise disjoint.[4] We define the function # returning the index of the component related to an atomic proposition, if it exists: $\# : AP_i \to \mathbb{N}$ such that $\#a \stackrel{\text{def}}{=} i$ if $\exists i \in [1;n] \cdot a \in AP_i$ and undefined otherwise. The behaviour u_i of each component C_i is represented by a *trace* of events, which for t time steps is encoded as $u_i = u_i(0) \cdot u_i(1) \cdots u_i(t-1)$ with $\forall t' < t \cdot\ u_i(t') \in \Sigma_i$. Finite traces over Σ are elements of Σ^* and are denoted by $u, u', \ldots$. Similarly, we use $w, w', \ldots$ to denote infinite traces, elements of Σ^ω, with $w(t)$ referring to the behaviour at time step t. The finite or infinite sequence w^t is the *suffix* of the trace $w \in \Sigma^\omega$, starting at time t, i.e., $w^t = w(t) \cdot w(t+1) \cdots$.

[4] This assumption simplifies the presentation but does not affect the generality of the results since any conflicting alphabet elements can be renamed and the LTL formula adapted accordingly, e.g., consider a proposition a observable on two components; renaming a to a_1 and a_2 on the respective components, we monitor for $a_1 \vee a_2$ instead of a. For simplicity we assume the alphabets are pair-wise disjoint.

Linear temporal logic. The system's global behaviour, $(u_1, u_2, \ldots, u_n)$ can now be described as a sequence of pair-wise unions of the local events in component's traces, each of which at time t is of length $t+1$ i.e., $u = u(0)\cdots u(t)$.

We monitor a system *wrt.* a global specification, expressed as a standard LTL [25] formula, i.e., the formula does not state anything about the system's architecture or how it should be distributed. LTL formulae can be described using the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi\,\mathbf{U}\,\varphi, \text{where } p \in AP \text{ and } AP = \bigcup_{i=1}^{n} AP_i.$$

Moreover, we extend the syntax of LTL with the following operators, defined in terms of the ones above: $\top \stackrel{\text{def}}{=} p \vee \neg p$, $\bot \stackrel{\text{def}}{=} \neg\top$, $\varphi_1 \wedge \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\mathbf{F}\varphi \stackrel{\text{def}}{=} \top\,\mathbf{U}\,\varphi$, and $\mathbf{G}\varphi \stackrel{\text{def}}{=} \neg\,\mathbf{F}\,(\neg\varphi)$.

Definition 1 (LTL semantics [25]) Let $w \in \Sigma^\omega$ and $i \in \mathbb{N}$. Satisfaction of an LTL formula by w at time i is defined inductively:

$$\begin{array}{ll}
w^i \models p & \Leftrightarrow p \in w(i), \text{ for any } p \in AP \\
w^i \models \neg\varphi & \Leftrightarrow w^i \not\models \varphi \\
w^i \models \varphi_1 \vee \varphi_2 & \Leftrightarrow w^i \models \varphi_1 \textit{ or } w^i \models \varphi_2 \\
w^i \models \mathbf{X}\varphi & \Leftrightarrow w^{i+1} \models \varphi \\
w^i \models \varphi_1\,\mathbf{U}\,\varphi_2 & \Leftrightarrow \exists k \in [i,\infty[\cdot\ w^k \models \varphi_2 \textit{ and } \forall l \in [i,k[\cdot w^l \models \varphi_1
\end{array}$$

For readability, since $w^0 = w$, when $w^0 \models \varphi$ holds we simply write $w \models \varphi$.

Remark 1 (Using pattern matching notation in mathematical definitions) In this paper, certain functions take LTL formulae as input and are defined inductively over the syntax of the input formula. For the sake of conciseness, we shall use pattern matching with OCaml-based syntax (`https://ocaml.org`). For instance, for some function f taking some formula $\varphi \in$ LTL as input, we shall write

$$\begin{array}{l}
f(\varphi) = \text{match } \varphi \text{ with} \\
\mid p \in AP \to e_1(p) \quad \mid \psi \vee \psi'' \to e_4(\psi, \psi''), \\
\mid \neg\psi \quad\ \to e_2(\psi), \mid \psi\,\mathbf{U}\,\psi' \to e_5(\psi, \psi''), \\
\mid \mathbf{X}\psi \quad\ \to e_3(\psi), \mid _ \quad\ \to e_6.
\end{array}$$

where the $e_i(\ldots)$ are expressions depending on their input parameters (if any), to denote

$$f(\varphi) = \begin{cases}
e_1(p) & \text{if } \exists p \in AP \cdot \varphi = p, \\
e_2(\psi) & \text{if } \exists \psi \in \text{LTL} \cdot \varphi = \neg\psi, \\
e_3(\psi) & \text{if } \exists \psi \in \text{LTL} \cdot \varphi = \mathbf{X}\psi, \\
e_4(\psi, \psi') & \text{if } \exists \psi, \psi' \in \text{LTL} \cdot \psi \vee \psi', \\
e_5(\psi, \psi') & \text{if } \exists \psi, \psi' \in \text{LTL} \cdot \varphi = \psi\,\mathbf{U}\,\psi', \\
e_6 & \text{otherwise.}
\end{cases}$$

Several approaches have been proposed for adapting LTL semantics for monitoring purposes (cf. [4]). Here, we follow previous work [6] and consider LTL_3 (introduced in [5]).

Definition 2 (LTL_3 verdicts [5]) Given $u \in \Sigma^*$, the satisfaction relation of LTL_3, $\models_3 : \Sigma^* \times \text{LTL} \to \mathbb{B}_3$, with $\mathbb{B}_3 \stackrel{\text{def}}{=} \{\top, \bot, ?\}$, is defined as:

$$u \models_3 \varphi = \begin{cases}
\top & \text{if } \forall w \in \Sigma^\omega \cdot u \cdot w \models \varphi, \\
\bot & \text{if } \forall w \in \Sigma^\omega \cdot u \cdot w \not\models \varphi, \\
? & \text{otherwise.}
\end{cases}$$

Two LTL formulae φ and ψ are semantically equivalent according to LTL_3 verdicts, or semantically equivalent for short, noted $\varphi \simeq \psi$, whenever $u \models_3 \varphi = u \models_3 \psi$, for any $u \in \Sigma^*$.

To monitor LTL formulae at runtime, we make use of formula rewriting (cf. [18] for instance), aka progression or derivative.

Definition 3 (Progression for LTL) When applied to some event σ, the progression function $\text{prog} : \text{LTL} \times \Sigma \to \text{LTL}$ is defined as follows:

$$
\begin{array}{l}
\text{prog}(\varphi,\sigma) = \text{match } \varphi \text{ with} \\
\quad \begin{array}{llll}
\mid \top/\bot & \to \top/\bot & \mid \psi \vee \psi'' & \to \text{prog}(\psi,\sigma) \vee \text{prog}(\psi'',\sigma), \\
\mid p \in AP & \to \begin{cases} \top \text{ if } p \in \sigma, \\ \bot \text{ otherwise,} \end{cases} & \mid \psi \,\mathbf{U}\, \psi' & \to \text{prog}(\psi',\sigma) \vee (\text{prog}(\psi,\sigma) \wedge \psi \,\mathbf{U}\, \psi'), \\
\mid \neg\psi & \to \neg\text{prog}(\psi,\sigma), & \mid \mathbf{G}\psi & \to \text{prog}(\psi,\sigma) \wedge \mathbf{G}\psi, \\
\mid \mathbf{X}\psi & \to \psi, & \mid \mathbf{F}\psi & \to \text{prog}(\psi,\sigma) \vee \mathbf{F}\psi, \\
 & & \mid \_ & \to \varphi.
\end{array}
\end{array}
$$

The definition of progression is inspired by the fix-point semantics of LTL. For example, it intuitively reads as follows: for $\psi \,\mathbf{U}\, \psi'$ to hold at time t given the observed event σ, either ψ' has to hold at time t with σ, or ψ has to hold at time t given σ and $\psi \,\mathbf{U}\, \psi'$ has to hold at the next time instant $(t+1)$. Note that, to be more effective and usable during monitoring, the progression function is also defined for syntactic-sugar LTL formulae such as $\top, \bot, \mathbf{G}\psi$, and $\mathbf{F}\psi$.

If a central observation point exists in the system, progression can serve as a sound monitoring algorithm by simply successively applying the progression function on each event of a trace in order [6,18]. More precisely, function prog is lifted to traces by setting $\text{prog}(\varphi,\varepsilon) = \varphi$, and $\text{prog}(\varphi, u \cdot \sigma) = \text{prog}\left(\text{prog}\left(\varphi,u\right),\sigma\right)$. The LTL formulae returned by function prog can be interpreted as verdicts (in $\mathbb{B}_3$) by associating verdict $\top$ (resp. $\bot$, ?) with formula $\top$ (resp. formula $\bot$, any other formula). Note that we compare the verdicts built from the formulae returned by function prog with the verdicts produced by our decentralised monitoring algorithm to prove their correctness (see the first two lemmata in Section 6).

Before extending the above framework to decentralised monitoring, we convey an important remark related to the practicality of monitoring LTL by rewriting.

Remark 2 (On the simplification of LTL formulae) A known shortcoming of monitoring LTL by progression (or more generally by rewriting) is the growth of the formula representing the evolving monitor state [2,18,29]. This issue is usually addressed by trying to simplify the formula at hand, that is, finding a smaller formula that remains semantically equivalent to the formula obtained after progression. We note that solving such simplification issue could be theoretically achieved by either (i) using translations of formulae into equivalent automata and using automata minimisation (cf. [1,23]) or (ii) by generating optimal equivalent monitors by coinduction [30]. However, the prohibitive complexity of such procedures (at least PSPACE-hard) renders these techniques not applicable at runtime and thus not suitable for our monitoring purposes. Note also that existing axiomatisations of LTL could not be used either for our purposes since what we need is a finite confluent rewriting system. To the best of the authors' knowledge, the existence or non-existence of such rewriting system for LTL formula simplification is still an open question. In our implementation, we leverage existing equivalences between LTL and propositional formulae (e.g., distributivity, idempotency, absorption, laws [1,22]) to implement a simplification procedure that is linear in the depth of the input formula. We note that syntactic simplification rules are also used as heuristics applied before synthesising Büchi automata from LTL formulae for model-checking purposes (cf. [9,33]). Finally, we note that the number of progressions that needs to be carried out

Table 1: Comparison of orchestration, migration, and choreography in terms of organisation of monitors, monitored formula, and communication between monitors.

	Orchestration	**Migration**	**Choreography**
Organisation	One monitor at a central location	One or more monitor(s)	One monitor at each component
Monitored formula	Global	Global	Local
Communication	All components report their observation to the central monitor	Monitors transport themselves across the network gathering information from the visited components	Monitors send verdicts to each other

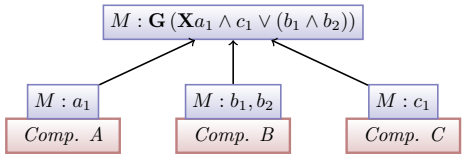


Fig. 2: An example of orchestration architecture.

at runtime is obviously influenced by the effectiveness of formula simplification: the more formula simplification is effective in reducing the size of monitored formulae, the less computation is required when progressing these. However, this effectiveness has no effect on the soundness of the decentralised monitoring algorithms based on progression.

Since simplification of LTL formulae is not the focus of this paper, we refer the reader to our implementation[5] for a description of the simplification procedure used in our approach to monitor LTL formulae.

3 Organising Monitors in a Decentralised Setting

A distributed system allows decentralised monitoring elements to be set in various configurations with different interaction schemes. The following subsections describe, at an abstract level, three main settings, namely orchestration, migration, and choreography, that can be used to monitor a formula on some components $C_1, \ldots, C_n$. Note that (in general) no component is aware of all the propositional values involved in the monitored formula.

To illustrate the differences between the three settings, we consider a system consisting of three components A, B, and C and the formula $\mathbf{G}\,(\mathbf{X}a_1 \wedge c_1 \vee (b_1 \wedge b_2))$ where proposition(s) a_1 can be observed on A, b_1, b_2 on B, and c_1 on C. Table 1 summarises the differences between the settings in terms of organisations of monitors, monitored formula, and communication between monitors.

3.1 Orchestration

The idea of orchestration-based monitoring [13] is to use a *central observation point* in the network (see Fig. 2) which can be introduced as an additional component or as a monitor

[5] Available at: `http://decentmon3.forge.imag.fr`.

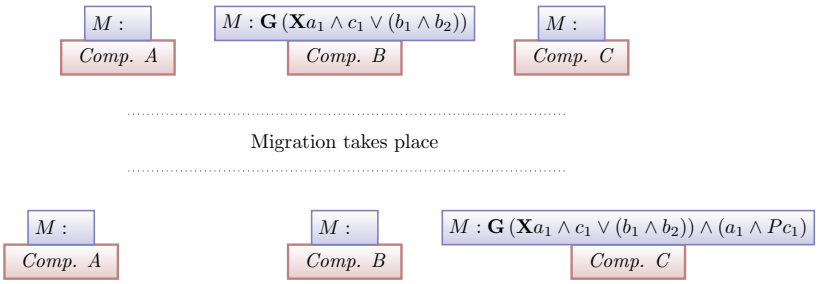


Fig. 3: An example of migration architecture.

attached to an existing component. A monitor for the global formula is attached to the central observation point and local monitors are in charge of producing events from their local observations. Several protocols can be used by local monitors to communicate events. For instance, local monitors can send their local event at every time instant. (Alternatively, the protocol may exploit the presence of a global clock in the system and just signal which propositions are true at any time instant or those whose value has changed.) Consequently, in orchestration-based monitoring, at any time t, the central observation point is aware of every event $u_i(t)$ occurring on each component C_i, and has thus the information about the global event $u_1(t) \cup \ldots \cup u_n(t)$ occurring in the system. From a theoretical perspective, putting aside instrumentation and communication, orchestration-based monitoring is not different from the usual (centralised) monitoring.

At each time instant, the steps involved in orchestration-based monitoring (at the central site) are the following:

1. Wait for all observations to arrive from the remote components.
2. Merge all observations to form an event.
3. Progress the LTL formula with the event and simplify the progressed formula.
4. If a verdict is reached, stop monitoring and report result.

For example, in Fig. 2, an additional component is added to the system as a central observation point to monitor the formula. At each time instant, the central observation point receives the local observations from other components A, B, and C, and merges them to form an event that is used by the monitor to progress the formula.

3.2 Migration

Migration-based monitoring was introduced in [6]. The idea of migration is to represent (the state of) a monitor as an LTL formula that travels across a network (see Fig. 3). There may be more than one formula travelling in the network. Each formula is an encoding of the global formula that has to be satisfied given the observations from the components traversed at previous time instants. Upon the reception of an LTL formula at some time t, a component C_i progresses it, i.e., C_i rewrites the formula given its local observation $u_i(t)$, so that the resulting formula is the formula that has to hold in the next computation step. Such formula may contain references to past time instants if it has been progressed by other components that could not evaluate some parts of it. Hence, component C_i may use its local trace with past observations $u_i(0) \cdots u_i(t-1)$ to resolve past references. More precisely, rewriting a

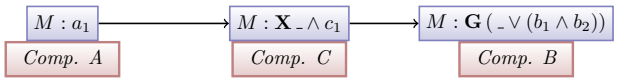


Fig. 4: An example of choreography architecture.

formula is done using the so-called *progression*, adapted to the decentralised case, i.e., taking into account the fact that a component only has information about the local propositions it has access to. Any verdict found by a component is an actual global verdict. However, because of past references, the verdict is typically reached with a delay depending on the size of the network.

At each time instant, the steps involved in migration-based monitoring (on a component with a formula to process) are the following:

1. Use the current local observations to resolve relevant propositions.
2. Use the local history to resolve any past references to local observations.
3. Progress the formula, adding an "obligation" to earlier observations when these are not locally available.
4. If a verdict is reached, stop monitoring and report result.
5. Otherwise, select the component which can resolve the pointer to the observation which goes most backwards in history and send the formula to this component.

For example, in Fig. 3 the formula is placed initially on component B and has been progressed with only the valuations of b_1 and b_2 (where, for this particular case, $b_1 \wedge b_2$ does not hold). For propositions a_1 and c_1 whose valuations are not available, an *obligation* is recorded which will have to be satisfied in a future time instant (by looking at the past). Pc_1 refers to the previous value of c_1. Note that no obligation is generated for a_1 since the proposition is placed under a next operator ($\mathbf{X}$). The rewritten formula is then sent to the most appropriate component (component C) — intuitively, the component that has the information about the proposition whose obligation reaches furthest into the past. Component C then progresses the received formula using its local observation but also using its local history of observations to evaluate the past propositions. After sending a formula, a component is left with nothing to evaluate, unless it receives a formula from another component.

3.3 Choreography

Rather than having the global formula at a single location (whether this is fixed as in orchestration or variable as in migration), choreography breaks down the formula across the network, forming a tree structure where results from subformulae flow up to the parent formula. Note that each component monitors subformulae that either contain references to local atomic propositions or *place holders*. Intuitively, place holders can be understood as three-state propositions that represent the verdict (true, false, or no verdict yet) of a remote subformula being evaluated on another component.

At each time instant, the steps involved in choreography-based monitoring (on each component) are the following:

1. If a verdict from a child is received:
   (a) Substitute the verdict for the corresponding place holder in the local formula;
   (b) Apply simplification rules to the local formula.
2. Progress the local formula using the local observation.

3. If the local formula reaches a verdict, send the verdict to the parent (if any).
4. If the formula at the root of the tree reaches a verdict, stop monitoring and report result.

Figure 4 shows how formula $\mathbf{G}(\mathbf{X}a_1 \wedge c_1 \vee (b_1 \wedge b_2))$ is spread across a network of three components. Component A only observes formula a_1 and sends its valuation to component C at each time instant. Component C evaluates formula $\mathbf{X}\_ \wedge c_1$ where _ is a place holder for the result of the evaluation of formula a_1 sent by component A. Whenever component C determines the evaluation of formula $\mathbf{X}a_1 \wedge c_1$, it sends the result to component B. Component B evaluates the main formula $\mathbf{G}(\_ \vee (b_1 \wedge b_2))$ where _ is a place holder for the result of the evaluation of formula $\mathbf{X}a_1 \wedge c_1$ sent by component C. Verdicts reported by component B are verdicts for the monitored formula.

4 Synthesising an LTL Network from an LTL Formula

Following the overview of choreography in the previous section, in this section, we explain how a network of LTL monitors can be setup to globally monitor a particular LTL formula over a decentralised system. Intuitively, a network of LTL monitors consists of a monitor per component where each monitor is a collection of what we will refer to as monitoring "cells". A monitoring cell contains an LTL formula possibly with "pointers" to other cells in the network. Monitoring cells communicate with each other until a verdict is reached.

4.1 LTL with Distribution Propositions and LTL Network

We start by extending the LTL syntax with special atomic propositions to support the distribution of LTL formulae in a network. We refer to these special atomic propositions as *distribution propositions*. Distribution propositions behave like normal atomic propositions except that they are not resolved through system events but rather through monitor communication.

Definition 4 (LTL with distribution propositions) LTL formulae with distribution propositions extend standard LTL syntax (as defined in Section 2 and represented below by ...) by adding two kinds of atomic propositions:

$$\varphi ::= \ldots \mid \langle\!\langle x,y \rangle\!\rangle \mid \langle\!\langle x,y,t \rangle\!\rangle, \text{ where } x,y,t \in \mathbb{N}.$$

We note by LTL_D the set of formulae in LTL augmented with distribution propositions.

Intuitively, proposition $\langle\!\langle x,y \rangle\!\rangle$ shall be used as a place holder pointing to cell y of component x in the network. This kind of proposition is used when statically generating a network before monitoring (in Definition 7 in Section 4.2). The other kind of proposition, $\langle\!\langle x,y,t \rangle\!\rangle$, is resolved and used at runtime when monitoring (see Section 5), with t referring to the value of the cell at a particular time instant.[6]

Remark 3 (About distribution propositions) Distribution propositions are only used internally in our definitions and functions for the purpose of decentralised monitoring. The end

[6] As opposed to [7], the introduced propositions do not need to contain a copy of the original formula since reconfiguration of the LTL network performed at runtime in [7] is now done by statically computing beforehand reconfiguration information through function compute_respawn (cf. Definition 8).

user, i.e., the one writing properties, does not need to be aware of them. Regarding simplification, distribution propositions are treated as atomic propositions, where two distribution propositions $\langle\!|x,y|\!\rangle$ and $\langle\!|x',y'|\!\rangle$ are considered to be equivalent when $x = x'$ or $y = y'$; and classical Boolean rules for simplification are adapted in a straightforward manner. Distribution propositions of the form $\langle\!|x,y,t|\!\rangle$ are simplified in the same way.

Given an LTL formula, we define a scoring function that returns a natural number representing the desirability of placing the monitor for that LTL formula on some particular component i — the one with the highest score is chosen.

Definition 5 (Choosing component)

- The scoring function $\mathrm{scor}_i : \mathrm{LTL} \to \mathbb{N}$ is defined as (using $\sim$ and $\odot$ to range over unary and binary LTL operators, resp.):

$$\begin{aligned}\mathrm{scor}_i(\varphi) \stackrel{\text{def}}{=} \text{match } \varphi \text{ with}\\ \quad | \sim\psi \to \mathrm{scor}_i(\psi) \qquad\qquad | \ \psi \odot \psi' \to \mathrm{scor}_i(\psi) + \mathrm{scor}_i(\psi')\\ \quad | \ p \quad \to \begin{cases} 1 \text{ if } \#p = i \\ 0 \text{ otherwise} \end{cases} \qquad | \ \_ \quad \to 0\end{aligned}$$

- The choice function $\mathrm{chc} : \mathrm{LTL} \to \mathbb{N}$ is defined as follows:

$$\mathrm{chc}(\varphi) \stackrel{\text{def}}{=} \arg\max_i(\mathrm{scor}_i(\varphi))$$

Note that chc might return a set of indices since components may have the same score. We leave it up to the implementer to choose any of such components, for example by choosing the one with the lowest index, or through some other strategy.

An important condition for choreography to function correctly is to ensure that for any proposition p, $\mathrm{chc}(p) = \#p$ holds since the value of p can only be resolved at component $\#p$. In what follows we assume this is always the case.

Remark 4 In Definition 5, the scoring function follows a greedy approach by trying to place a formula in the component where there is the highest number of propositions which can be resolved locally. Our evaluation and comparison of decentralised monitoring approaches (see Section 7) show that such a scoring function is effective in reducing the number of messages exchanged between choreography-based monitors. However, there are several ways of varying the scoring function. The following are two alternative examples: (i) Vary the weight of operands of binary operators, e.g., for a formula of the form $\psi \mathbf{U} \psi'$, ψ' can be given more weight than ψ; (ii) Giving more weight to a particular component, e.g., to create an orchestration where the whole formula except the remote propositions are on a single component.

Given the list of components of a system, a monitor network is a corresponding list of monitors (with one monitor per component) where each monitor has a number of monitoring cells where each cell checks an LTL_D formula.

Definition 6 (LTL network) An LTL network is a partial function $N : \mathbb{N} \rightharpoonup \mathbb{N} \rightharpoonup \mathrm{LTL}_D$ which, given a component identifier, returns a component's monitor, which in turn is a function returning the LTL_D formula contained in the monitoring cell given the reference number of that cell.

We use $\mathscr{N}$ to denote the set of networks and N, N', N'' to denote networks. As abbreviations we use N_i to refer to $N(i)$, i.e., the i-th component in network N, and $N_{i,j}$ to refer to $N_i(j)$, i.e., the j-th formula of the i-th component in N. Moreover, $|N_i| = |\mathrm{dom}(N_i)|$ refers to the size of the domain of N_i, i.e., the number of formulae in component i, while $N_{i,j} \mapsto \varphi$ is used as abbreviation for $N \dagger \{i \mapsto (N_i \dagger \{j \mapsto \varphi\})\}$ and $N_{i,*}$ as abbreviation for $N_{i,|N_i|}$.

4.2 Synthesising an LTL Network from an LTL Formula

Monitoring an LTL formula using a choreographed approach requires that the formula be split into parts and distributed on different components in the network such that the effort of progression is split correspondingly.

Intuitively, distributing a formula across a network requires two operations: modifying the formula to point to its subparts which reside in other components within the network, and inserting the formula with pointers inside the network. The function net defined below handles the latter aspect while the former is handled by distr. In turn distr (through recurs) recursively calls itself on subformulae until it encounters a subpart which should be placed on a different component (due to the scoring function). In this case, function net is called once more so that the remote subformula is inserted in the network accordingly. Using function chc, the subparts of a formula that "choose" a different component from their parent's can be marked as distributed using LTL_D distribution propositions and placed at a different point in the network.

The generation of an LTL network, i.e., the tree-like structure in which submonitors are organised, is defined as follows, using $N_E = \{i \mapsto \emptyset \mid 1 \leq i \leq n\}$ to denote the empty network.

Definition 7 (Generating an LTL network) For an LTL formula φ, the generated network is $\text{net}_{\text{chc}(\varphi)}(N_E, \varphi)$ where $\text{net}_i : \mathscr{N} \times \text{LTL} \to \mathscr{N}$ is defined using functions $\text{distr}_i : \mathscr{N} \times \text{LTL}_D \to \mathscr{N} \times \text{LTL}_D$ and $\text{recurs}_i : \mathscr{N} \times \text{LTL}_D \to \mathscr{N} \times \text{LTL}_D$ as follows:

$$\text{net}_i(N, \varphi) \stackrel{\text{def}}{=} \text{let } N', \varphi' = \text{distr}_i(N, \varphi) \text{ in } N'_{i,*} \mapsto \varphi'$$

$$\begin{aligned}
\text{where } \text{distr}_i(N, \varphi) \stackrel{\text{def}}{=} & \text{ match } \varphi \text{ with} \\
& \mid \sim \psi \quad \to \text{let } N', \psi' = \text{distr}_i(N, \psi) \text{ in } N', \sim \psi' \\
& \mid \psi \odot \psi' \to \text{let } N', \psi'' = \text{recurs}_i(N, \psi) \text{ in} \\
& \qquad\quad \text{let } N'', \psi''' = \text{recurs}_i(N', \psi') \text{ in } N'', \psi'' \odot \psi''' \\
& \mid _ \quad \to N, \varphi \\
\text{and } \text{recurs}_i(N, \varphi) \stackrel{\text{def}}{=} & \text{ let } j = \text{chc}(\varphi) \text{ in } \begin{cases} \text{distr}_i(N, \varphi) & \text{if } j = i, \\ \text{net}_j(N, \varphi), \langle\!\langle j, |N_j| \rangle\!\rangle & \text{otherwise.} \end{cases}
\end{aligned}$$

A network is generated by three mutually recursive functions: $\text{net}_i, \text{distr}_i$, and recurs_i, all parameterised by a component index i:

- Function net_i is applied to a network N and a formula φ when φ has to be distributed on N from component i. Function net_i calls function distr_i, which computes φ', the distributed version of formula φ over N, and returns a new network N' containing φ' and its referees. Formula φ' is then "planted" at the first available location in component i in network N'.
- Function distr_i, applied to a network N and a formula φ, computes the distribution of φ over N knowing that φ has to be placed in component i. Function distr_i is inductively defined over the structure of φ. Two main cases can be distinguished depending on whether φ is an atomic proposition or not. If φ is an atomic proposition, the input network and the formula are returned unchanged, since the formula refers only to one of the atomic propositions of component i. If φ is not an atomic proposition, it means that it is defined as a unary operation on a subformula, or a binary composition of subformulae. In the former case, distr_i is called on the subformula beneath the unary operator since $\text{chc}(\sim \psi) = \text{chc}(\psi)$. In the latter cases, while φ has to be monitored on this component (as indicated by function chc), it might be the case that one or both subformulae have to

be monitored from other components. Consequently, function recurs_i is called to check this latter fact.

- Function recurs_i, applied to a network N and a formula φ, first computes the most appropriate component for φ through function chc. If component i is indeed the most appropriate component for distributing φ (case $i = j$), then recurs_i calls distr_i with the same arguments. Otherwise, a pointer $\langle\!| j, |N_j| |\!\rangle$ is placed on component i and formula φ is distributed from component j. Pointer $\langle\!| j, |N_j| |\!\rangle$ refers to the first available location on component j where the distributed version of φ is placed.

In the following, $\text{net}(\varphi)$ stands for $\text{net}_{\text{chc}(\varphi)}(N_E, \varphi)$ and the last formula at component $\text{chc}(\varphi)$ in $\text{net}(\varphi)$ is referred to as the main formula, denoted $\lceil \text{net}(\varphi) \rceil$, i.e., where a global verdict may eventually be reached. Moreover, we use $\text{init}(i, j)$ to denote the formula to which cell j in component i is initialised, i.e., the value of $N_{i,j}$ after calling $net(\varphi)$.

Example 1 (Generating an LTL network) We consider two examples illustrating the generation of LTL networks.

– As a simple example, we consider the scenario of constructing a network for formula $\varphi = a \,\mathbf{U}\, b$ for a system with two components, A and B (numbered 1 and 2 resp.), with the former having proposition a at its disposal while the latter having proposition b.
Starting with a call to net, $\text{chc}(\varphi)$ may return either 1 or 2 depending on which element of the set returned by *argmax* is selected. In this case, we assume the former and call the distribution function on an empty network: $\text{distr}_1(N_E, \varphi)$. The corresponding network is generated as follows:

$$
\begin{aligned}
N, \varphi' &= \text{recurs}_1(N_E, a) = \text{distr}_1(N_E, a) = \{1 \mapsto \emptyset, 2 \mapsto \emptyset\}, a \\
O, \psi' &= \text{recurs}_1(N, b) = \text{net}(N, b), \langle\!| 2,0 |\!\rangle = \{1 \mapsto \{0 \mapsto b\}, 2 \mapsto \emptyset\}, \langle\!| 2,0 |\!\rangle \\
\text{distr}_1(N_E, \varphi) &= \{1 \mapsto \emptyset, 2 \mapsto \{0 \mapsto b\}\}, a \,\mathbf{U}\, \langle\!| 2,0 |\!\rangle \\
\text{net}(N_E, \varphi) &= \{1 \mapsto \{0 \mapsto a \,\mathbf{U}\, \langle\!| 2,0 |\!\rangle\}, 2 \mapsto \{0 \mapsto b\}\}
\end{aligned}
$$

– As a more complex example, which will be used as a running example, consider the formula: $c \wedge (a \,\mathbf{U}\, (a \wedge (b \wedge c)))$. The generated network is:

$$
\begin{aligned}
&\text{Comp. 0 } (a\text{'s}) : \{0 \mapsto \langle\!| 2,0 |\!\rangle \wedge (a \,\mathbf{U}\, (a \wedge \langle\!| 1,0 |\!\rangle))\}, \\
&\text{Comp. 1 } (b\text{'s}) : \{0 \mapsto b \wedge \langle\!| 2,1 |\!\rangle\}, \\
&\text{Comp. 2 } (c\text{'s}) : \{0 \mapsto c, 1 \mapsto c\}.
\end{aligned}
$$

Note that the main formula is placed in component 0 since it has two a's and two c's but precedence is given (arbitrarily) to components with a lower index. Note also that the third component is monitoring the same formula twice.

4.3 Compacting a Network

Networks as generated by Definition 7 may contain duplicate formulae on a component (as in the previous example). To remedy this situation, we apply a compacting algorithm to the network before monitoring. The following steps are applied until "stabilisation" occurs, i.e.,

until none of the components is modified in the network:

foreach $i \in \mathrm{codom}(N)$ **do** * scanning the network component by component *
  $duplicate\_forms = \{N_{i,j} \mid j \leq |N_i| \wedge \exists j' \leq |N_i| : j \neq j' \wedge N_{i,j} = N_{i,j'}\}$;
  foreach $(X, \varphi) \in \{(X, \varphi) \mid \varphi \in duplicate\_forms \wedge X = \{x \leq |N_i| \mid N_i^x = \varphi\}\}$;
  do
    foreach $k \in [1; |N|]$ $(k \neq i)$ *and* $j \in [1; |N_k|]$ **do**
      $N_{k,j} \mapsto N_{k,j}\,[\langle k, x\rangle$ *and* $x \neq \min X \mapsto \langle k, \min X\rangle]$
    end
    $N_i \mapsto \{x \mapsto \texttt{undef} \mid x \in X \setminus \{\min\ X\}\}$
  end
end

Algorithm 1: Compacting the network

Algorithm 1 scans the entire network, component by component. For each component, it computes the set of duplicate formulae and their indices on the component. For each set of equivalent formulae, it identifies the one with the smallest index and then replaces all references to the formulae in the network by a reference to the one with the smallest index. All cells, except the one with the smallest index, are emptied.

Remark 5 The computation performed in the second line of Algorithm 1 requires the determination of whether two formulae are semantically equivalent. In our implementation, however, we approximate this semantic equivalence check with a syntactic equality check after formula simplification.

Algorithm 1 may leave "holes" within the network. That is, the network may have at least one component where the set of indices of formulae does not form an interval of $\mathbb{N}$ starting from 0. A reindexing of formulae and pointers can be used to index formulae following natural numbers.

Example 2 Taking the case of the third component, $duplicate\_forms = \{c\}$ and consequently the following loop iterates twice: once on $(0, c)$ and a second time on $(1, c)$. Next, we loop through all formulae of the other components updating the pointers (from $\langle 2, 1\rangle$ to $\langle 2, 0\rangle$) and finally we remove the unused formulae with the resulting network below:

Comp. 0 (a's): $\{0 \mapsto \langle 2, 0\rangle \wedge (a\,\mathbf{U}\,(a \wedge \langle 1, 0\rangle))\}$,
Comp. 1 (b's): $\{0 \mapsto b \wedge \langle 2, 0\rangle\}$,
Comp. 2 (c's): $\{0 \mapsto c\}$.

4.4 Recording Communication Paths

After building, compacting, and reindexing, we scan the network and, whenever a cell $N_{i,j}$ has a pointer $\langle x, y\rangle$ to cell $N_{x,y}$,

- we record in cell $N_{i,j}$, the fact that this cell has a pointer to cell $N_{x,y}$;
- we record in cell $N_{x,y}$, the fact that component i has a pointer to this cell.

Keeping track of pointers among cells facilitates communication in the decentralised monitoring algorithm: when on component x, cell y resolves to a truth-value ($\top$ or $\bot$), component x knows that it should communicate this information to component i. Moreover, if at some point during monitoring, component i does not need the evaluation of cell $N_{x,y}$ any more, component i shall send a message to component x to notify the latter that the evaluation of its y-th cell is no longer needed. The above information is modelled by two partial functions, referents : $\mathbb{N} \times \mathbb{N} \rightharpoonup 2^{\mathbb{N}\times\mathbb{N}}$ and referrers : $\mathbb{N} \times \mathbb{N} \rightharpoonup 2^{\mathbb{N}}$. Intuitively, in a network N,

$(x,y) \in \text{referents}(i,j)$ means that cell $N_{i,j}$ (cell j in component i) has a pointer $\langle\!|x,y|\!\rangle$ to $N_{x,y}$ as a subformula. Conversely, $x \in \text{referrers}(i,j)$ means that component x has a cell with a pointer $\langle\!|i,j|\!\rangle$ referring to $N_{i,j}$ as a subformula.

Example 3 In the case of the running example, we have: referents $= \{(0,0) \mapsto \{(1,0),(2,0)\}, (1,0) \mapsto \{(2,0)\}\}$, and referrers $= \{(1,0) \mapsto \{0\}, (2,0) \mapsto \{0,1\}\}$.

4.5 Automatic Respawning

Each cell in the network may automatically respawn after progression. Intuitively, a cell of the network automatically respawns when another cell in the network has a pointer to this formula, and this cell is either a cell that automatically respawns or the pointer is in the scope of a $\mathbf{X}, \mathbf{U}, \mathbf{G}$, or $\mathbf{F}$ operator. Indeed, when a pointer is in the scope of such operators, after progression, the pointer may appear again in the progressed formula. The information regarding the automatic respawning of a cell shall be used in the decentralised monitoring algorithm presented in Section 5.4.

The computation of the respawning cells is defined by function compute_respawn : $\mathscr{N} \to 2^{\mathbb{N}\times\mathbb{N}}$, where compute_respawn$(N)$ is the set of indices of cells of network N that should automatically respawn.

Definition 8 (Automatically respawning cells) Function compute_respawn : $\mathscr{N} \to 2^{\mathbb{N}\times\mathbb{N}}$ is defined as follows:

$$\text{compute\_respawn}(N) = \text{cr}(\lceil N \rceil, \text{false}),$$

where $\text{cr} : \text{LTL}_D \times \{\text{true}, \text{false}\} \to 2^{\mathbb{N}\times\mathbb{N}}$ is defined as:

$$\begin{aligned}
\text{cr}(\varphi, b) = \text{match } \varphi, b \text{ with}\\
&\mid \neg\psi, b \to \text{cr}(\psi, b)\\
&\mid \psi \vee \psi', b \to \text{cr}(\psi, b) \cup \text{cr}(\psi', b)\\
&\mid \mathbf{X}\psi, b \to \text{cr}(\psi, \text{true})\\
&\mid \psi\,\mathbf{U}\,\psi', b \to \text{cr}(\psi, \text{true}) \cup \text{cr}(\psi', \text{true})\\
&\mid \langle\!|i,j|\!\rangle, \text{false} \to \text{cr}(N_{i,j}, \text{false})\\
&\mid \langle\!|i,j|\!\rangle, \text{true} \to \{(i,j)\} \cup \text{cr}(N_{i,j}, \text{true})\\
&\mid _, _ \to \emptyset
\end{aligned}$$

Function compute_respawn applied to a network N calls function cr on $\lceil N \rceil$, the main formula of N, scanning the network and recording the cells that need to automatically respawn. The second argument b is a marker recording that (when set to true) the cells traversed from that point onwards need to respawn. When called on a formula φ (a subformula of a cell) with a marker value b, a case analysis is done.

- When φ is of the form $\neg\psi$ for some formula ψ, function cr is simply called on subformula ψ with the same marker value.
- When φ is of the form $\mathbf{X}\psi$ (resp. $\psi\,\mathbf{U}\,\psi'$) for some formula(e) ψ (resp. ψ and ψ'), the result is the call (resp. the union of the calls) to cr on the subformula(e), but this time turning the marker to true so that from this point on, any encountered subformula during the scan will be in the scope of the next operator.
- When φ is of the form $\psi \vee \psi'$ for some formulae ψ and ψ', the respawning cells are those resulting from the union of the calls to function cr on ψ and ψ' with same marker.
- When φ is a pointer to some cell $N_{i,j}$, if the marker is false then function cr is called on the formula at $N_{i,j}$ with the same marker. Otherwise, cell j on component i is recorded as automatically respawning and function cr is called on the formula that is referred to.

Example 4 (Computing respawning cell) Applying function compute_respawn on the running network example, we obtain:

$$
\begin{aligned}
&\mathrm{cr}(\langle\!|2,0|\!\rangle \wedge (a\,\mathbf{U}\,(a \wedge \langle\!|1,0|\!\rangle)), \mathrm{false})\\
&\quad \text{(pointer } \langle\!|2,0|\!\rangle \text{ is ignored as it does not occur under a respawning operator)}\\
&\quad = \emptyset \cup \mathrm{cr}(a\,\mathbf{U}\,(a \wedge \langle\!|1,0|\!\rangle), \mathrm{false})\\
&\quad \text{(applying the case for } \mathbf{U})\\
&\quad = \emptyset \cup (\emptyset \cup \mathrm{cr}(a \wedge \langle\!|1,0|\!\rangle))\\
&\quad \text{(applying the case for conjunction and then } \langle\!|i,j|\!\rangle, \mathrm{true})\\
&\quad = \emptyset \cup (\emptyset \cup (\emptyset \cup (\{\langle\!|1,0|\!\rangle\} \cup \mathrm{cr}(b \wedge \langle\!|2,0|\!\rangle, \mathrm{true}))))\\
&\quad \text{(applying the case for conjunction and simplifying)}\\
&\quad = \{\langle\!|1,0|\!\rangle, \langle\!|2,0|\!\rangle\}.
\end{aligned}
$$

This concludes the process required to set up the monitoring network. In the next section, we specify how such a network evolves while processing the events from a distributed system.

5 Evolution of the LTL Network at Runtime

The definitions in the previous section are used to populate the data structures necessary for the choreographed monitoring of an LTL formula. In this section, we describe how using these data structures, LTL formulae are progressed independently and subsequently information flows across components, leading to a global verdict for the initial LTL formula.

5.1 The Memory of a Cell

There may be several evaluations of the same LTL (sub)formula on the same component due to the possibility of respawning; each at different stages of evaluation. Consequently, it is important to keep track of the different versions of the formula over time. Moreover, due to communication latency, a verdict might take time to reach the parent from the child. For these reasons, we choose the current progression iteration number to act as a kind of timestamp and version number at the same time.

This mechanism is managed by the "memory" of a cell: The memory of cell j on component i is modelled by a partial function $\mathrm{M}_{i,j} : \mathbb{N} \rightharpoonup \mathrm{LTL}_D$ that associates some of the time instants with an LTL_D formula. Prior to monitoring, $\mathrm{M}_{i,j}$ is initialised in such a way that time 0 is associated with the formula generated through Definition 7. We use $\mathrm{M}^t_{i,j}$ as an abbreviation for $\mathrm{M}_{i,j}(t)$, borrow abbreviations already introduced for networks and use $\lceil M \rceil$ to refer to the main formula at the latest time instant. For this reason, we use the LTL_D distribution proposition $\langle\!|i,j,t|\!\rangle$ (introduced in Definition 4), enabling the pointer to refer to a value in $\mathrm{M}_{i,j}$ at some specific time instant. Finally, we use $\mathscr{M}$ to represent the set of possible networks of memory cells: $\mathscr{M} \stackrel{\text{def}}{=} \mathbb{N} \rightharpoonup \mathbb{N} \rightharpoonup \mathbb{N} \rightharpoonup \mathrm{LTL}_D$ and use M, M' to range over $\mathscr{M}$.

Remark 6 A formula of the form $\langle\!|i,j,t|\!\rangle$ cannot appear as a subformula in a generated network (before monitoring).

5.2 Communication in a Push Architecture

There are two main reasons for communication in the proposed LTL network: either a child reaches a verdict and sends it to its parents (note that a child might have more than one par-

ent); or a parent signals its children that their verdict is no longer needed (a child terminates when the last parent sends such a signal).

For the first kind of messages, a component x sends (hence the term "push") a message $\text{verdict}(x,y,v,t)$ to a component c so as to notify component c that on component x, the y-th formula evaluated to v at time t. As for the second kind of messages, a component x sends a message $\text{kill}(x,y)$ to another component signifying that it is no longer interested in the verdict of formula $\mathrm{M}_{x,y}$ (for any future time point).

5.3 Progression for LTL_D

The progression of an LTL network consists of the progressions of individual formulae on the respective components. As such, the progression function given below is identical to the standard one except that it includes the extra cases of LTL_D.

Definition 9 (Progression for LTL_D) When applied at some time $t \in \mathbb{N}$, the progression function $\text{prog}_t : \text{LTL}_D \times \Sigma \to \text{LTL}_D$ is defined as follows[7]:

$$
\begin{array}{l}
\text{prog}_t(\varphi,\sigma) \stackrel{\text{def}}{=} \text{match } \varphi \text{ with} \\
\quad \mid \langle\!\langle i,j \rangle\!\rangle \quad \to \langle\!\langle i,j,t \rangle\!\rangle, \\
\quad \mid \langle\!\langle i,j,t' \rangle\!\rangle \to \langle\!\langle i,j,t' \rangle\!\rangle, \\
\quad \mid p \in AP \to \begin{cases} \top \text{ if } p \in \sigma, \\ \bot \text{ otherwise,} \end{cases} \\
\quad \mid \neg\psi \quad \to \neg\,\text{prog}_t(\psi,\sigma),
\end{array}
\qquad
\begin{array}{ll}
\mid \mathbf{X}\psi & \to \psi, \\
\mid \psi \vee \psi' & \to \text{prog}_t(\psi,\sigma) \vee \text{prog}_t(\psi',\sigma), \\
\mid \psi\,\mathbf{U}\,\psi' & \to \text{prog}_t(\psi',\sigma) \vee (\text{prog}_t(\psi,\sigma) \wedge \psi\,\mathbf{U}\,\psi'), \\
\mid \mathbf{G}\psi & \to \text{prog}_t(\psi,\sigma) \wedge \mathbf{G}\psi, \\
\mid \mathbf{F}\psi & \to \text{prog}_t(\psi,\sigma) \vee \mathbf{F}\psi, \\
\mid \_ & \to \varphi.
\end{array}
$$

Progression for LTL_D extends progression for LTL by adding cases for distribution propositions $\langle\!\langle i,j \rangle\!\rangle$ and $\langle\!\langle i,j,t \rangle\!\rangle$, with $i,j,t \in \mathbb{N}$. Progressing $\langle\!\langle i,j \rangle\!\rangle$ at time t yields $\langle\!\langle i,j,t \rangle\!\rangle$, i.e., it adds a time stamp indicating when the formula was first progressed (a corresponding formula in the network would have been spawned with the same number). Progressing $\langle\!\langle i,j,t \rangle\!\rangle$ leaves the formula unchanged. Progression for the other cases are defined as for standard LTL.

5.4 Decentralised Monitoring Algorithm

In this subsection, we present the algorithm running on each component, managing both the progression of the formulae and the communication to and from other components.

Definition 10 (Decentralised choreography-based monitoring algorithm) The algorithm continually repeats the following steps in order until termination. Let us assume that the algorithm is executed on component i with t being the current time instant on the global system clock. Furthermore, in the algorithm we abstract away from the splitting of the event across components, and use σ to represent $u(t)$. (Recall that a monitor only refers to local propositions by design ($\text{chc}(p) = \#p$).)

1 Wait for an external stimulus: either a communication message or an event.
2 In case a message $\text{verdict}(x,y,v,t')$ is received, replace every occurrence of $\langle\!\langle x,y,t' \rangle\!\rangle$ by v in every cell. Re-evaluate the formulae and apply simplification rules.

[7] We note that unlike in [7], the progression function is no longer responsible for reconfiguring the network (now this is achieved through the function compute_respawn), and thus the progression function is identical to the standard one except for the handling of the distribution propositions.

3 In case a kill(x,y) message is received:
   3.1 Suppress x from referrers(i,y).
   3.2 If referrers$(i,y)=\emptyset$, suppress cell (i,y) and send a kill(u,v) message to each $(u,v)\in$ referents(i,y), and set referents(i,y) to $\emptyset$.
4 If a verdict message is received, report it, and send verdict and termination signals to the referent components (i.e., those with indices in $(\{x\}\times\mathbb{N})\cap$referents$(i,j)$), and terminate.
5 In case an event σ is received.
   5.1 For each cell j that automatically respawns and that has a referrer, i.e., each cell in $\{j \mid (i,j)\in \text{compute\_respawn}(N)\wedge \text{referrers}(i,j)\neq\emptyset\}$, create a new entry in the cell memory with the initial formula of this cell, i.e., $\mathrm{M}^{t}_{i,j}\mapsto \text{init}(i,j)$.
   5.2 Apply the progression function to every cell in memory: for each time instant t', $\mathrm{M}^{t'}_{i,j}=\text{prog}_{t'}(\mathrm{M}^{t'}_{i,j},\sigma)$.
   5.3 Apply simplification rules (see Remark 3).
6 If the component has the main cell at coordinates (i,y'), and the formula in this cell evaluates to $\bot$ or $\top$, report the verdict, send termination and verdict signals to the referent components (i.e., the components in $\{x \mid \exists y\cdot(x,y)\in\text{referents}(i,y')\}$), and terminate.
7 For each cell j, compute the set referents$'(i,j)$ of cells that are currently referred to in j.
   7.1 For each cell (x,y) that is not referred to any more, i.e., for each $(x,y)\in$referents$'(i,j)\setminus$ referents(i,j): send kill(x,y) to component x.
   7.2 Set referents(i,j) to referents$'(i,j)$.
8 For each cell j, for each entry t' in $\mathrm{M}_{i,j}$ that resolves to a truth-value v, if the cell is referred to by another component x:
   8.1 Send a message verdict(i,j,v,t') to x.
   8.2 Suppress entry t' from $\mathrm{M}_{i,j}$.

Later in the proofs, function alg : $\mathscr{M}\times\Sigma\to\mathscr{M}$ is used to refer to the algorithm executed on each component in separation, such that each component first applies respawn and then progression with a projection of the global event (of each global trace element) on the local set of atomic propositions.

Example 5 (Decentralised choreography-based monitoring algorithm) We show how the running example would be monitored with the trace $\{a,c\}\cdot\{a,b\}\cdot\{b,c\}$. The network at runtime starts with $\mathrm{M}_{i,j}$ initialised with all formulae set at time zero as follows:

$$\text{Comp. 0 } (a\text{'s}):\ \{0\mapsto(\langle\!|2,0|\!\rangle\wedge(a\,\mathbf{U}\,(a\wedge\langle\!|1,0|\!\rangle)))\}\qquad \begin{array}{l}\text{Comp. 1 } (b\text{'s}):\ \{0\mapsto(b\wedge\langle\!|2,0|\!\rangle)\}\\ \text{Comp. 2 } (c\text{'s}):\ \{0\mapsto c\}\end{array}$$

Since none of the components would have received a message at the start of execution, step 5 of the algorithm is activated on event $\{a,c\}$. Upon applying respawning and progression on all components (simplification is suppressed at times to facilitate the understanding), the network results in the following:

$$\begin{array}{ll}\text{Comp. 0 } (a\text{'s}): & \{0\mapsto(\langle\!|2,0,0|\!\rangle\wedge(\langle\!|1,0,0|\!\rangle\vee(\text{true}\wedge a\,\mathbf{U}\,(a\wedge\langle\!|1,0|\!\rangle))))\}\\ \text{Comp. 1 } (b\text{'s}): & \{0\mapsto(\text{false}\wedge\langle\!|2,0,0|\!\rangle)\ ;\ 1\mapsto(b\wedge\langle\!|2,0|\!\rangle)\}\\ \text{Comp. 2 } (c\text{'s}): & \{0\mapsto\text{true}\ ;\ 1\mapsto c\}\end{array}$$

Next, step 7 updates the referent data structure but no kill signals are sent since all formulae are referred to. Subsequently, step 8 gets activated, leading the first item of the second and third components to be communicated as three messages: verdict$(0,0,\text{true},0)$, verdict$(1,0,\text{true},0)$, verdict$(0,0,\text{false},0)$ since $\langle\!|2,0,0|\!\rangle$ has two referrers. These messages in turn activate step 2 for the first component resulting in the following network:

Comp. 0 (a's): $\{0 \mapsto a\,\mathbf{U}\,(a \wedge \langle\!|1,0|\!\rangle)\}$ Comp. 1 (b's): $\{1 \mapsto (b \wedge \langle\!|2,0|\!\rangle)\}$ Comp. 2 (c's): $\{1 \mapsto c\}$

The following steps are similar. However, upon the third event, the *Until* fails and the main formula resolves to $\bot$. Step 6 is triggered, which in turn triggers step 4 in the other components resulting in the termination of all subformulae.

6 Guarantees and Algorithm Semantics

Following the formalisation of the progression of a choreographed monitoring network, we proceed to prove two main properties of the proposed choreography: the maximum number of nested placeholders and the correctness of the verdict reached.

6.1 Maximum Depth in a Network

For the purpose of guaranteeing the maximum number of nested distribution pointers in a choreographed LTL network, we define two depth-measuring functions: one which measures the maximum number of nesting levels in a formula, and another which measures the number of indirections in the network (typically starting from the main formula).

Definition 11 (Depth) The depth-measuring function $\mathrm{dpth} : \mathrm{LTL} \to \mathbb{N}$ is defined as:

$$\begin{aligned}\mathrm{dpth}(\varphi) = \text{match } \varphi \text{ with}\\ &\mid \sim\psi &&\to 1 + \mathrm{dpth}(\psi)\\ &\mid \psi \odot \psi' &&\to 1 + \max(\mathrm{dpth}(\psi), \mathrm{dpth}(\psi'))\\ &\mid \_ &&\to 1\end{aligned}$$

Assuming a network N and taking a formula φ as input, the function measuring the depth of nested distribution propositions, $\mathrm{dpth}_D : \mathrm{LTL}_D \to \mathbb{N}$, is defined as:[8]

$$\begin{aligned}\mathrm{dpth}_D(\varphi) = \text{match } \varphi \text{ with}\\ &\mid \langle\!|x,y|\!\rangle &&\to 1 + \mathrm{dpth}_D(N_{x,y})\\ &\mid \sim\psi &&\to \mathrm{dpth}_D(\psi)\\ &\mid \psi \odot \psi' &&\to \max(\mathrm{dpth}_D(\psi), \mathrm{dpth}_D(\psi'))\\ &\mid \_ &&\to 1\end{aligned}$$

Example 6 (Measuring depth) Referring back to the running example, given $c \wedge (a\,\mathbf{U}\,(a \wedge (b \wedge c)))$, function dpth returns 4. If we generate the corresponding network and evaluate function dpth_D on it, we get 3.

The maximum number of indirections in a network never exceeds the maximum number of nesting levels in a formula:

Proposition 1 (Maximum level of nested distributions) $\forall \varphi \in \mathit{LTL} \cdot \mathrm{dpth}_D(\mathrm{net}(\varphi)) \leq \mathrm{dpth}(\varphi)$, *where* $\mathrm{dpth}_D(\mathrm{net}(\varphi))$ *denotes* $\mathrm{dpth}_D(\lceil \mathrm{net}(\varphi) \rceil)$.

[8] Note that dpth_D operates on the untimed network and for this reason the timed distribution proposition is left out.

6.2 Soundness of Choreography

Following the result concerning depth, we turn our attention to proving the soundness of choreography. The idea is to show that the network is correctly initialised in that it reflects the structure of the formula (Proposition 2). Subsequently, in Lemma 1 we show that upon each progression step, the network remains consistent with the formula as if it had been progressed centrally under the same symbol. The network consistency relies on local and global consistency of progression steps: Local consistency means that each independent progression is correct while global consistency refers to the fact that respawning and messaging should maintain a network whose verdict corresponds to traditional LTL monitoring with a global view. Local and global consistency are reflected in the base case and the inductive case within the proof of the inductive case of Lemma 1, respectively. Next, we lift the result to a trace of progressions in Lemma 2 and, in Theorem 1 we show that the minimalistic messaging protocol adopted for optimisation is correct. Finally, Corollary 1 states the soundness of choreography: whenever the choreography algorithm produces a $\top$ or $\bot$ verdict on a trace, such a trace has the same evaluation under LTL_3.

Remark 7 (Consequence of formula simplification in choreography) We note that since formula simplification in a choreography is applied to local formulae in separation, there can be cases where a formula could be simplified only knowing the formulae on other components (which could be done only by an orchestration or migration algorithm). Hence, when comparing the formulae obtained with progression under choreography with the formulae that would have been progressed centrally, our results guarantee semantic equivalence (which we denote by the symbol $\simeq$) rather than syntactic equality.

To aid in the proof of correctness, we define function msg° which, assuming a choreography network and given an LTL_D formula, returns the (undistributed) LTL formula being monitored in the network by simulating a fully instantaneous communication where all submonitors are forced to communicate their current state to their parents.

Definition 12 (Instantaneous full messaging) Assuming a network memory M and its corresponding network N, function $\mathrm{msg}^\circ : \mathrm{LTL}_D \to \mathrm{LTL}$ is defined as:

$$\begin{array}{l}\mathrm{msg}^\circ(\varphi) = \text{match } \varphi \text{ with} \\ \quad \begin{array}{lll} \mid \langle\!\langle x,y,t \rangle\!\rangle & \to \mathrm{msg}^\circ(M^t_{x,y}) & \mid \psi \odot \psi' \to \big(\mathrm{msg}^\circ(\psi)\big) \odot \big(\mathrm{msg}^\circ(\psi')\big) \\ \mid \langle\!\langle x,y \rangle\!\rangle & \to \mathrm{msg}^\circ(N_{x,y}) & \mid \_ \quad\quad \to \varphi \\ \mid \sim \psi & \to \sim \big(\mathrm{msg}^\circ(\psi)\big) & \end{array}\end{array}$$

Informally, function msg° goes through a given formula and for each distribution proposition encountered, fetches the most recently updated valuation from the network: if the distribution proposition is untimed, the value will be the same as in the initialised network N; otherwise, the corresponding valuation from the network memory M is obtained. For brevity, we use $\mathrm{msg}^\circ(M)$ instead of $\mathrm{msg}^\circ(\lceil M \rceil)$ and $\mathrm{msg}^\circ(N)$ instead of $\mathrm{msg}^\circ(M')$, where $\forall i,j \cdot M'^0_{i,j} = N_{i,j}$.

As a step towards the main correctness proof, we show that generating a distributed network for a formula φ and forcing all local monitors to communicate everything to the main formula, results in the formula φ we started with:

Proposition 2 $\forall \varphi \in \mathit{LTL} \cdot \mathrm{msg}^\circ(\mathrm{net}(\varphi)) = \varphi$.

Intuitively, when starting from the main formula, msg° returns the main formula of the network as if all of its subformulae had instantaneously communicated their current status to

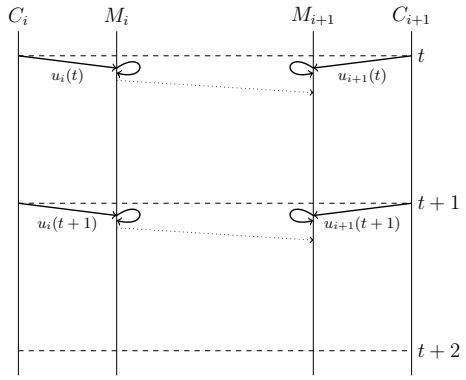


Fig. 5: Illustration of the timing model used in the implementation DecentMon.

their parents. While in practice having all the components communicating their exact status (rather than just the verdict) defeats the whole purpose of decentralised monitoring, we use this notion as a step towards proving correctness. The idea is that the result is achieved if (i) a formula is choreographed over the network, (ii) such a formula is processed using the algorithm, and (iii) all intermediate results are "communicated" through msg°. The outcome of our algorithm is the same as if the formula had been progressed using standard progression with a global view of events, as stated in the following lemma.

Lemma 1 (Correctness for one step under fully instantaneous communication)

$$\forall \varphi \in LTL, \forall \sigma \in \Sigma \cdot \text{prog}(\varphi, \sigma) \simeq \text{msg}^\circ(\text{alg}(\text{net}(\varphi), \sigma))$$

To lift the correctness of the above result to traces, we lift function alg to accept traces in the same way as we lifted prog (see the end of Definition 3). Note that we still apply messaging (through msg°) only once after all applications of alg. This still works since our approach does not rely on any guarantees on the arrival of messages.

Lemma 2 (Correctness under fully instantaneous communication)

$$\forall \varphi \in LTL, \forall u \in \Sigma^* \cdot \text{prog}(\varphi, u) \simeq \text{msg}^\circ(\text{alg}(\text{net}(\varphi), u))$$

The messaging model used so far (msg°) assumes fully instantaneous communication at each global clock tick, i.e., all child monitors report their state to their parent. While this model is useful to reason about the correctness of the network, in practice one would typically decide to communicate with lower frequency or when some verdict is resolved. Another choice, apart from the frequency of updates, is how many updates can be sent during a global time unit: if a verdict is received by a parent, causing the parent to also reach a verdict, can the parent send the verdict to the grandparent within the same time unit?

Remark 8 The model we chose to implement, mainly to have a comparable implementation to our previous work, communicates only verdicts and permits at most one communication step for every global clock tick (see Fig. 5 where progression are denoted by loops and communication by dotted arrows). As a consequence of this choice, a verdict may be delayed

at best and unreachable if one of the child monitors cannot reach a verdict[9] (even though it may well be possible to evaluate the global formula and reach a verdict).

To enable us to reason about the verdict-only, time-stepped, i.e., one step per time unit, we formalise the communication protocol below.

Definition 13 (Verdict-only communication) Assuming the value of a runtime network M and given a distributed formula, function $\mathrm{msg}^{\mathrm{v}} : \mathrm{LTL}_D \to \mathrm{LTL}_D$ returns the LTL formula with any verdicts available replacing the placeholders:

$$\begin{aligned}
\mathrm{msg}^{\mathrm{v}}(\varphi) = {} & \text{match } \varphi \text{ with} \\
& \mid \langle\!| x,y,t' |\!\rangle \to \text{match } \mathrm{msg}^{\mathrm{v}}(M^{t'}_{x,y}) \text{ with} \\
& \qquad \mid \top/\bot \to \top/\bot \\
& \qquad \mid _ \quad \to \langle\!| x,y,t' |\!\rangle \\
& \mid \sim\psi \quad \to \sim\big(\mathrm{msg}^{\mathrm{v}}(\psi)\big) \\
& \mid \psi \odot \psi' \to \big(\mathrm{msg}^{\mathrm{v}}(\psi)\big) \odot \big(\mathrm{msg}^{\mathrm{v}}(\psi')\big) \\
& \mid _ \quad \to \varphi
\end{aligned}$$

Since with verdict-only messaging, the main formula would generally still have "holes" to be filled in in the future, reasoning about the correctness of the protocol is more challenging than for the fully instantaneous one. To start with, we introduce the notion of assignment, which assigns an LTL formula to a subset of "hole"s.

Definition 14 (Assignment) An assignment, $A \in \mathscr{A}$ is a partial function where

$$\mathscr{A} = \{\langle\!| i,j |\!\rangle, \langle\!| i,j,t |\!\rangle \mid i,j,t \in \mathbb{N}\} \rightharpoonup \mathrm{LTL}.$$

Function $\mathrm{assign} : (\mathrm{LTL}_D \times \mathscr{A}) \to \mathrm{LTL}_D$ takes a distributed formula and an assignment, applies the assignment to the formula and returns the updated formula:

$$\begin{aligned}
\mathrm{assign}(\varphi, \mathrm{A}) = {} & \text{match } \varphi \text{ with} \\
& \mid \langle\!| x,y,t |\!\rangle \mid \langle\!| x,y |\!\rangle \to \begin{cases} \mathrm{A}(\varphi) & \text{if } \varphi \in \mathrm{dom}(\mathrm{A}), \\ \varphi & \text{otherwise,} \end{cases} \\
& \mid \sim\psi \quad \to \sim\big(\mathrm{assign}(\psi, \mathrm{A})\big), \\
& \mid \psi \odot \psi' \to \big(\mathrm{assign}(\psi, \mathrm{A})\big) \odot \big(\mathrm{assign}(\psi', \mathrm{A})\big), \\
& \mid _ \quad \to \varphi.
\end{aligned}$$

For brevity we use $\mathrm{A}(\langle\!| x,y,t |\!\rangle)$ instead of $\mathrm{assign}(\langle\!| x,y,t |\!\rangle, \mathrm{A})$.

Definition 15 (More defined formula) A distributed LTL formula φ is said to be more defined than, or equally defined to another formula φ', noted $\varphi \succeq \varphi'$, iff $\exists\, \mathrm{A} \in \mathscr{A} \cdot \mathrm{A}(\varphi') = \varphi$.

Example 7 (How defined a formula is) Consider formula $\varphi = \langle\!| 2,0 |\!\rangle \wedge (a \,\mathbf{U}\, (a \wedge \langle\!| 1,0 |\!\rangle))$ from the running example: $\varphi' = \top \wedge (a \,\mathbf{U}\, (a \wedge \langle\!| 1,0 |\!\rangle))$ is more defined than φ since one of the placeholders has been assigned, i.e., there exists an assignment $\mathrm{A} = \{\langle\!| 2,0 |\!\rangle \mapsto \top\}$ such that $\varphi' = \mathrm{A}(\varphi)$.

A formula $\varphi \in \mathrm{LTL}_D$ which is more defined than a formula with no distribution constructs, $\varphi' \in \mathrm{LTL}$, must be equal to the latter:

[9] This situation is related to the so-called notion of monitorability of formulae (cf. [5,11]). Intuitively, a formula is non-monitorable whenever there exists a trace that could lead a monitor to be unable to produce a verdict. There are cases where a formula is monitorable but its subformulae are not, e.g., $\mathbf{GF}(a) \wedge \neg(\mathbf{GF}(a))$ is monitorable although its subparts are both non-monitorable.

Proposition 3 (More defined formula) $\forall \varphi \in LTL_D, \forall \varphi' \in LTL \cdot \varphi \succeq \varphi' \implies \varphi = \varphi'$.

The notion of how defined a formula is, becomes useful in reasoning about different communication protocols: for example fully instantaneous communication would not leave any "holes" in the main formula, while verdict-only messaging would generally leave parts of the global formula undefined as the verdict for these parts would not have yet been reached. This reasoning is captured in the following lemma.

Lemma 3 $\forall M \in \mathscr{M} \cdot \text{msg}^{\circ}(M) \succeq \text{msg}^{\text{v}}(M)$.

The above lemma states that for all possible network memories, full messaging yields more defined formulae than verdict-only, time-stepped messaging.

Theorem 1 (Correctness of verdict-only time-stepped messaging) *If a verdict is reached when using verdict-only time-stepped messaging, then the verdict is correct:*

$$\forall \varphi \in LTL, \forall \text{u} \in \Sigma^* \cdot \text{msg}^{\text{v}}(\text{alg}(\text{net}(\varphi), \text{u})) \in \{\top, \bot\} \implies \text{msg}^{\text{v}}(\text{alg}(\text{net}(\varphi), \text{u})) \simeq \text{prog}(\varphi, \text{u}).$$

As mentioned before, we note that there is no guarantee that the verdict is actually reached due to the issue of non-monitorable subformulae and due to the fact that a subformula might never reach a verdict over a finite trace while the main formula would have due to simplification.

Summing up the above, the verdict of choreographed monitors is the verdict reached by the topmost monitor:

Definition 16 (Verdicts of the decentralised monitoring algorithm) The verdicts of the decentralised monitoring algorithm are given by function $\text{verdict}_{chor} : \Sigma^* \times \text{LTL} \to \{\top, \bot, ?\}$ defined as follows: let $u \in \Sigma^*$ and $\varphi \in \text{LTL}$,

$$\text{verdict}_{chor}(u, \varphi) \stackrel{\text{def}}{=} \begin{cases} \top \text{ if } \text{msg}^{\text{v}}(\text{alg}(\text{net}(\varphi), u)) = \top \\ \bot \text{ if } \text{msg}^{\text{v}}(\text{alg}(\text{net}(\varphi), u)) = \bot \\ ? \text{ otherwise} \end{cases}$$

Relating the above verdicts of the decentralised monitoring algorithm to the verdicts assigned according to LTL_3 (see Definition 2), we have the following corollary:

Corollary 1 (Correspondence of verdicts)

$$\forall u \in \Sigma^*, \forall \varphi \in LTL \cdot \text{verdict}_{chor}(u, \varphi) \in \{\top, \bot\} \implies \text{verdict}_{chor}(u, \varphi) = u \models_3 \varphi.$$

The above corollary states that if the decentralised monitoring algorithm assigns a verdict to a trace-formula pair, then the verdict assigned according to LTL_3 is the same.

7 Evaluation and Discussion

We present the evaluation of the choreography-based decentralised monitoring algorithm. Our empirical evaluation is based on the tool DecentMon[10] used in a previous study comparing orchestration with migration [6]. We significantly extended DecentMon with an implementation of the choreography algorithm and extensions allowing us to compare the three decentralised monitoring algorithms along some (new) monitoring metrics.[11]

[10] `http://decentmonitor.forge.imag.fr`

[11] The new implementation is available at: `http://decentmon3.forge.imag.fr`.

Section 7.1 presents the evaluation criteria, that is, the considered monitoring metrics. Sections 7.2 overviews the conducted experiments and their objectives. Section 7.3 states the hypothesis used in the experiments and presents how these were setup. Section 7.4 compares choreography to migration and Section 7.5 compares choreography to orchestration. We refrain from comparing orchestration to migration as this has already been carried out extensively in [6] and our experiment results confirm the conclusions in [6]. Section 7.6 summarises the detailed conclusions drawn from the comparisons made in Sections 7.4 and 7.5. Appendix B contains additional plots for the complementary visualisation of the same experimental results.

7.1 Monitoring Metrics for Decentralised Monitoring

Numerous criteria can be considered for comparing different organisations of LTL monitoring over a network. We ignore implementation-dependent measurements such as the actual overhead of monitors and focus on metrics related to decentralised monitoring. Below are a number of them which are treated in this study:

Delay. Because of the network organisation and the need of monitors to communicate, it takes some communication steps to propagate intermediate results. The delay of migration and choreography monitoring is the difference between the length of the trace needed to reach a verdict with these algorithms and the length of the trace needed to reach a verdict if all events were available at a central location, i.e., the extra time instants required for the network to reach a verdict due to communication. The lengths of the traces needed to reach a verdict are in columns $|\mathrm{tr}|$ in the experiment tables (and can thus be used to obtain the delay of each algorithm).

Number and size of messages. Since none of the components in the network can observe the full behaviour of the system, components have to communicate. Thus, we measure the number and size of required messages, reported in columns #msg and $|\mathrm{msg}|$ in the experiment tables, respectively.

Progressions. The configuration of the network affects the number of progressions[12] that need to be carried out, reported in columns #prog in the experiment tables.

For each of these metrics, the experiment tables report the average values, the standard deviation, and the average of the values per event (i.e., the average of the metric value to trace length ratio).

7.2 Conducted Experiments and their Objectives

In the following, we describe the three main experiments that we carried out and the objectives of these experiments.

First experiment: investigating the effect of different network structures with varying formula sizes (see Table 2 and Figs. 6, 7, 8, and 9 in Appendix B). The first experiment varies the size of the formulae (indicated by $|\varphi|$). Note that we measure the size of a formula in terms of its maximum nesting of operators. This measurement turned out to reflect more

[12] Since the number of progressions is also influenced by the formula simplification procedure (see Remark 2), we use the same formula simplification procedure for the three monitoring algorithms.

Table 2: Report on the experiment varying the size of formulae ($|\varphi|$) with alphabet $\{a,b,c\}$. Column bias indicates whether the generation of random formulae is biased (✓) or not (✗). Each group of three lines reports the values obtained after the tests of 1,000 (freshly generated) formulae against (freshly generated) traces of length 100: the first (resp. second, third) line reports the average values (resp. standard deviations, the average values per event).

Variables		Orchestration			Migration				Choreography			
$\|\varphi\|$	bias	|tr|	#msg	#prog	|tr|	#msg	|msg|	#prog	|tr|	#msg	|msg|	#prog
1	✗	2.428	4.284	4.725	3.473	1.262	0.863	8.311	2.482	0.147	0.128	5.002
		(1.04)	(3.12)	(2.66)	(1.04)	(1.4)	(1.01)	(9.62)	(1.05)	(0.55)	(0.44)	(3.26)
			[1.648]	[1.917]		[0.343]	[0.25]	[2.173]		[0.048]	[0.046]	[1.942]
	✓	2.513	4.539	4.546	3.513	1.04	0.672	6.6	2.513	0	0	4.546
		(1.11)	(3.33)	(2.48)	(1.11)	(0.99)	(0.71)	(5.41)	(1.11)	(0)	(0)	(2.48)
			[1.675]	[1.76]		[0.284]	[0.198]	[1.707]		[0]	[0]	[1.76]
2	✗	6.553	16.66	29.9	7.861	4.376	2.302	80.18	6.94	3.431	1.042	36.86
		(2.38)	(7.16)	(23.34)	(2.56)	(3.37)	(2.92)	(85.32)	(2.57)	(4.85)	(1.33)	(33.62)
			[2.503]	[4.502]		[0.559]	[0.312]	[9.948]		[0.424]	[0.141]	[5.056]
	✓	6.069	15.21	18.83	7.073	2.142	0.874	33.76	6.081	0.089	0.034	19.07
		(1.5)	(4.51)	(11.51)	(1.5)	(1.0)	(0.73)	(20.31)	(1.5)	(0.62)	(0.23)	(12.02)
			[2.482]	[3.106]		[0.313]	[0.131]	[4.764]		[0.014]	[0.005]	[3.134]
3	✗	10.72	29.16	107.7	12.42	8.362	4.97	290	11.51	13.99	2.902	143.2
		(3.99)	(11.97)	(107.2)	(4.21)	(6.34)	(6.56)	(357.8)	(4.16)	(15.08)	(2.49)	(150.9)
			[2.696]	[9.657]		[0.671]	[0.422]	[22.52]		[1.109]	[0.25]	[11.77]
	✓	9.584	25.75	61.721	10.67	3.067	1.34	102.6	9.693	1.54	0.355	66.15
		(2.33)	(7.0)	(60.84)	(2.38)	(3.29)	(2.17)	(97.99)	(2.39)	(4.18)	(0.93)	(68.21)
			[2.674]	[6.334]		[0.287]	[0.13]	[9.481]		[0.138]	[0.033]	[6.653]
4	✗	15.35	43.06	291	17.32	12.97	9.927	780.1	16.45	33.46	5.134	420.2
		(7.26)	(21.8)	(344.5)	(7.42)	(9.58)	(14.35)	(1054)	(7.39)	(40.5)	(3.97)	(516.9)
			[2.783]	[18.29]		[0.744]	[0.613]	[43.61]		[1.864]	[0.319]	[24.16]
	✓	14.03	39.1	182.7	15.24	5.806	3.295	320.7	14.29	6.849	1.19	202.7
		(4.39)	(13.18)	(212.1)	(4.45)	(7.05)	(7.22)	(448.3)	(4.45)	(13.27)	(1.92)	(234.6)
			[2.772]	[12.6]		[0.379]	[0.225]	[20.47]		[0.441]	[0.082]	[13.7]
5	✗	20.22	57.65	653.3	22.341	18.26	21.22	1958	21.5	59.8	7.543	955
		(9.64)	(28.93)	(803.9)	(9.73)	(11.65)	(44.86)	(3694)	(9.75)	(57.1)	(5.21)	(1216)
			[2.833]	[32.05]		[0.826]	[1.062]	[88.75]		[2.647]	[0.371]	[43.14]
	✓	18.24	51.72	397.7	19.62	9.032	6.719	779.8	18.61	15.48	2.137	441.7
		(5.68)	(17.03)	(431.7)	(5.75)	(11.83)	(15.15)	(1330)	(5.78)	(23.7)	(2.77)	(475.8)
			[2.825]	[21.32]		[0.434]	[0.342]	[37.49]		[0.758]	[0.112]	[23.08]

how difficult it is to monitor a formula in a choreographed fashion (as the results show). As such, a randomly generated formula reported to be of size n contains between n and $2^n - 1$ symbols (including atomic propositions) depending on the operators selected during its generation. This experiment enabled us to assess the scalability of the approaches and how they perform on different network structures. In particular, we considered two kinds of networks: one whose formula is generated purely randomly, and another where we biased the formula generator such that the bottommost LTL operators always have operands from the same component; essentially emulating networks where the basic subformulae of an LTL formula can be evaluated without communicating.

Table 3: Report on the experiment varying the pattern of formulae with alphabet $\{a,b,c\}$. The patterns are: absence (abs), existence (exis), universal (unive), precedence (prec), response (resp), precedence chain (pchain), response chain (rchain), constrained-chain pattern (cchain). Each group of three lines reports the values obtained after the tests of 1,000 (freshly generated) formulae against (freshly generated) traces of length 100: the first (resp. second, third) line reports the average values (resp. standard deviations, the average values per event).

Pattern	Orchestration			Migration				Choreography			
	|tr|	#msg	#prog	|tr|	#msg	|msg|	#prog	|tr|	#msg	|msg|	#prog
abs	16.92	47.76	338.8	19.15	13.00	9.202	801.2	17.89	48.50	7.332	444.6
	(5.47)	(16.42)	(119.6)	(5.65)	(5.18)	(3.95)	(336.0)	(5.67)	(25.11)	(2.3)	(173.5)
		[2.808]	[19.94]		[0.681]	[0.51]	[41.54]		[2.631]	[0.425]	[24.59]
exist	22.398	64.194	470.8	24.61	17.88	10.05	1104	23.20	63.21	7.545	577.0
	(11.24)	(33.74)	(265.4)	(11.31)	(8.87)	(4.04)	(600.7)	(11.29)	(42.56)	(2.79)	(328.5)
		[2.84]	[20.70]		[0.728]	[0.47]	[44.31]		[2.611]	[0.361]	[24.32]
unive	22.58	64.73	432.9	24.46	15.25	7.171	841.6	23.44	52.87	6.327	539.0
	(12.46)	(37.38)	(249.4)	(12.41)	(7.12)	(2.71)	(415.2)	(12.48)	(37.64)	(2.66)	(313.8)
		[2.837]	[19.00]		[0.642]	[0.357]	[34.94]		[2.197]	[0.308]	[22.69]
prec	17.19	48.57	369.1	19.50	12.53	8.316	789.4	18.42	59.26	8.707	491.3
	(6.28)	(18.85)	(172.1)	(6.38)	(4.89)	(3.76)	(365.5)	(6.35)	(35.82)	(3.63)	(235.1)
		[2.809]	[21.18]		[0.648]	[0.456]	[40.08]		[3.11]	[0.491]	[26.22]
resp	27.36	79.08	1212	29.66	22.67	27.13	3498	28.18	87.62	8.974	1306
	(17.29)	(51.87)	(1004)	(17.31)	(12.75)	(12.84)	(2643)	(17.29)	(63.56)	(2.94)	(1020)
		[2.855]	[41.29]		[0.78]	[1.123]	[112.7]		[3.046]	[0.388]	[43.65]
pchain	24.76	71.28	1326	26.99	18.22	24.86	3238	26.31	103.2	10.66	3582
	(16.08)	(48.24)	(1372)	(16.08)	(10.65)	(17.82)	(3226)	(16.2)	(93.59)	(4.99)	(9556)
		[2.841]	[47.50]		[0.687]	[1.073]	[108.2]		[3.638]	[0.47]	[99.91]
rchain	18.09	51.26	951.9	20.33	16.88	36.46	3183	19.55	74.38	9.8	1529
	(7.71)	(23.15)	(855.2)	(7.8)	(8.38)	(31.24)	(3158)	(7.91)	(68.89)	(6.03)	(1727)
		[2.814]	[49.55]		[0.843]	[1.87]	[146.3]		[3.455]	[0.505]	[70.91]
cchain	20.42	58.27	1141	22.86	18.7	35.31	3266	22.40	105.8	13.13	1485
	(10.22)	(30.68)	(880)	(10.31)	(8.82)	(21.35)	(2308)	(10.41)	(78.25)	(6.6)	(1238)
		[2.828]	[56.35]		[0.825]	[1.766]	[143.3]		[4.589]	[0.642]	[65.42]

Second experiment: investigating how the approaches handle realistic specifications (see Table 3, and Figs. 10, 11, 12, and 13 in Appendix B). The second experiment varies the pattern [8] of the formulae. This experiment enabled us to assess the behaviour of the approaches when faced with realistic specifications. We generated LTL formulae against all of the so-called pattern mappings[13], namely the absence, existence, universal, precedence, response, precedence chain, response chain, and constrained-chain patterns. Each pattern mapping corresponds to between 5 and 10 "shapes of formulae".

Third experiment: investigating how varying the number of components and the number of redirections in the resulting LTL network effect the approaches (see Table 4). The experi-

[13] The exact definitions of the pattern mappings and associated LTL formulae are available at `http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`.

Table 4: Varying the number of components (i.e., size of the distributed alphabet, $|\Sigma_d|$) and distribution depth (dep.) for the same formulae and traces. Column # reports the number of formulae falling in each category. The considered alphabet is $\{a,b,c\}$. Each group of two lines reports the average values (first line) and the standard deviations (second line).

Variables				Orchestration			Migration				Choreography			
$\|\varphi\|$	$\|\Sigma_d\|$	dep.	#	|tr|	#msg	#prog	|tr|	#msg	|msg|	#prog	|tr|	#msg	|msg|	#prog
3	2	1	2,246	1.109	2.218	30.8	2.109	0.473	2.606	33.08	1.109	0	0	30.8
				(2.18)	(4.36)	(38.11)	(2.18)	(0.49)	(3.83)	(52.28)	(2.18)	(0)	(0)	(38.11)
		2	13,864	1.043	2.086	34.68	2.414	1.81	5.39	52.60	1.554	2.001	1.652	39.18
				(1.89)	(3.79)	(37)	(1.99)	(2.21)	(7.11)	(92.52)	(1.97)	(2.91)	(1.35)	(41.43)
		3	12,611	1.037	2.074	35.42	2.375	1.735	5.46	52.43	2.059	4.149	2.657	53.56
				(1.88)	(3.77)	(37.6)	(1.98)	(2.04)	(7.47)	(89.4)	(2.08)	(4.97)	(1.81)	(59.45)
		4	424	0.933	1.867	39.856	2.271	1.82	6.249	58.528	2.233	5.478	3.292	68.431
				(1.95)	(3.91)	(50.65)	(2.02)	(2.71)	(8.07)	(120.42)	(2.28)	(7.75)	(2.15)	(82.04)
	3	2	4,732	1.076	3.229	33.51	2.823	3.539	7.622	116.9	1.803	3.59	3.118	41.19
				(1.99)	(5.98)	(37.1)	(2.18)	(3.12)	(9.07)	(175.79)	(2.05)	(4.27)	(1.89)	(44.22)
		3	4561	1.03	3.092	35.55	2.705	3.344	8.559	125.90	2.21	5.736	3.947	57.893
				(1.87)	(5.61)	(38.39)	(2.07)	(3.08)	(10.55)	(189.1)	(2.06)	(7.04)	(2.46)	(66.21)
		4	420	0.842	2.528	40.97	2.547	3.326	11.13	153.2	2.373	7.904	4.908	76.804
				(1.59)	(4.78)	(34.37)	(1.81)	(2.9)	(13.47)	(212.7)	(1.88)	(8.01)	(3.08)	(86.64)
		5	14	0.571	1.714	60.78	2.142	2.714	22.17	277.1	2.428	12.21	6.88	131.6
				(0.49)	(1.47)	(15.57)	(0.79)	(2.04)	(12.99)	(148.84)	(1.69)	(6.78)	(3.68)	(87.6)
5	2	1	1,466	1.881	3.762	69.33	2.881	0.6	4.262	84.54	1.881	0	0	69.33
				(3.55)	(7.11)	(124.69)	(3.55)	(0.48)	(6.1)	(147.47)	(3.55)	(0)	(0)	(124.69)
		2	11,538	1.801	3.602	94.201	3.143	2.358	12.77	173.11	2.283	3.276	1.969	101.83
				(2.99)	(5.99)	(169.45)	(3.08)	(3.19)	(21.11)	(415.13)	(3.06)	(5.24)	(1.54)	(173.9)
		3	11,546	1.815	3.63	101.84	3.149	2.366	13.915	190.455	2.777	7.145	3.481	149.633
				(3.06)	(6.12)	(185.17)	(3.16)	(3.31)	(23.39)	(482.9)	(3.29)	(11.24)	(2.65)	(421.2)
		4	710	1.86	3.721	122.474	3.185	2.833	18.321	245.9	3.126	9.704	4.325	193.9
				(2.98)	(5.97)	(223.62)	(3.09)	(3.53)	(27.07)	(567.27)	(3.3)	(13)	(3.02)	(313.34)
	3	2	3,296	1.767	5.303	83.83	3.491	4.054	15.83	309.6	2.46	5.464	3.511	96.05
				(2.76)	(8.3)	(151.5)	(2.97)	(3.8)	(24.52)	(609.4)	(2.84)	(7.6)	(2.12)	(158.58)
		3	4,182	1.813	5.441	96.65	3.492	4.162	19.08	376.49	2.951	9.379	4.891	150.64
				(3.15)	(9.46)	(170.5)	(3.34)	(4.5)	(28.31)	(825.3)	(3.36)	(13.21)	(3.19)	(357.8)
		4	873	1.993	5.979	141.2	3.78	4.878	29.60	590.13	3.507	15.45	6.724	260.1
				(3.61)	(10.84)	(245.2)	(3.84)	(5.04)	(43.34)	(1,120)	(3.95)	(23.37)	(4.19)	(676.5)
		5	130	1.807	5.423	169.6	3.5	4.623	36.27	757.7	3.538	18.03	7.84	323.8
				(3.18)	(9.55)	(294.61)	(3.41)	(5.73)	(47.23)	(1,738.4)	(3.69)	(24.66)	(4.75)	(671.9)
		6	11	2.909	8.727	313.2	4.454	6.181	43.76	1,653.	4.09	18.09	6.995	912
				(4.9)	(14.72)	(605.2)	(5.39)	(7.73)	(49.8)	(4,041.9)	(5.45)	(28.31)	(5.02)	(2,270)

ment has been carried out with formulae of sizes 3 and 5. This experiment is crucial since the migration approach is sensitive to the size of the network [6], while intuitively we expect the choreography approach to be affected by the depth of the LTL network. In this experiment, for a given alphabet, we generate all possible distributions of this alphabet amongst three components, hence giving the effective number of components in the monitoring network. We grouped the experiments according to the number of components and then the depth of

the associated network. (We also report the number of formulae falling in each category). Hence, for 10,000 formulae, the tool generated 30,000 tests for the possible (distributed) alphabets of size 2 (considering only distributed alphabets that are not isomorphic) and 10,000 tests for the possible distributed alphabets of size 3 (meaning that there are 3 distinct distributed alphabets of size 2 and one of size 3 when considering an alphabet of size 3).[14] For a given alphabet, we do not report on the experiments where the number of components was 1 as it amounts to centralised monitoring.

7.3 Experiment Setup

In this subsection we describe some choices that needed to be made with respect to the architectural setup of the experiments.

Benchmark generation. For the first experiment, since the considered variable was the size of the formulae, for each formula size and each kind of network, we generated 1,000 formulae, each monitored against a freshly generated trace of length 100. For the second experiment, since the variable considered was the pattern of the formulae, for each pattern mapping, we generated 1,000 formulae, each monitored against a freshly generated trace of length 100. For the third experiment, for each formula size, we first generated 10,000 LTL formulae and traces randomly, subsequently tweaking the alphabet to manipulate the number of referenced components and the depth of the resulting LTL network. Note that for the third experiment, we needed more sample formulae to ensure enough tests under all depths.

It is important to note that our tables report results obtained with only monitorable properties. We considered the notion of monitorability as defined in [26] (and characterised in [5, 11]) which intuitively states that a property is monitorable if there is always a trace that leads to a definitive verdict. Monitorable properties represent the actual properties that are sensible to monitor in practice. Hence, the lengths of the traces in our experiments are relatively small because we discarded all experiments where a verdict could not be reached (in either centralised or decentralised approach) due to non-monitorability. Should we have kept the results for non-monitorable properties, we would have augmented the observed numbers in all metrics artificially without much difference in the trends.

Communication protocol. Choosing a communication protocol such as communicating only the propositions which are true while assuming that unsent ones are false, makes a significant difference to our results. The chosen protocols were as follows: In the case of orchestration, each component sends a bit vector consisting of the values of its atomic propositions referenced in the formula. In the case of migration, since the whole formula is sent, it is less straightforward to gain quick savings as in the case of propositions. Thus, in this case we measure the size of the formula and use it as the size of the message. To measure the size of a formula, (i) we consider the total number of symbols (including those from the alphabet) that can appear in the formula to obtain the number of bits needed to encode one symbol, and then (ii) we count the number of symbols needed to encode the abstract syntax tree of the formula. In the case of choreography we have two kinds of messages: verdict messages of the form $\langle i, j, v, t' \rangle$ (from components to their referrer formulae) and kill messages of the form $\mathrm{kill}(x, y)$ (from components to their referent formulae). The former kind are similar

[14] For the alphabet $\{a, b, c\}$, the distinct distributed alphabets of size 2 are $\{a|b,c\}$, $\{b|a,c\}$, $\{c|b,a\}$, the unique distributed alphabet of size 3 is $\{a|b|c\}$, where | separates atomic propositions on distinct components. For instance, $\{a|b,c\}$ denotes the distributed alphabet with two components where proposition a is observed on the first component and propositions b and c are observed on the second component.

to those of orchestration but are transmitted only when a verdict has been reached. Each message counts as twice the number of bits needed to encode a coordinate, one bit for the verdict, and the number of bits needed to encode the current time stamp.

Execution cycles. A major difference between choreography and migration is that, in the latter, components could send one message each per cycle while in the case of our new choreography algorithm, there is no assumption on the communication frequency of components. However, the picture is even more complex because the progression within a component may depend on the verdict of others. Thus, while migration (as in [6]) strictly allowed one progression and messaging cycle per system cycle, the choreography algorithm introduced in this paper does not make any assumption on the number of cycles. This makes the choreography approach general and versatile to many settings: on the one hand of the spectrum, a delay-free approach without reference to history but relatively more expensive in terms of the number of cycles and the messages required for each system cycle, and on the other hand of the spectrum, an unbounded-delay approach with components communicating at their own rate and in an asynchronous manner.

To have a fair comparison between migration and choreography, we restrained the choreography algorithm to have one messaging cycle per system cycle, as in the case of migration. In the following subsections, we discuss the outcome by first comparing choreography with migration, and subsequently comparing choreography with orchestration. Note that all experiments reported in the upcoming subsections are public and reproducible through DataMill.[15] Moreover, the choreography algorithm proposed in this paper is better than the one initially proposed in [7] on a number of levels: (i) it is more practical as it does not make unrealistic assumptions such as instantaneous communication and garbage collects non-referenced cells in the network, i.e., cells which do not have any referents, (ii) it is more general since it does not impose synchronisation between the system and the monitor, and (iii) it runs faster mainly because runtime network maintenance is reduced.

7.4 Choreography and Migration

We start by comparing the choreography approach to the migration approach by considering each criterion in turn.

Delay. We observe that in all cases, the delay induced by decentralised monitoring is low, and remains lower than 2.0 on average. Except for the cases of randomly generated formulae with a network of depth 5 (see Table 4), the delay induced by choreography is lower than the one induced by migration. Biasing the generation of formulae has a positive impact on the delay as it is reduces it both in migration and choreography. However, the reduction factor in choreography is more important in choreography than in migration. The delay in migration seems not to be influenced by an increase in the size of formulae, while it slightly augments

[15] DataMill is a platform for rigorous and reproducible experiments. The reported numbers are available as two DataMill benchmarks at `https://datamill.uwaterloo.ca/experiment/X/` where `X` is 1185, 1649, and 1651. The interested reader can also examine other benchmarks carried out for this paper. These benchmarks evaluate different aspects such as different alphabets, probability distributions for traces, average and maximum delay of decentralised monitoring. Because of page limitation, we do not report the numbers of these experiments in this paper, but the numbers are publicly available as benchmark numbers 1298, 1299, 1300, and 1301 on the DataMill website. Moreover, the source code of the benchmarks is available inside the experiment archives.

in choreography. Moreover, contrary to migration, the delay in choreography is sensitive to the depth of the network.

Number and size of messages. A significant difference between choreography and migration is that in migration the whole formula is transmitted over the network while in choreography only when a subformula reaches true or false is the verdict transmitted. This distinction contributes to the significant difference in the size of the messages sent, which are lower in choreography than in migration, as can be seen in the experiment tables and plots. Both choreography and migration have their size of messages growing with the size of formulae, depth and size of the network. Yet, the growing factor induced by the size of formulae seems slightly more in the case of migration than in the case of choreography.

The situation is however reversed in the case of the number of messages. Indeed, in choreography, the network has to propagate both the verdicts for subformulae and also the kill messages. The higher number of messages is expected since in the case of choreography, in terms of monitoring information pertaining to the evaluation of formulae, only verdicts corresponding to the evaluation of subformulae are transmitted and thus contain less information than in the case of migration where entire formulae are transmitted.

As part of the evaluation, we changed the number of components involved in a formula while keeping everything constant. Unsurprisingly, changing the number of components did not affect the performance of the choreography approach as much as it affected the performance of the migration approach. Table 4 shows this clearly: for a given formula size, augmenting the number of components induces the compound size of messages and the number of progressions to augment by a bigger factor in the migration approach. The results for choreography still fluctuated[16] but not clearly in any direction and less than a factor of two in the worst case.

Similarly, keeping everything constant, we altered the alphabet once more, this time keeping the number of components constant but changing the number of indirections required in the choreography, i.e., a deeper tree of monitors. Again, the results in Table 4 confirm the intuition that this change affects the choreography much more than the migration approach. Moreover, the trends generally observed for random formulae and specification patterns are even more pronounced.

Progressions. The variations in the number of progressions is similar to the number of messages sent/received. The two are linked indirectly in the sense that both the number of messages and progressions increase if the monitoring activity in the network increases. However, we note that the required number of progressions is much lower in choreography than in migration in most of the cases (except for formulae from the precedence chain pattern).

7.5 Choreography and Orchestration

In this subsection, we compare the choreography and the orchestration approaches.

Delay. Since orchestration is a special case of choreography with depth one, the delay of an orchestration is always fixed at one (not shown in the table) and as such better than or as good as that of a choreography. However, the delay induced by choreography is generally

[16] The reasons for the fluctuations are probably due to the random adaptations of the alphabet to change the number of components a formula is based upon.

lower than one event on average compared to orchestration (except for non-biased randomly generated formulae of size 5 and complex patterns, i.e., the precedence, precedence chain, response chain, and constrained chain patterns).

Number and size of messages. Similarly to the case of delay, in general (as shown in the empirical results) the number of messages required by an orchestration is higher than that required by a choreography. In terms of required number of messages, choreography generally outperforms orchestration, except for the precedence, response, and constrained-chain patterns, for non-biased randomly generated formulae of size 5, and for deep networks, for which the results are slightly in favour of orchestration. However, the measured performance greatly depends on the topology of the tree. For example, having a distributed subformula $b_1 \wedge b_2$, sending updates for the conjunction is generally cheaper than sending updates for b_1 and b_2 separately. This phenomenon is hinted at in Table 4 where the results of the performance of choreography degrades for deeper networks. In other words, the performance of choreography is greatly dependent on how much the leaves can propagate their results towards the root of the tree without having to communicate. The hint is then confirmed in Table 2 where we intentionally biased the formula generation algorithm such that propositions from the same component are more likely to appear on the same branch. The results show a significant gain for the choreography approach.

As for the size of messages, for orchestration with alphabet $\{a,b,c\}$ they are of size 1. We can observe that for both patterns and non-biased randomly generated formulae, messages are smaller with orchestration, except for small formulae for which choreography performs better. This later behaviour is as expected, because for small formulae, the main monitor generally requires little information from other monitors. Moreover, for biased and randomly generated formulae, choreography performs better than orchestration for reasons similar to the ones for small formulae. Finally, we note that generally, the size of messages varies according to the pattern of the monitored formula, and that the size of messages seems to grow gently with the size of the monitored formula.

Progressions. As for the number of progressions, choreography generally requires a much higher number of progressions to reach a verdict. This behaviour is as expected because orchestration progresses the currently monitored formula with the value of all atomic propositions at hand, whereas choreography, on a given component, may have duplicate formulae over different time instants and has to progress pointers for the parts of the formula that are monitored on other components.

7.6 Summary of the Conclusions Drawn from the Experiments

Clearly, none of the approaches ticks all the boxes. In the following, we summarise the main trends and shed some light as to when it makes more sense to use one approach over another depending on the importance of the monitoring metrics in a given scenario. Recall that conclusions are based on average values. More details on the experiments can be found in Appendix B.

Delay. Orchestration generally offers better results than choreography which in turn offers better results than migration. Moreover, biasing the generation of random formulae significantly improves the performance of choreography (obtaining the same results as orchestration in a few cases).

Number of messages. Migration performs better than choreography which in turn performs better than orchestration, noting a couple of exceptions where orchestration performs better than choreography: for non-biased randomly generated formulae of size 5, or when the depth of the network is greater than or equal to 3.

Size of messages. Choreography performs much better than migration. Let us note one exception: in the case of formulae generated from the precedence pattern performs slightly better than choreography. Orchestration generally performs better than choreography. There are however three exceptions where choreography performs better than orchestration: for randomly generated formulae of size lower than or equal to 2, for randomly generated formulae of size lower than or equal to 3 when the network is of size lower than or equal to 2 and its depth lower than or equal to 3, and for randomly generated formulae of size 5 with a network of size 2 and a depth lower than or equal to 2.

Number of progressions. Orchestration generally offers better results than choreography which in turn offers better results than migration. Let us note that for randomly generated formulae with a network of size 2 and depth greater than or equal to 3, migration performs better than choreography.

Additional conclusions and discussion. When measuring the performance of the monitoring algorithms in terms of the above metrics, the standard deviation was higher for migration than choreography, which in turn was higher than the one of orchestration. The standard deviation numbers are relatively high. However, this does not reduce the usefulness of the results allowing to exhibit trends when comparing the approaches. Moreover, reasons for the relatively-high standard deviations are twofold. First, as one can expect, when fixing a formula size or a specification pattern, our benchmarks monitor diverse formulas. For instance, formulae $a \wedge b \wedge c$ and $\mathbf{G}(a \wedge b)$ are both of size 2, which is the depth of the parse tree. Nevertheless, these two formulae can potentially require different trace lengths to reach a verdict. Second, as our complementary visualisation plots in Appendix B indicate, the distributions of the values obtained for the monitoring metrics are neither normal nor symmetric. Hence, the standard deviation is a less appropriate measure of the variability, and the inter-quartile range (which can be obtained from the box plots in Appendix B) should preferably be used for that purpose.

Consequently, in practice, the values that a system designer shall expect when using the algorithms shall vary significantly according to the various parameters involved in decentralised monitoring (and more importantly according to the monitored specification and structure of the network). Hence, when given some formulae to monitor in a decentralised fashion, DecentMon should be useful to the system designer to simulate the execution of monitors for the formulae at hand, and determine the most suitable monitoring algorithm.

Another observation is that the implemented choreography algorithm is based on the greedy approach of distributing the formula into a network (see Definition 5). We expect that more sophisticated distribution approaches would give better results and therefore the numbers presented in paper for choreography can be improved.

Finally, let us recall that, although orchestration provides better results in a majority of the considered metrics, it involves introducing a central observation point in the system where all the monitoring computation is carried out.

8 Related Work

Splitting the progression of an LTL formula into subparts and propagating the results across a network is somewhat similar to the ideas used in parallel prefix networks [16]. In such networks intermediate results are evaluated in parallel and then combined to achieve the final result more efficiently. Furthermore, this work has two other main sources of inspiration: the work by Bauer and Falcone [6] about monitoring LTL properties in the context of distributed systems having a global clock, and the work by Francalanza et al. [13] which classifies modes of monitoring in the context of distributed systems. We have thus adapted the classification of distributed monitoring showing how orchestration, choreography, and migration can be applied to LTL monitors. We note, however, that we have introduced the global clock assumption which is not present in [13]. Without this assumption, our correctness theorem does not hold due to the loss of the total order between system events. From another point of view, we have classified the approach presented in [6] as a migration approach and extended the work by presenting a choreography approach. Furthermore, we have also empirically compared the approaches on a number of criteria.

As pointed out in [6, 10], decentralised monitoring is related to several monitoring techniques. We recall some of them and refer to [6] for a detailed comparison. One of the closest approaches is [31] which monitors MTTL formulae specifying the safety properties over parallel asynchronous systems. Contrary to [31], our approach considers the full set of ("off-the-shelf") LTL properties, does not assume the existence of a global observation point, and focuses on automatic splitting of an LTL formula according to the architecture of the system.

Also closely related to this paper is a monitoring approach of invariants using knowledge [14]. This approach leverages an apriori model-checking of the system to pre-calculate the states where a violation can be reported by a process acting alone. Both [14] and our approach try to minimise the communication induced by the distributed nature of the system but [14] (i) requires the property to be stable (and considers only invariants) and (ii) uses a Petri net model to compute synchronisation points.

Furthermore, the idea of splitting a formula to obtain a compositional evaluation procedure resembles in principle the approach in [27] where formulae are monitored on circuits in a compositional manner using the so-called notion of temporal tester.

As mentioned in the introduction, this paper is a revised and extended version of [7]. The main improvements of this paper compared to [7] are two-fold. First, in [7], between any two time instants, the monitoring network needed to be reconfigured so as to redistribute the formulae monitored on each component. How this reconfiguration could happen at runtime was not fully described in [7] and implied i) instantaneous communication between monitors since the number of such messages was not bounded, and ii) reconfiguration messages increased the communication overhead of the monitoring algorithm. Leveraging the global clock in the system and using messages of the form $\text{verdict}(i, j, t, v)$, the intermediate results exchanged by monitors can be time stamped with the global clock. Our monitoring algorithm can be applied to systems where monitors can communicate at any frequency compared to the global clock while still ensuring the consistency of verdicts. That is, the monitoring algorithm of this paper supports asynchronous communication between monitors in the following sense: any message sent at time t by a monitor can be received at any later time instant $t' \geq t$ (where t and t' are given by the global clock), the order of messages can be inverted, but messages cannot be lost. Hence, the monitoring algorithm proposed in this paper is more amenable to a real implementation in a particular application scenario. Second, our monitoring algorithm leverages the offline analysis described in Section 4 to determine the cells of the network that need to respawn automatically and the communica-

tion paths prior to monitoring. Such analysis allows monitors to communicate verdicts with each other without requiring the transmission of the resolved formula itself. While we do not compare the performance of the monitoring algorithm proposed in [7] with the one in this paper, we carried out experiments showing improvements in the number and size of messages exchanged by monitors by orders of magnitude in most cases.

9 Conclusions and Future Work

In the context of distributed systems becoming increasingly ubiquitous, further studies are required to understand the variables involved and how these affect the numerous criteria which constitute good monitoring strategies. This would help architects choose the correct approach, particularly the monitor organisation and communication protocol, depending on the circumstances.

This study shows that while choreography can be advantageous in specific scenarios such as in the case of systems with many components and formulae which can be shallowly distributed, generally such an arrangement requires a significant number of messages and cannot fully exploit the potential of LTL simplification routines. We have noted that a substantial increase in the number of progressions required for choreography because of the monitors keeping track of versions of their local formulae over time while waiting for information from other components. This means that LTL might not be the best candidate when opting for a choreography. In contrast, non-progression-based monitoring algorithms where the monitors are not constantly modified, might lend themselves better to choreography.

We consider future work in two main directions: First, we would like to investigate how LTL equivalence rules can be used to make the choreography tree shallower. For example distributing $(a_1 \wedge a_2) \wedge ((a_3 \wedge b_1) \wedge b_2)$ might require two hops to reach a verdict while using associativity rules (obtaining $((a_1 \wedge a_2) \wedge a_3) \wedge (b_1 \wedge b_2)$), it can be easily reduced to one. Second, using other notations instead of LTL and/or different monitoring algorithms, particularly ones which are not progression-based, can potentially tip the balance more in favour of choreography approaches.

Acknowledgment. The work reported in this article has been done in the context of the COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology). The authors would like to thank Adrian Francalenza (U of Malta), Susanne Graf (Vérimag), and César Sanchez (IMDEA Madrid) for discussions the issue on simplifying LTL formulae. The authors are grateful to the DataMill team at the University of Waterloo for providing us with such a nice experimentation platform. The authors gratefully thank the anonymous reviewers for their comments and suggestions allowing to improve the quality of this paper.

References

1. C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.
2. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.
3. E. Bartocci. Sampling-based decentralized monitoring for networked embedded systems. In *3rd Int. Work. on Hybrid Autonomous Systems*, volume 124 of *EPTCS*, pages 85–99, 2013.
4. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Logic and Computation*, 20(3):651–674, 2010.

5. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 20(4):14, 2011.
6. A. K. Bauer and Y. Falcone. Decentralised LTL monitoring. In *18th Int. Symp. on Formal Methods*, volume 7436 of *LNCS*, pages 85–100. Springer, 2012.
7. C. Colombo and Y. Falcone. Organising LTL monitors over distributed systems with a global clock. In *Proceedings of the 5th International Conference Runtime Verification (RV 2014)*, Lecture Notes in Computer Science, pages 140–155. Springer, 2014.
8. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Intl. Conf. on Software Engineering (ICSE)*, pages 411–420. ACM, 1999.
9. K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In C. Palamidessi, editor, *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2000.
10. Y. Falcone, T. Cornebize, and J.-C. Fernandez. Efficient and generalized decentralized monitoring of regular languages. In E. Ábrahám and C. Palamidessi, editors, *FORTE 2014: 34th IFIP Int. Conference on Formal Techniques for Distributed Objects, Components and Systems*, volume 8461 of *LNCS*, pages 66–83. Springer, 2014.
11. Y. Falcone, J. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.
12. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In M. Broy, D. Peled, and G. 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. A. Francalanza, A. Gauci, and G. J. Pace. Distributed system contract monitoring. *J. Log. Algebr. Program.*, 82(5-7):186–215, 2013.
14. S. Graf, D. Peled, and S. Quinton. Monitoring distributed systems using knowledge. In R. Bruni and J. Dingel, editors, *Proc. of the Joint 13th IFIP WG 6.1 Int. Conference and 31st IFIP WG 6.1 Int. Conference*, volume 6722 of *LNCS*, pages 183–197. Springer, 2011.
15. M. Gunzert and A. Nägele. Component-based development and verification of safety critical software for a brake-by-wire system with synchronous software components. In *Intl. Symp. on SE for Parallel and Distributed Systems (PDSE)*, pages 134–. IEEE, 1999.
16. D. Harris. A taxonomy of parallel prefix networks. In *Signals, Systems and Computers*, volume 2, pages 2213–2217, 2003.
17. K. Havelund and A. Goldberg. Verify your runs. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, volume 4171 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2005.
18. K. Havelund and G. Rosu. Monitoring programs using rewriting. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 135–143, 2001.
19. R. Larrieu and N. Shankar. A framework for high-assurance quasi-synchronous systems. In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2014, Lausanne, Switzerland, October 19-21, 2014*, pages 72–83. IEEE, 2014.
20. M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
21. W. C. Lynch. Computer systems: Reliable full-duplex file transmission over half-duplex telephone line. *Commun. ACM*, 11(6):407–410, 1968.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.
23. R. Mayr and L. Clemente. Advanced automata minimization. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 63–74. ACM, 2013.
24. S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53:58–64, Feb. 2010.
25. A. Pnueli. The temporal logic of programs. In *SFCS'77: Proc. of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.
26. A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.
27. A. Pnueli and A. Zaks. On the merits of temporal testers. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 172–195. Springer, 2008.

28. T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Syst.*, 39:205–235, 2008.
29. G. Rosu and K. Havelund. Rewriting-based techniques for runtime verification. *Autom. Softw. Eng.*, 12(2):151–197, 2005.
30. K. Sen, G. Rosu, and G. Agha. Generating optimal linear temporal logic monitors by coinduction. In V. A. Saraswat, editor, *Advances in Computing Science - ASIAN 2003 Programming Languages and Distributed Computation, 8th Asian Computing Science Conference, Mumbai, India, December 10-14, 2003, Proceedings*, volume 2896 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2003.
31. K. Sen, A. Vardhan, G. Agha, and G. Rosu. Decentralized runtime analysis of multithreaded applications. In *20th Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006.
32. O. Sokolsky, K. Havelund, and I. Lee. Introduction to the special section on runtime verification. *STTT*, 14(3):243–247, 2012.
33. F. Somenzi and R. Bloem. Efficient büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.

A Proofs

In this appendix, we provide the proofs for the propositions, lemmata, and the theorem of this paper.

Proposition 1 (Maximum level of nested distributions) $\forall \varphi \in \mathrm{LTL} \cdot \mathrm{dpth}_D(\mathrm{net}(\varphi)) \leq \mathrm{dpth}(\varphi)$.

Proof The proof follows by induction on the structure of φ:

Base case: $\varphi \in \{\mathrm{true}, \mathrm{false}\} \cup AP$
$\mathrm{net}(\varphi) = N_{E\,\mathrm{chc}(\varphi),1} \mapsto \varphi$ *(def. of* net*)*
$\mathrm{dpth}_D(\mathrm{net}(\varphi)) = 1$ *(def. of* dpth_D*)*
$\mathrm{dpth}_D(\mathrm{net}(\varphi)) \leq \mathrm{dpth}(\varphi)$ *(def. of* dpth *and basic arithmetic)*
Inductive case: $\varphi \in \{\sim \psi, \psi \odot \psi'\}$
Case: $\varphi = \sim \psi$
$\mathrm{dpth}_D(\mathrm{net}(\psi)) \leq \mathrm{dpth}(\psi)$ *(inductive hypothesis)*
$\mathrm{dpth}_D(\mathrm{net}(\sim \psi)) \leq \mathrm{dpth}(\psi) + 1$ *(def. of* dpth_D *and basic arithmetic)*
$\mathrm{dpth}_D(\mathrm{net}(\sim \psi)) \leq \mathrm{dpth}(\sim \psi)$ *(def. of* dpth*)*
Case: $\varphi = \psi \odot \psi'$
Let $N, \varphi' = \mathrm{net}(\psi), \lceil N \rceil$ and $N', \varphi'' = \mathrm{net}(N, \psi'), \lceil N' \rceil$
Subcase: $\mathrm{net}(\psi \odot \psi') = N'_{\mathrm{chc}(\varphi),*} \mapsto \varphi' \odot \varphi''$
$\mathrm{dpth}_D(\mathrm{net}(\psi \odot \psi'))$
$= \max(\mathrm{dpth}_D(N), \mathrm{dpth}_D(N'))$ *(def. of* dpth_D*)*
$\leq 1 + \max(\mathrm{dpth}(\psi), \mathrm{dpth}(\psi'))$ *(inductive hypothesis and basic arithmetic)*
$\leq \mathrm{dpth}(\psi \odot \psi')$ *(def. of* dpth*)*
Subcase: $\mathrm{net}(\psi \odot \psi') = N'_{\mathrm{chc}(\varphi),*} \mapsto \varphi' \odot \langle i, j \rangle$
$\mathrm{dpth}_D(\mathrm{net}(\psi \odot \psi'))$
$= \max(\mathrm{dpth}_D(N), 1 + \mathrm{dpth}_D(N'))$ *(def. of* dpth_D*)*
$\leq 1 + \max(\mathrm{dpth}(\psi), \mathrm{dpth}(\psi'))$ *(inductive hypothesis and basic arithmetic)*
$\leq \mathrm{dpth}(\psi \odot \psi')$ *(def. of* dpth*)*
The other cases are similar.

□

Proposition 2 $\forall \varphi \in \mathrm{LTL} \cdot \mathrm{msg}^\circ(\mathrm{net}(\varphi)) = \varphi$.

Proof The proof follows by induction on the structure of the LTL formula.

Base case: $\varphi \in \{\mathrm{true}, \mathrm{false}\} \cup AP$
Case: $\varphi = \mathrm{true}$
$\mathrm{net}(\varphi) = N_{E\,\mathrm{chc}(\varphi),1} \mapsto \varphi$ *(def. of* net*)*
$\mathrm{msg}^\circ(\mathrm{net}(\varphi)) = \mathrm{true}$ *(def. of* msg°*)*
The other cases are similar
Inductive case: $\varphi \in \{\sim \psi, \psi \odot \psi'\}$ [17]

[17] $\mapsto \varphi$ is an abbreviation for $\mapsto \varphi'$ where $(N', \varphi') = \mathrm{distr}(N, \varphi)$

Case: $\varphi = \sim\psi$
$\mathrm{net}(\sim\psi) = \mathrm{net}(\psi)_{\mathrm{chc}(\varphi),*} \mapsto \sim\psi$ *(def. of* net*)*
$\mathrm{msg}^\circ(\mathrm{net}(\sim\psi)) = \sim\mathrm{msg}^\circ(\mathrm{net}(\psi))$ *(def. of* msg° *and structure of* $\mathrm{net}(\sim\psi)$*)*
$\mathrm{msg}^\circ(\mathrm{net}(\sim\psi)) = \sim\psi$ *(by inductive hypothesis)*
Case: $\varphi = \psi \odot \psi'$
Let $N, \varphi' = \mathrm{net}(\psi), \lceil N \rceil$ and $N', \varphi'' = \mathrm{net}(N, \psi'), \lceil N' \rceil$
Subcase: $\mathrm{net}(\psi \odot \psi') = {N'}_{\mathrm{chc}(\varphi),*} \mapsto \varphi' \odot \varphi''$
$\mathrm{msg}^\circ(\mathrm{net}(\psi \odot \psi'))$
$= \mathrm{msg}^\circ(\mathrm{net}(\psi)) \odot \mathrm{msg}^\circ(\mathrm{net}(\psi'))$ *(def. of* msg° *and structure of* $\mathrm{net}(\psi \odot \psi')$*)*
$= \psi \odot \psi'$ *(by inductive hypothesis)*
Subcase: $\mathrm{net}(\psi \odot \psi') = {N'}_{\mathrm{chc}(\varphi),*} \mapsto \varphi' \odot \langle i, j \rangle$
$\mathrm{msg}^\circ(\mathrm{net}(\psi \odot \psi'))$
$= \mathrm{msg}^\circ(\mathrm{net}(\psi)) \odot \mathrm{msg}^\circ(\mathrm{net}(\psi \odot \psi')_{i,j})$ *(def. of* msg° *and structure of* $\mathrm{net}(\psi \odot \psi')$*)*
$= \mathrm{msg}^\circ(\mathrm{net}(\psi)) \odot \mathrm{msg}^\circ(\mathrm{net}(\psi'))$ *(def. of* net*)*
$= \psi \odot \psi'$ *(by inductive hypothesis)*
The other cases are similar.

□

Lemma 1 (Correctness for one step under fully instantaneous communication)

$$\forall \varphi \in \mathrm{LTL}, \forall \sigma \in \Sigma \cdot \mathrm{prog}(\varphi, \sigma) \simeq \mathrm{msg}^\circ(\mathrm{alg}(\mathrm{net}(\varphi), \sigma))$$

The verdict reached by choreographed monitoring under fully instantaneous communication is the same as the one reached under standard progression with global view of events.

Proof Since we assume fully instantaneous communication, in this proof we will ignore the algorithm's communication mechanism and focus on part 5 of the algorithm, i.e., the respawning and distributed progression mechanism.

The proof follows by induction on the distribution structure, i.e., the linkages within the network memory, of $M = \mathrm{alg}(\mathrm{net}(\varphi), \sigma)$ with corresponding initial network N:

Base case: Network of M has no linkages
We note that when M has no distribution, $\mathrm{compute\_respawn}(N) = \emptyset$. Furthermore, prog_t behaves exactly as prog in the non-distribution cases. Therefore, the base case follows by the inductive hypothesis and these observations.

Inductive case: Formula of M has distribution linkages
Due to the inductive hypothesis, which establishes a correspondence between the existing distribution placeholders in the formulae and the existing formulae being monitored in the network, we only need to prove that these maintain their correspondence and that new ones also correspond.

Case: Existing linkages of the form $\langle i, j, t \rangle$ correspond to $M^t_{i,j}$
We note that the t in $\langle i, j, t \rangle$ and $M^t_{i,j}$ match and are not altered until the messaging system transmits the contents of $M^t_{i,j}$, meaning that the correspondence is maintained.

Case: New linkages of the form $\langle i, j \rangle$ correspond to new formulae $M^t_{i,j}$
By case-by-case analysis of prog_t, we note that there are only two means of introducing new $\langle i, j \rangle$ in the formula: either within the **X** operator or within the **U** operator. Correspondingly, we note that by analysis of compute_respawn, there are only two means of introducing new $M^t_{i,j}$ in the formula: either within the **X** operator or within the **U** operator.

Case: Deleted linkages of the form $\langle i, j \rangle$, $\langle i, j, t \rangle$ correspond to discarded formulae $M^t_{i,j}$
Following progression and simplification, a number of distribution propositions may be discarded. Part 7 of the algorithm is responsible for sending corresponding *kill* messages which are correspondingly handled by part 3. Conversely, if a cell $M^t_{i,j}$ reaches a verdict, part 8 of the algorithm is responsible for sending a verdict message to the corresponding distribution propositions and discarding the cell. Such a message is in turn handled by part 2 of the algorithm which replaces the placeholder with the verdict. Once more this preserves the correspondence of placeholders and cells.

□

Lemma 2 (Correctness under fully instantaneous communication) Lifting Lemma 1 to trace of events, a trace of events still yields correct result when using the choreographed monitoring approach:

$$\forall \varphi \in \mathrm{LTL}, \forall u \in \Sigma^* \cdot \mathrm{prog}(\varphi, u) \simeq \mathrm{msg}^\circ(\mathrm{alg}(\mathrm{net}(\varphi), u))$$

Proof The proof follows by induction on the trace structure.

Base case: An empty trace — $\text{prog}(\varphi,\varepsilon) \simeq \text{msg}^\circ(\text{alg}(\text{net}(\varphi),\varepsilon)$
$\text{prog}(\varphi,\varepsilon) \simeq \varphi$ *(no applications of* prog*)*
$\text{prog}(\varphi,\varepsilon) \simeq \text{net}(\varphi)$ *(Proposition 2)*
$\text{prog}(\varphi,\varepsilon) \simeq \text{alg}(\text{net}(\varphi),\varepsilon)$ *(no applications of* alg*)*
Inductive case: An additional trace element — $\text{prog}(\varphi,\text{u}\cdot\sigma) \simeq \text{msg}^\circ(\text{alg}(\text{net}(\varphi),\text{u}\cdot\sigma)$ By the definitions of prog and alg, the statement can be reformulated to:

$$\text{prog}(\text{prog}(\varphi,\text{u}),\sigma) \simeq \text{msg}^\circ(\text{alg}(\text{alg}(\text{net}(\varphi),\text{u}),\sigma)$$

This follows by Lemma 1.

Proposition 3 (More defined) $\forall\varphi \in \text{LTL}_D, \forall\varphi' \in \text{LTL}\cdot\varphi \succeq \varphi' \implies \varphi = \varphi'$.

Proof

$\forall \text{A} \in \mathscr{A}\cdot \text{A}(\varphi') = \varphi'$ *(*A *only modifies distribution constructs)*
$\text{A}(\varphi') = \varphi$ *(def. of* $\succeq$*)*
$\varphi' = \varphi$ *(applying the empty* A*)*

□

Lemma 3 For all possible network memories, full messaging yields more defined formulae than verdict-only, time-stepped messaging: $\forall M \in \mathscr{M}\cdot \text{msg}^\circ(M) \succeq \text{msg}^{\text{v}}(M)$.

Proof By choosing the values of corresponding $\text{msg}^\circ(M_{i,j})$ for undefined assignments of $\text{msg}^{\text{v}}(M)$, we would have $\text{A}(\text{msg}^{\text{v}}(M)) = \text{msg}^\circ(M)$, which by definition of $\succeq$ lead us to conclude $\text{msg}^\circ(M) \succeq \text{msg}^{\text{v}}(M)$ as required.

□

Theorem 1 (Correctness of verdict-only time-stepped messaging) If a verdict is reached when using verdict-only messaging, then the verdict is correct: $\forall\varphi \in \text{LTL}, \text{u} \in \Sigma^*\cdot \text{msg}^{\text{v}}(\text{alg}(\text{net}(\varphi),\text{u})) \in \{\top,\bot\} \implies \text{msg}^{\text{v}}(\text{alg}(\text{net}(\varphi),\text{u})) \simeq \text{prog}(\varphi,\text{u})$

Proof The proof follows directly from Lemma 1, Proposition 3, and Lemma 3.

□

Corollary 1 (Correspondence of verdicts) If decentralised semantics assign a verdict to a trace-formula pair, then the LTL_3 semantics assigns the same verdict: if $\text{verdict}_{chor}(u,\varphi) \in \{\top,\bot\}$ then $\text{verdict}_{chor}(u,\varphi) = u \models_3 \varphi$.

Proof The proof follows directly from Theorem 1.

□

B Plots for the Visualisation of the Results of the Experiments

Recall that the experiments described in Section 7 aim at benchmarking and comparing the performance of the three decentralised monitoring algorithms (orchestration, migration, and choreography) along four metrics: the delay induced by decentralised monitoring, the number and size of messages exchanged by monitors, and number of progressions that monitors need to carry out to find a verdict (see Section 7 for the description of the metrics and objectives of the experiments).

In this section, we provide plots for the complementary visualisation of the results of the first and second experiments described in Section 7. For each metric, in each plot, we display information on the value of the metric according to formula size for the first experiment and according to specification pattern for the second experiment. For each metric, we provide three plots: (what is referred to as) a custom plot, a box plot, and a scatter plot.

B.1 Description of the Plots

Custom plots report the average value (circle) and median (cross mark) of the metric. Moreover, the 99% confidence intervals of the means are depicted as crossbars centred around the mean value of the metric. In addition, to facilitate the visualisation and comparison of the algorithms based on the obtained average values, the symmetry or asymmetry of the metric value distribution can be hinted by inspecting the difference between the average and median values.

Box plots intuitively focus on "the main cases" and their dispersion. They also confirm the a(symmetry) of the distributions of observations hinted with custom plots. More precisely, recall that in a box plot the upper and lower "hinges" mark the first and third quartiles respectively. The line inside the box marks the

second quartile (i.e., the median). Moreover, box plots are Tukey ones where the upper whisker extends up to the last values inside the upper inner fence, i.e., the highest value that lies within 1.5 times the inter-quartile range to the third quartile; and the lower whisker extends down to the last value inside the lower inner fence, i.e., the lowest value within 1.5 times the inter-quartile range to the first quartile. Outliers (i.e., values lower than the lower whisker and greater than the upper whisker) are not displayed.

Scatter plots display the values obtained for the metrics for all samples. They provide a global view of the obtained values and can be useful to estimate the number of outliers and their “distance to the middle values". Horizontal jittering is applied to facilitate the estimation of the density of points. Figures 6, 7, 8, and 9 contain the plots for the visualisation of the results of Experiment 1, for unbiased and biased formula generation. Figures 10, 11, 12, and 13 contain the plots for the visualisation of the results of Experiment 2.

B.2 Using the Plots to Analyse Data

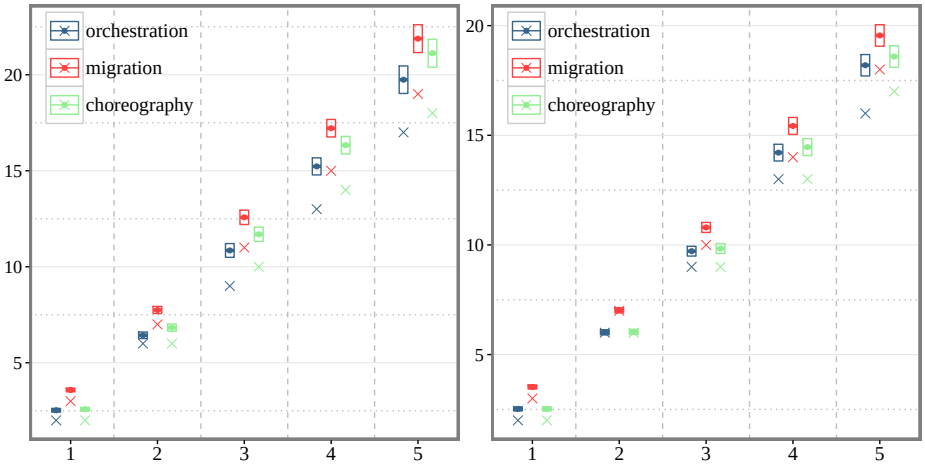
We now describe how to use the plots to draw conclusions from the experimental data. We refrain from examining each metric for each formula size and specification pattern but rather recall general methods to analyse the plots and mention some of the interesting cases. The plots confirm and refine the trends mentioned in Sections 7.4, 7.5, and 7.6.

The relative positions of the mean and median give hints on data skewness. A positive (resp. negative) value for the difference between the mean and median can hint a positive (resp. negative) skew.

Moreover, box plots indicate centrality (with the median), spread (the size of the box), symmetry or skewness of data, and tail length and shape of the distribution (with the relative lengths of the whiskers and box). Positive (resp. negative) skewness is characterised by a median in the lower (resp. upper) part of the box and an upper (resp. lower) whisker that is longer than the lower (resp. upper) whisker. However, we note that the above rule does not cover all the cases and other situations may arise, for instance for the number of messages obtained with the choreography algorithm for random formulae of size 5 where the mean and median are close to each other, the median is in the upper part of the box, and the lower whisker is inexistant. Hence, both the median position and the lengths of whiskers have to be examined to draw conclusions on the distribution of values. For instance:

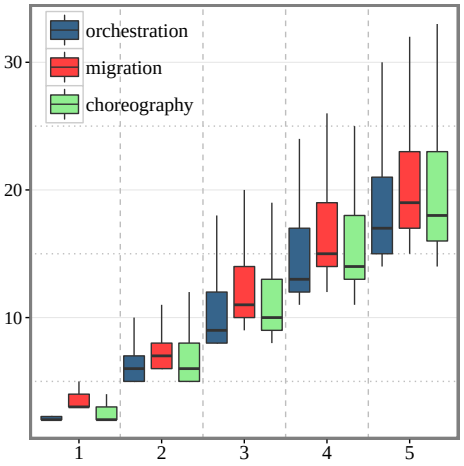
- the number of messages obtained for response formulae (cf. Fig. 11b) has no skew for the three algorithms,
- the trace length and number of progressions obtained for (unbiased and biased) random formulae (cf. Fig. 6 and 9) has a positive skew,
- none of the obtained distributions for the metrics has a negative skew.

Scatter plots help visualising the positions and density of outliers compared to the “main values" for each distribution. For instance, for the trace lengths obtained when monitoring randomly-generated formulae, the number and positions of outliers seem to be the same for the three algorithms. For the number of messages obtained when monitoring formulae with unbiased random formula generation, choreography has more and further outliers than orchestration, while the situation is reversed when formulae are obtained with biased formula generation. For Experiment 2 (related to specification patterns), examining the scatter plots with far outliers, we can notice that, for some precedence chain formulae, choreography was defeated in terms of number of progressions while it performed similarly or better for trace length and size of messages. Finally, a question that arose was whether the outliers were for the same formulae across all algorithms. After observing the scatter plots and the data set obtained from the experiments, we confirm that this is generally the case, meaning that outliers reflect the differences between formulae rather than algorithms.

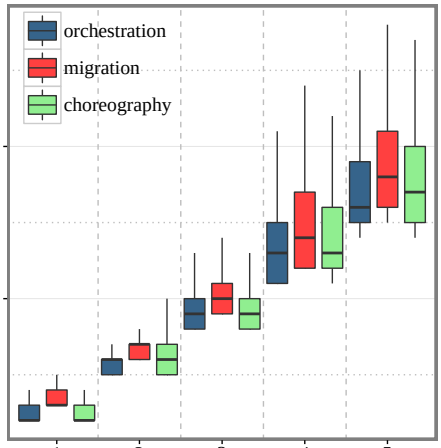


(a) Custom plot - unbiased formula generation

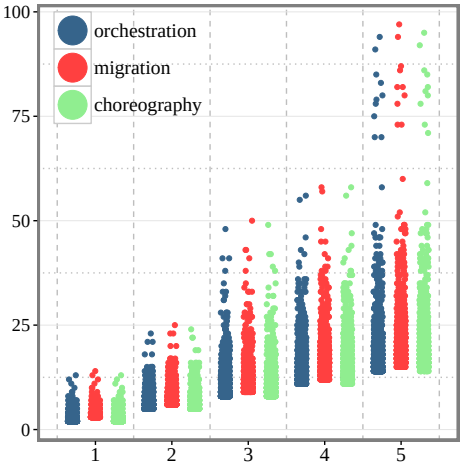
(b) Custom plot - biased formula generation



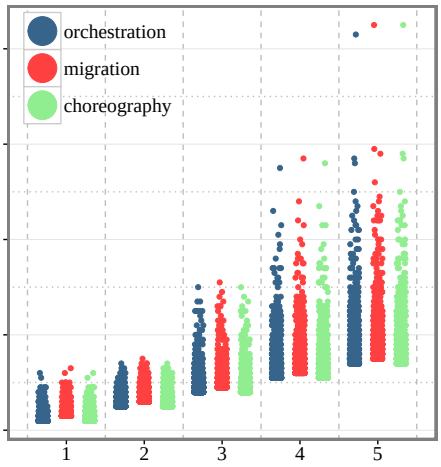
(c) Boxplot - unbiased formula generation



(d) Box plot - biased formula generation

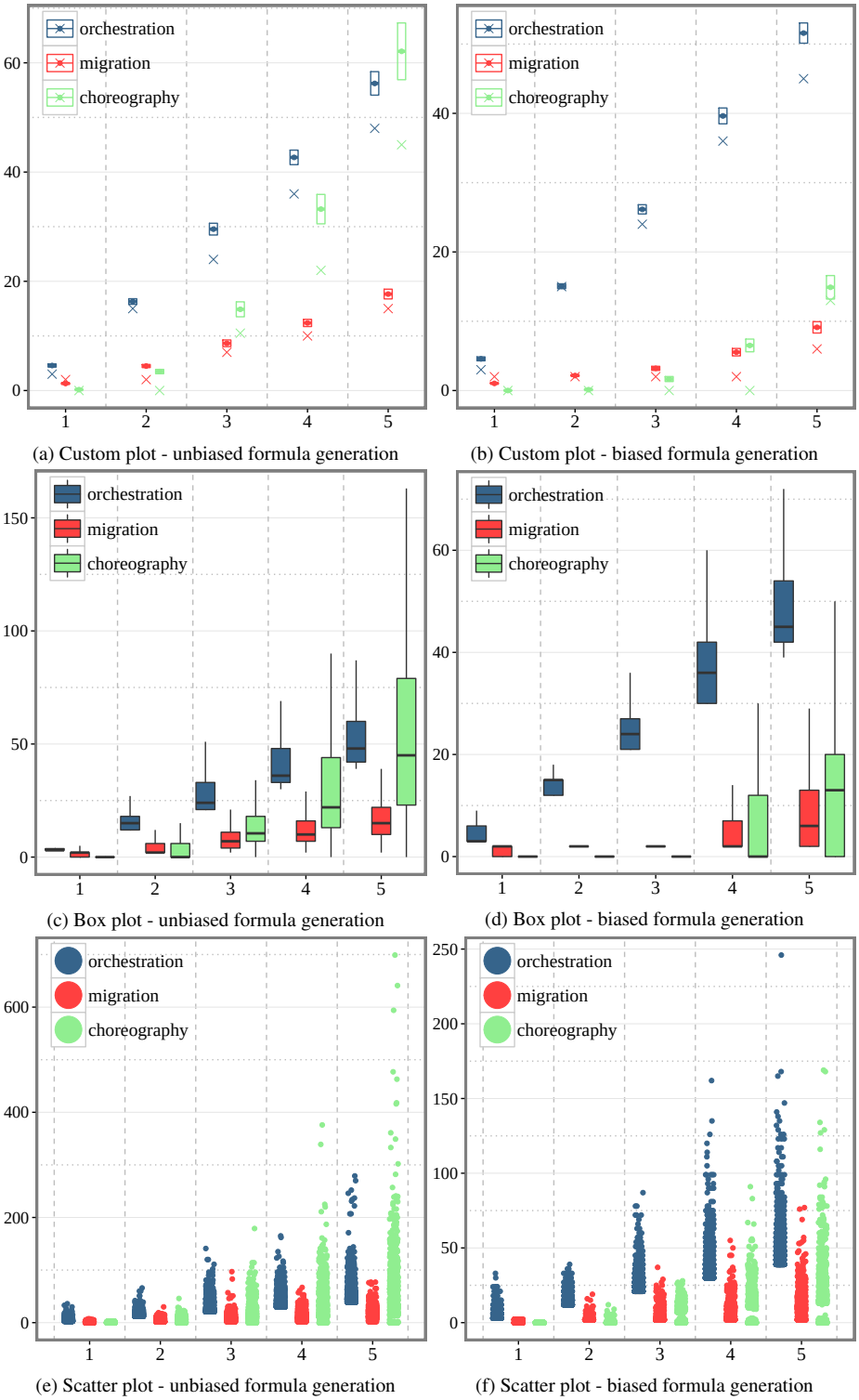


(e) Scatter plot - unbiased formula generation



(f) Scatter plot - biased formula generation

Fig. 6: Visualisation of the results (reported in Table 2) obtained for the trace length ($|\mathrm{tr}|$ in ordinate) with the experiment varying the size of formulae ($|\varphi|$ in abscissa).



(a) Custom plot - unbiased formula generation

(b) Custom plot - biased formula generation

(c) Box plot - unbiased formula generation

(d) Box plot - biased formula generation

(e) Scatter plot - unbiased formula generation

(f) Scatter plot - biased formula generation

Fig. 7: Visualisation of the results (reported in Table 2) obtained for the number of messages (#msg in ordinate) with the experiment varying the size of formulae ($|\varphi|$ in abscissa).

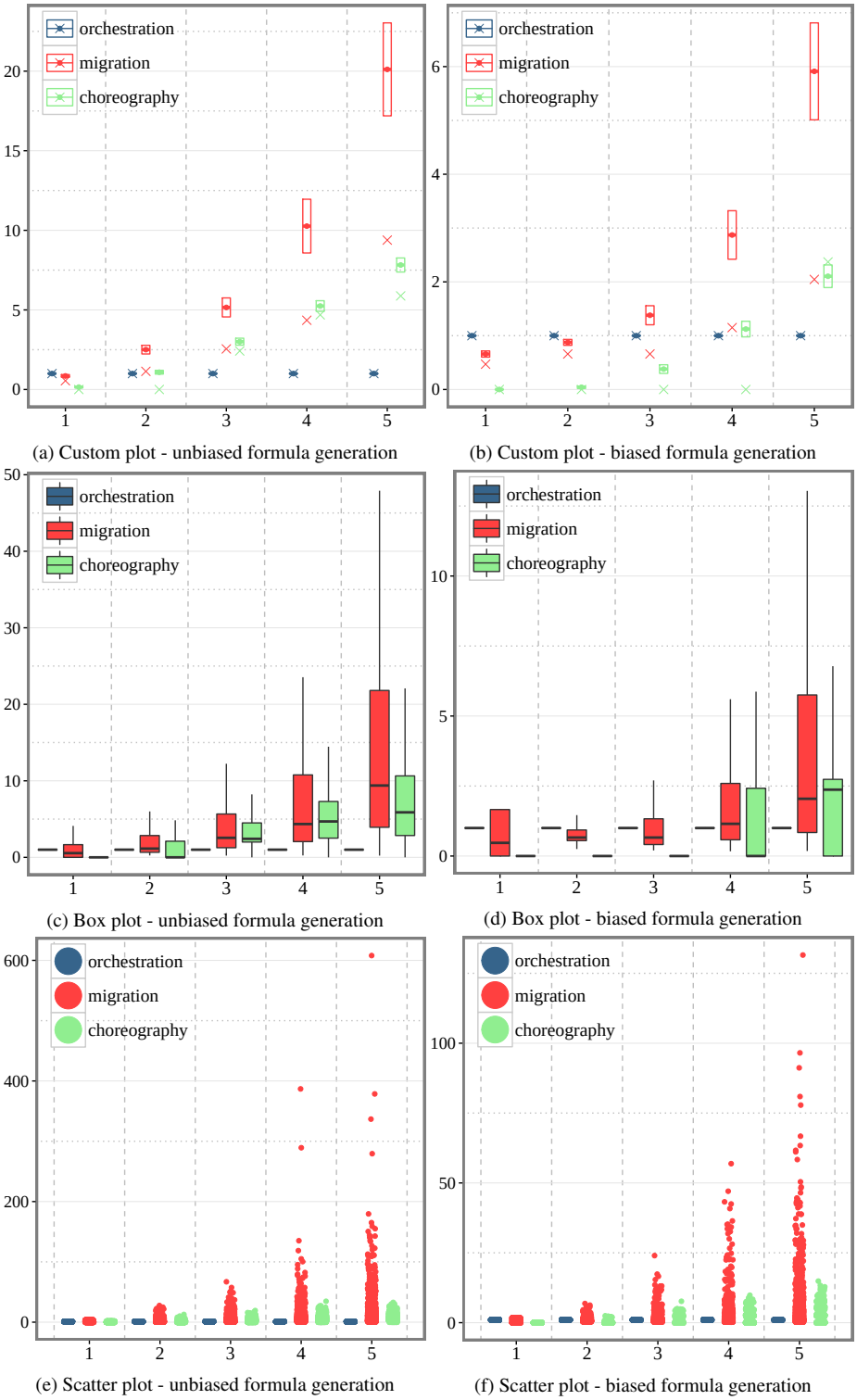
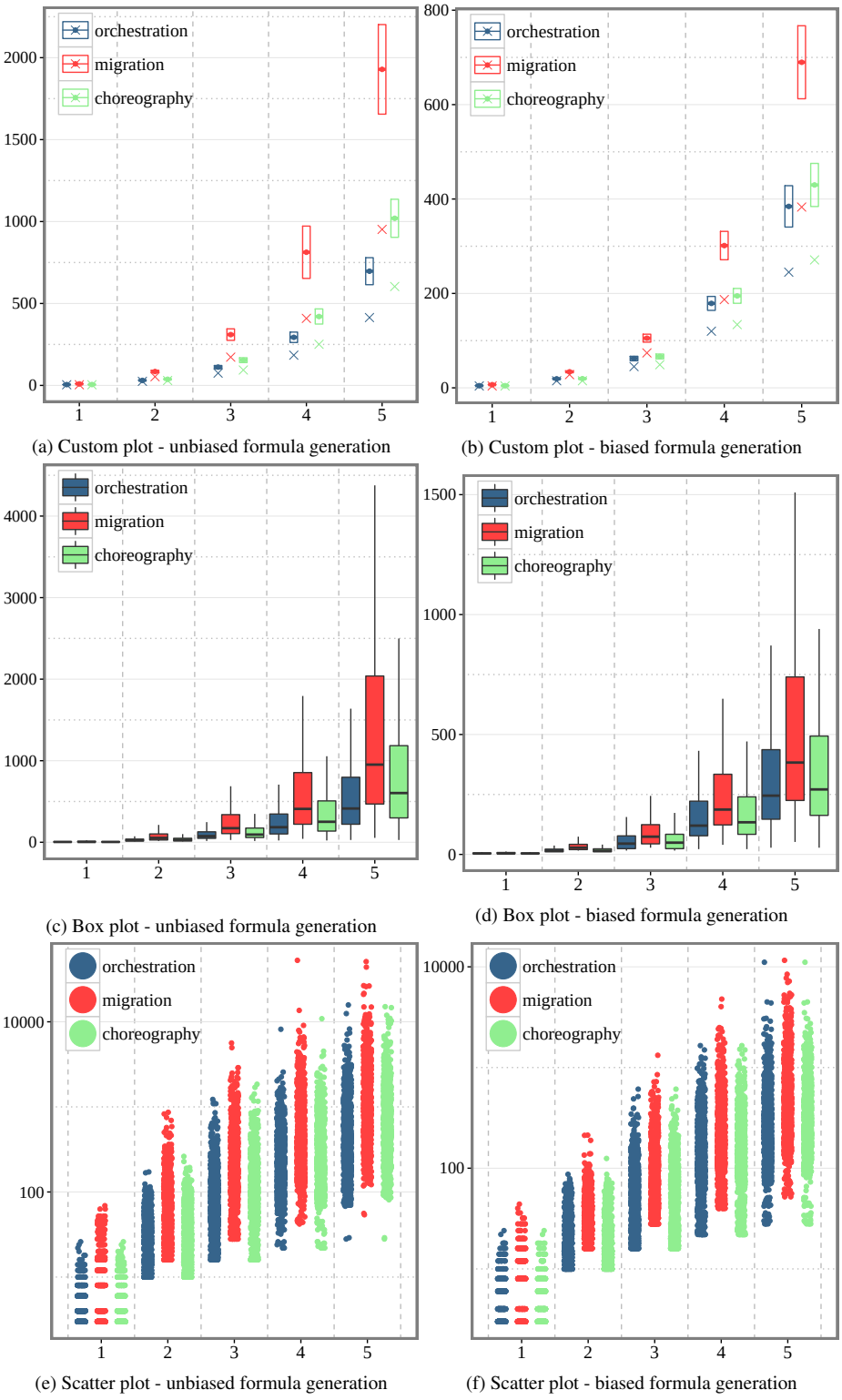


Fig. 8: Visualisation of the results (reported in Table 2) obtained for the size of messages (|msg| in ordinate) with the experiment varying the size of formulae ($|\varphi|$ in abscissa).



(a) Custom plot - unbiased formula generation

(b) Custom plot - biased formula generation

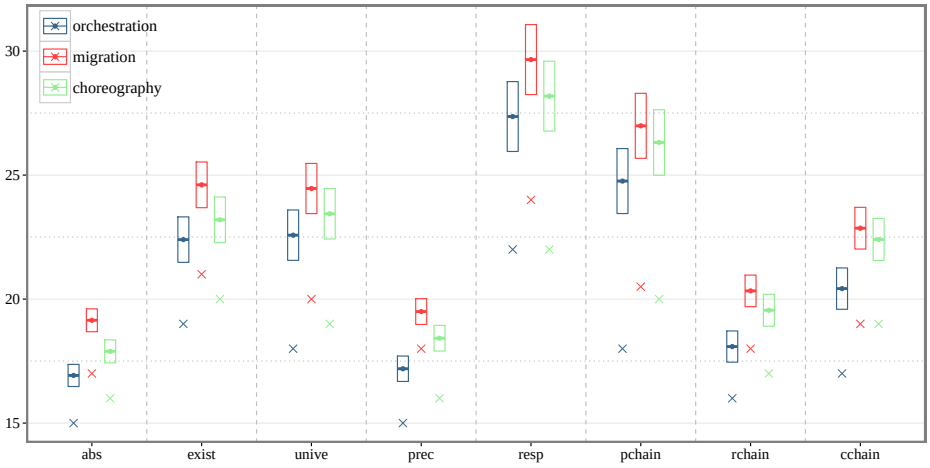
(c) Box plot - unbiased formula generation

(d) Box plot - biased formula generation

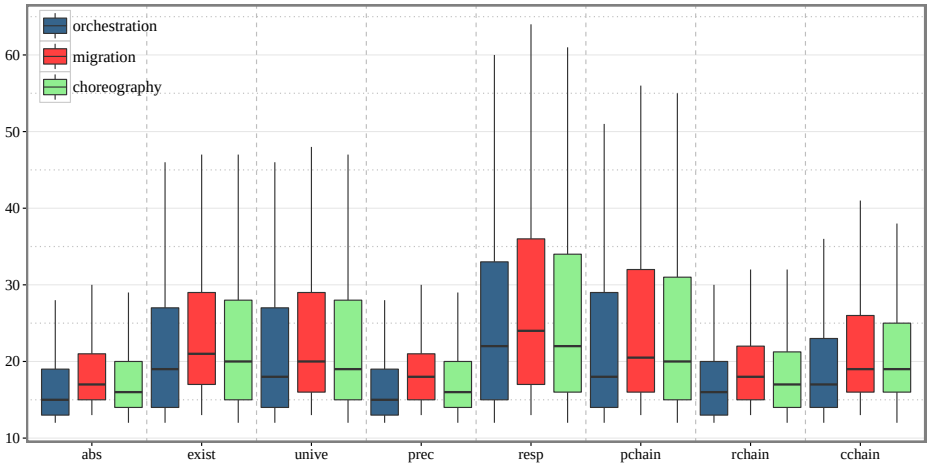
(e) Scatter plot - unbiased formula generation

(f) Scatter plot - biased formula generation

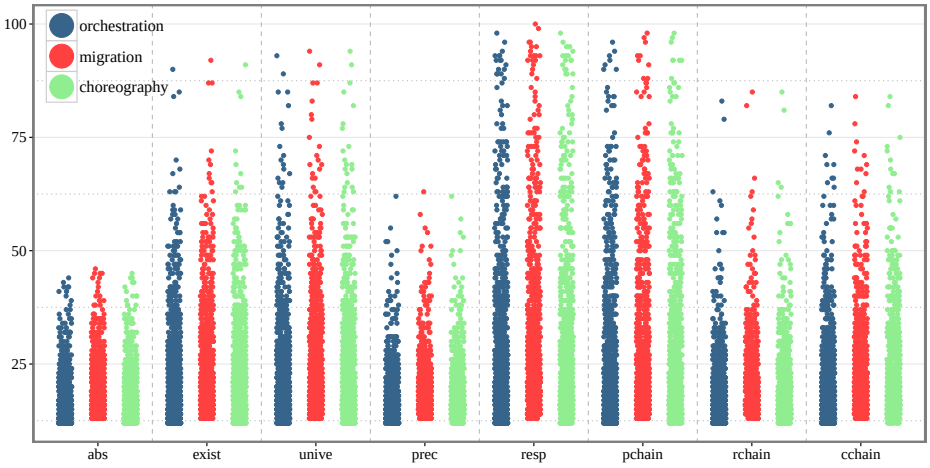
Fig. 9: Visualisation of the results (reported in Table 2) obtained for the number of progressions (#prog in ordinate with the experiment varying the size of formulae ($|\varphi|$ in abscissa). A base-10 logarithmic scale is used for the ordinate axis of scatter plots.



(a) Custom plot

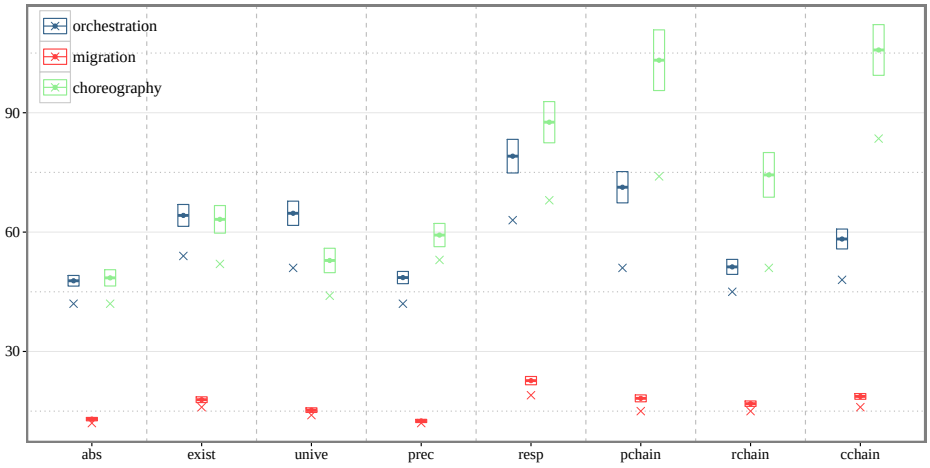


(b) Box plot

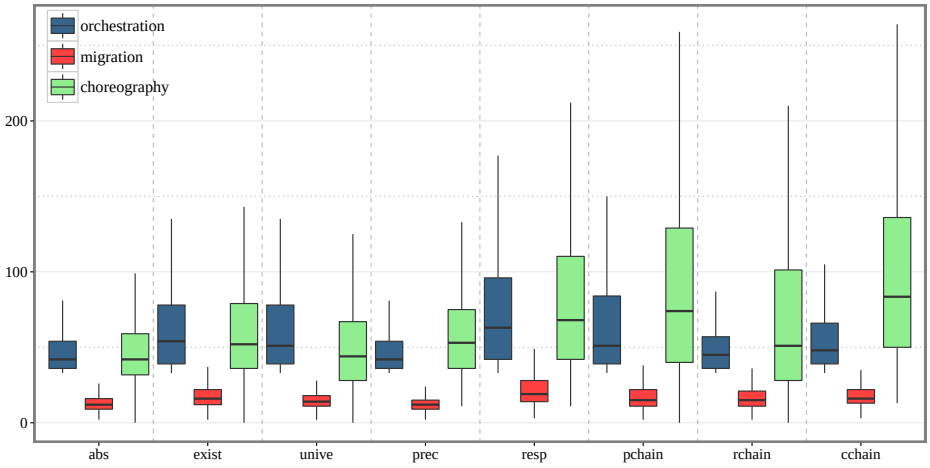


(c) Scatter plot

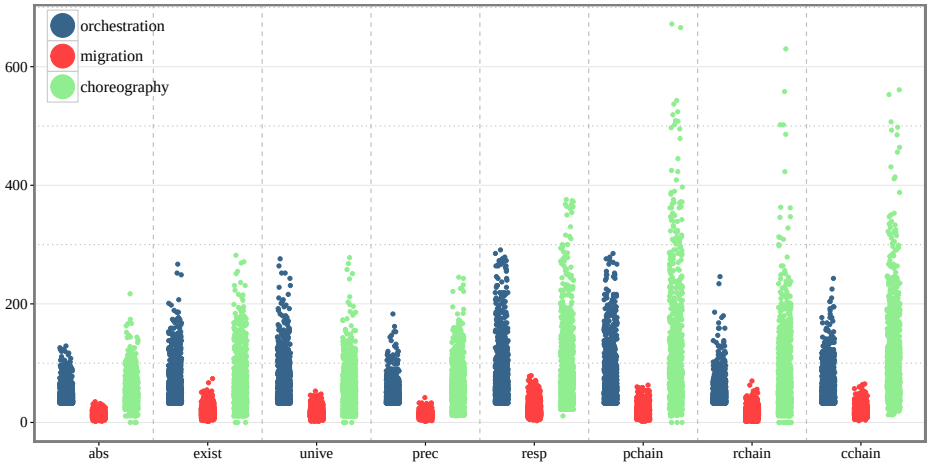
Fig. 10: Visualisation of the results (reported in Table 3) obtained for the trace length (|tr| in ordinate) with the experiment varying the pattern of formulae (in abscissa).



(a) Custom plot

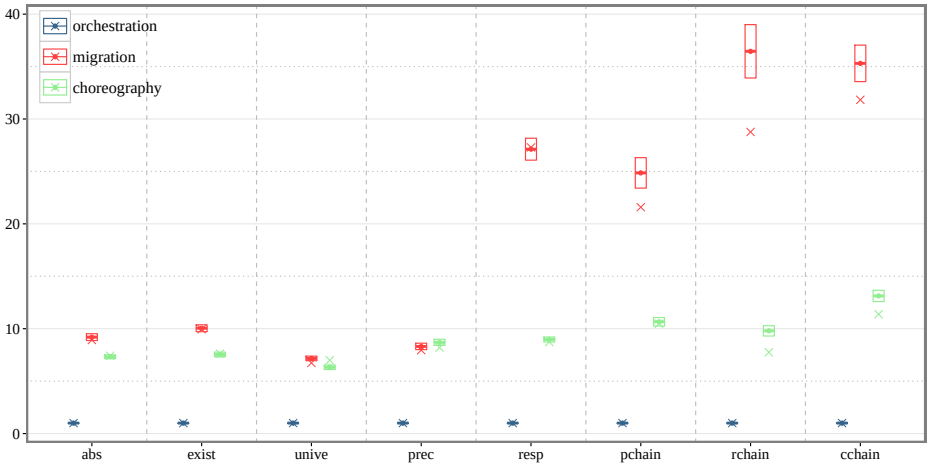


(b) Box plot

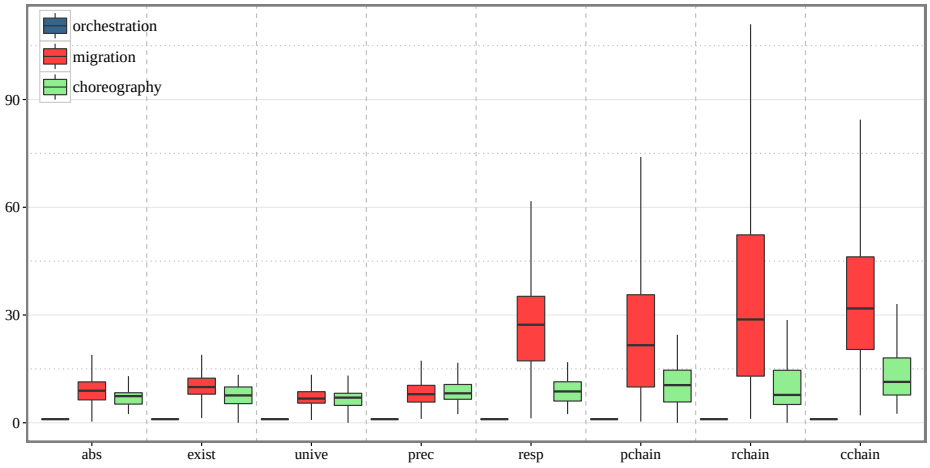


(c) Scatter plot

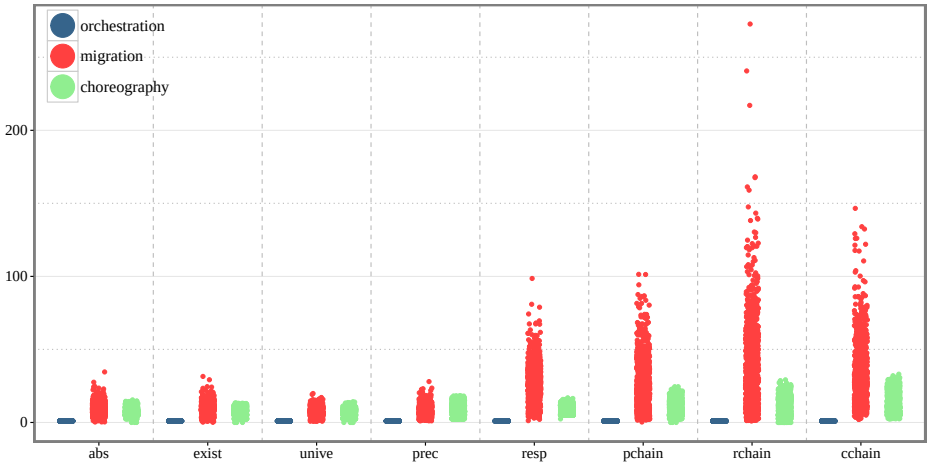
Fig. 11: Visualisation of the results (reported in Table 3) obtained for the number of messages (#msg in ordinate) with the experiment varying the pattern of formulae (in abscissa).



(a) Custom plot

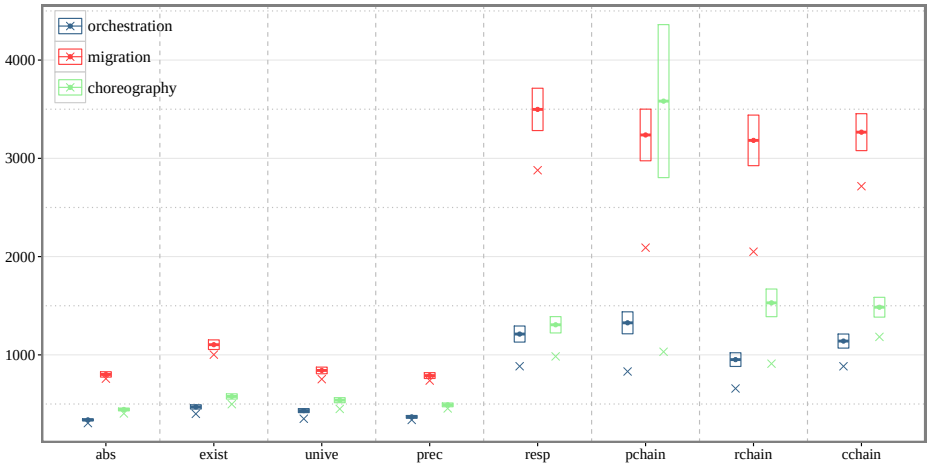


(b) Box plot

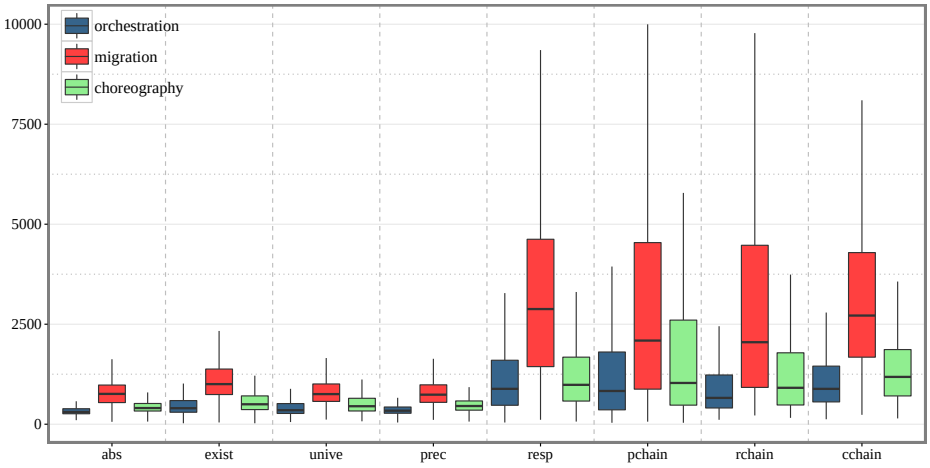


(c) Scatter plot

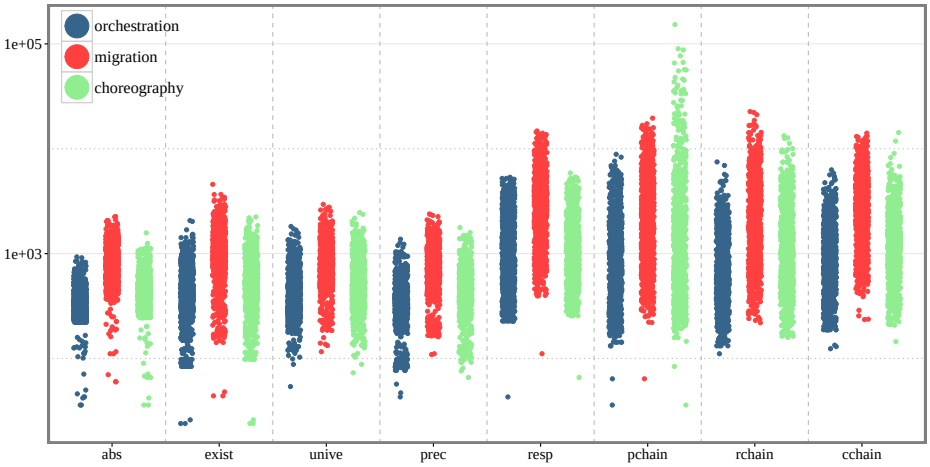
Fig. 12: Visualisation of the results (reported in Table 3) obtained for the size of messages (|msg| in ordinate) with the experiment varying the pattern of formulae (in abscissa).



(a) Custom plot



(b) Box plot



(c) Scatter plot (with base-10 logarithmic scale for the ordinate axis)

Fig. 13: Visualisation of the results (reported in Table 3) obtained for the number of progressions (#prog in ordinate) with the experiment varying the pattern of formulae (in abscissa).