On the Monitoring of Decentralized Specifications: Semantics, Properties, Analysis, and Simulation

Antoine El-Hokayem, Yliès Falcone

▶ **To cite this version:**

Antoine El-Hokayem, Yliès Falcone. On the Monitoring of Decentralized Specifications: Semantics, Properties, Analysis, and Simulation. ACM Transactions on Software Engineering and Methodology, 2019, pp.1-57. hal-02283429v2

HAL Id: hal-02283429
https://hal.science/hal-02283429v2

Submitted on 1 Oct 2019

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

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

On the Monitoring of Decentralized Specifications: Semantics, Properties, Analysis, and Simulation

ANTOINE EL-HOKAYEM, Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, France
YLIÈS FALCONE, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, France

We introduce two complementary approaches to monitor decentralized systems. The first approach relies on systems with a centralized specification, i.e., when the specification is written for the behavior of the entire system. To do so, our approach introduces a data structure that (i) keeps track of the execution of an automaton (ii) has predictable parameters and size and (iii) guarantees strong eventual consistency. The second approach defines decentralized specifications wherein multiple specifications are provided for separate parts of the system. We study two properties of decentralized specifications pertaining to monitorability and compatibility between specification and architecture. We also present a general algorithm for monitoring decentralized specifications. We map three existing algorithms to our approaches and provide a framework for analyzing their behavior. Furthermore, we present `THEMIS`, a framework for designing such decentralized algorithms and simulating their behavior. We demonstrate the usage of `THEMIS` to compare multiple algorithms and validate the trends predicted by the analysis in two scenarios: a synthetic benchmark and the Chiron user interface.

CCS Concepts: • **Software and its engineering** → **Software verification**; • **Theory of computation** → *Formal languages and automata theory*; *Data structures design and analysis*; • **Computing methodologies** → *Simulation tools*;

Additional Key Words and Phrases: Runtime Verification, Monitoring, Simulation, Decentralized Monitoring, Automata, Eventual Consistency

1 INTRODUCTION

Runtime Verification (RV) [6, 30, 31, 38] is a lightweight formal method which consists in verifying that a run of a system is correct with respect to a specification. The specification formalizes the behavior of the system typically in logics (such as variants of Linear-Time Temporal Logic, LTL) or finite-state machines. Typically the system is considered as a black box that feeds events to a monitor. An event usually consists of a set of atomic propositions that describe some abstract operations or states in the system. The sequence of events transmitted to the monitor is referred to as the trace. Based on the received events, the monitor emits verdicts in a truth domain that indicate whether or not the run complies with the specification. A typical truth domain is the set $\{\top, \bot, ?\}$ where verdicts $\top$ and $\bot$ indicate respectively that a program complies or violates the specification, and verdict ? indicates that no final verdict could be reached yet. Truth domains can also include additional verdicts such as currently true and currently false, to indicate a finer grained truth value. RV techniques have been used for instance in the context of decentralized automotive [16] and medical [39] systems. In both cases, RV is used to verify correct communication

Authors' addresses: Antoine El-Hokayem, Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, CS 40700, 38000 Grenoble, France, antoine.el-hokayem@univ-grenoble-alpes.fr; Yliès Falcone, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, CS 40700, 38000 Grenoble, France, ylies.falcone@univ-grenoble-alpes.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1049-331X/2019/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

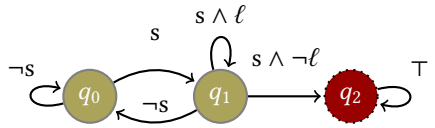


Fig. 1. LTL_3 monitor for the light switch and bulb specification. The verdicts associated with the states are ⊥: dotted/red, and ?: solid/yellow.

patterns between the various components and their adherence to the architecture and their formal specifications. While RV comprehensively deals with monolithic systems, multiple challenges are presented when scaling existing approaches to decentralized systems, that is, systems with multiple components with no central observation point. These challenges are inherent to the nature of decentralization; the monitors have a partial view of the system and need to account for communication and consensus. Our assumptions on the system are as follows: No monitors are malicious, i.e., messages do not contain wrong information; No messages are lost, they are eventually delivered in their entirety but possibly out-of-order; All components share one logical discrete time marked by round numbers indicating relevant transitions in the system specification.

Example 1.1 (Switch triggers bulb). Consider a system that contains two components: a light switch and a light bulb. The possible states of the switch and bulb can be *on* or *off*. Let us associate the states of the switch and bulb with the atomic propositions s and ℓ, respectively. Using the atomic propositions, we can encode the observations about the system. For example, the observation $\langle \mathrm{s}, \bot \rangle$ indicates that the switch is in the state *off*. An event is simply a set of observations. The event $\{\langle \mathrm{s}, \top \rangle, \langle \ell, \bot \rangle\}$ indicates that the switch is *on*, and the light is *off*. We next define (informally) a property of the system: *"The light bulb must be on one timestamp after the switch is on, until the switch is turned off"*. The property is used to synthesize a monitor that checks it (for example, the monitor in Figure 1, introduced in Example 2.1). Now let us consider two traces: $\mathrm{tr}_0 \stackrel{\text{def}}{=} \{\langle \mathrm{s}, \bot \rangle, \langle \ell, \bot \rangle\} \cdot \{\langle \mathrm{s}, \top \rangle, \langle \ell, \bot \rangle\} \cdot \{\langle \mathrm{s}, \top \rangle, \langle \ell, \top \rangle\} \cdot \{\langle \mathrm{s}, \bot \rangle, \langle \ell, \bot \rangle\}$ and $\mathrm{tr}_1 \stackrel{\text{def}}{=} \{\langle \mathrm{s}, \bot \rangle, \langle \ell, \bot \rangle\} \cdot \{\langle \mathrm{s}, \top \rangle, \langle \ell, \bot \rangle\} \cdot \{\langle \mathrm{s}, \top \rangle, \langle \ell, \bot \rangle\}$. We see that tr_0 complies with the specification, while tr_1 violates it as the light is not turned on after the switch is turned on. When monitoring such a system, we can construct a decentralized monitor of the property. For this, we need to decide on where to deploy the monitors, and what will each monitor check. By being able to vary monitor placement, we are able to control the resources needed to monitor on a given component of the system. Furthermore, due to the nature of decentralization, each monitor will have a partial view of the system. While the monitor on the switch can see the value of atomic proposition s, it cannot determine that of ℓ, and as such, it must communicate with other monitors to establish it.

Challenges. Several algorithms have been designed [8, 10, 11, 14, 15, 28] and used [4] to monitor decentralized systems. Algorithms are primarily designed to address one issue at a time and are typically experimentally evaluated by considering runtime and memory overheads. However, such algorithms are difficult to compare as they may combine multiple approaches at once. For example, algorithms that use LTL rewriting [10, 15, 45] not only exhibit variable runtime behavior due to the rewriting, but also incorporate different monitor synthesis approaches that separate the specification into multiple smaller specifications. Such techniques start from a global specification and then synthesize local monitors with either a copy of the global specification [8, 11] or a completely different specification to monitor (typically a subformula of the original formula) [14, 28]. In this paper, we refer to the former as a centralized specification and to the latter as a decentralized specification. These different approaches of synthesis are separate from monitoring and their evaluation is of interest. In this case, we would like to split the problem of generating equivalent

decentralized specifications from a centralized one (synthesis) from the problem of monitoring. In addition, works on characterizing what one can monitor (i.e., monitorability [29, 37, 44]) for centralized specifications exist [9, 19, 29], but do not extend to decentralized specifications. For example, by splitting an LTL formula ad hoc, it is possible to obtain a non-monitorable subformula[1] which interferes with the completeness of a monitoring algorithm.

Contributions. We tackle the presented challenges using two complementary approaches. The first approach consists in using the data structure *Execution History Encoding* (EHE) that encodes automata executions. Since by using EHE one only needs to rewrite Boolean expressions, we are able to determine the parameters and their respective effect on the size of expressions, and fix upper bounds. In addition, EHE is designed to be particularly flexible in processing, storing and communicating the information in the system. EHE operates on an encoding of atomic propositions and guarantees strong-eventual consistency [48]. The second approach introduces decentralized specifications. We introduce decentralized specifications, define their semantics, interdependencies and study some of their properties. We aim at abstracting the high-level steps of decentralized monitoring. By identifying these steps, we elaborate a general decentralized monitoring algorithm. We view a decentralized system as a set of components C. A decentralized specification is thus as a set of n finite-state automata with specific properties, which we call *monitors*. We associate n monitors to these components with the possibility of multiple monitors being associated to a component. Therefore, we generalize monitoring algorithms to multiple monitors. Monitoring a centralized system can be seen as a special case with one component, one specification, and one monitor. As such, we present a general decentralized monitoring algorithm that uses two high level steps: setup and monitor. The setup phase creates the monitors, defines their dependencies and attaches them to components. As such, the setup phase defines a *topology* of monitors and their dependencies. The monitor phase allows the monitors to begin monitoring and propagating information to reach a verdict when possible. Therefore, the two high-level operations help decompose monitoring into different subproblems and define them independently. For example, the problem of generating a decentralized specification from a centralized specification is separated from checking the monitorability of a specification, and also separated from the computation and communication performed by the monitor. We formulate and solve the problems of deciding *compatibility* and *monitorability* for decentralized specifications. *Compatibility* ensures that a monitor topology can be deployed on a given system, *monitorability* ensures that given a specification, monitors are able to eventually emit a verdict, for all possible traces. We present THEMIS, a JAVA tool that implements the concepts in this paper; and show how it can be used to design and analyze new algorithms. We use THEMIS to create new metrics related to load-balancing and our data structures. We use two scenarios to compare four existing algorithms. The first scenario is a synthetic benchmark, using random traces and specifications, while the second scenario is a real example that uses the publish-subscribe pattern in the *Chiron* graphical user interface system. The synthetic scenario examines the trends of the analysis, and the *Chiron* scenario examines more specific differences in behavior.

This paper extends the work presented at the ACM SIGSOFT International Symposium on Software Testing and Analysis 2017 [23], as follows:

- introducing a high-level informal overview of the approach (Section 2);
- improving on the clarity by providing a running example that tackles all introduced concepts;
- adding the property that the EHE construction guarantees its determinism (Proposition 4.8);
- elaborating and adding properties of decentralized specifications (monitorability, compatibility) as well as the algorithms for checking them (Section 6);

[1]We use the example from [15]: $\mathbf{GF}(a) \wedge \neg(\mathbf{GF}(a))$ (where $\mathbf{GF}(a)$ means that a should hold infinitely often) is monitorable, but its subformulas are both non-monitorable.

- improving THEMIS by optimizing the EHE performance, and adding distributed and multi-threaded support (Section 8);
- elaborating on the results and providing a discussion of the synthetic benchmarks (Section 9.1);
- including additional insight on the effect of network delay on the EHE size (Section 9.1.7);
- evaluating the algorithms on a new use case based on the *Chiron* example that relies on publish subscribe and has a formalized specification (Section 9.2);
- extending related work (Section 10); and
- formulating additional problems for the future (Section 11).

Overview. We begin by introducing an informal view of the approaches in Section 2, by first distinguishing between centralized and decentralized for two aspects of RV: the specification and the monitoring itself. Then, we introduce a high-level view of decentralized specifications semantics, advantages, and the need to manage partial observations. We lay out the basic blocks, by introducing our basic data structure (`dict`), and the basic notions of monitoring with expressions in Section 3. We present our first approach, a middle ground between rewriting and automata evaluation by introducing the Execution History Encoding (EHE) data structure in Section 4. The EHE is designed to be particularly useful for handling partial observations. We shift the focus on studying decentralized specifications by defining their semantics (Section 5), and their properties (Section 6). In Section 7, we use our analysis of EHE to study the behavior of three existing algorithms and discuss the situations that advantage certain algorithms over others. In Section 8, we present the THEMIS tool, which we use in Section 9 to compare the algorithms presented in Section 7 under two different scenarios: a synthetic random benchmark, and an example of a publish-subscribe system. In Section 10, we present related work that covers decentralized monitoring using rewriting, global predicate detection, and streams. In Section 11, we present future work and formulate additional interesting properties for decentralized specifications. Finally, we conclude in Section 12.

2 METHODOLOGICAL OVERVIEW

In this section, we discuss the basic terminology referenced in the paper. We distinguish between centralized and decentralized for two aspects of RV: the specification and the monitoring itself. The distinction allows us to introduce and overview our approach to monitoring decentralized specifications.

2.1 Centralized Monitoring of a Centralized Specification

An *LTL3 monitor* is a typical automaton used in RV (c.f. [9, 28]). An LTL3 monitor is a complete, minimal, and deterministic Moore automaton where states are labeled with the verdicts in the set $\mathbb{B}_3 = \{\top, \bot, ?\}$, and transitions are labeled with atomic propositions, which are used to represent abstract states of the system. Verdicts $\top$ and $\bot$ respectively indicate that the current execution complies and does not comply with the specification, while verdict ? indicates that the verdict has not been determined yet. Verdicts $\top$ and $\bot$ are called "final", as once the monitor outputs $\top$ or $\bot$ for a given trace, it cannot output a different verdict for any extension of that trace.

Example 2.1 (LTL3 monitor). We introduced in Example 1.1 the system composed of a lightbulb and a switch. The LTL_3 monitor that checks for its specification is presented in Figure 1. The automaton consists of three states: q_0, q_1, and q_2 associated respectively with the verdicts ?, ?, and $\bot$. Upon reaching q_2, the verdict is final as it can no longer change. The final verdict indicates that, at some point in the execution, the light was off while the switch was on.

In the case of Example 2.1, it is possible to imagine one monitor running (with or alongside) the program, and having access to the global state of the program. We refer to such a scenario

as *centralized monitoring of a centralized specification*. There is one global specification of the system, being checked by a given monitor that has access to all the information about the atomic propositions.

2.2 Decentralized Monitoring of Centralized Specifications

Section 2.1 introduces a monitor capable of observing all of the atomic propositions. Such monitor is able to immediately take transitions upon receiving observations. However, it is not always the case that we have a central point of observation. That is, it is possible that atomic propositions are not all observable by one monitor. Such a scenario is typically called *decentralized monitoring*, and is the topic of many research efforts (see Section 10). The monitoring is done by various monitors communicating to determine the global state efficiently, and be able to check the property.

Example 2.2 (Decentralized monitoring). Consider the monitor presented in Example 2.1. We can see that the switch and lightbulb are two separate components, a monitor placed on the switch cannot observe any atomic proposition related to the lightbulb. In this case, the associated RV technique considers the global specification of the system, and then proceeds to create monitors and determine their communication patterns.

We note that the monitors are all monitoring *one global specification*. The main challenge is that monitors must deal with partial observations. We refer to such a scenario as *decentralized monitoring of a centralized specification*, as all monitors are verifying the same specification. Details on centralized and decentralized monitoring of a centralized specification are provided in Section 4.

2.3 Decentralized Monitoring of Decentralized Specifications

We noticed that so far, *decentralized monitoring* allows for several monitors that monitor the *same* specification. Typically, a decentralized monitoring algorithm will consider one (global) specification, and either generate necessary monitors that are tasked with monitoring a part of the specification, or allow for consensus among multiple monitors to find the global verdict[2]. In this paper, we focus on multiple monitors each having their own independent specification, of which others are normally unaware. We thus focus on *decentralized monitoring of decentralized specifications*[3].

2.3.1 Informal Semantics. Informally, a decentralized specification considers the system as a set of components, defines a set of LTL3 monitors (see Section 2.1), additional atomic propositions that represent references to monitors, and attaches each monitor to a component. Attaching monitors to components allows a monitor specification to explicitly reference atomic propositions that are associated with the component. However, the transition labels in a monitor are restricted to only atomic propositions related to the component on which the monitor is attached, and references to other monitors.

A monitor reference is evaluated as if it were an oracle as shown in Figure 2. That is, to evaluate a monitor reference m_j, in a monitor $\mathcal{A}_i$, at a timestamp n, the monitor referenced ($\mathcal{A}_j$) is executed starting from the initial state by looking at observations in the trace starting at n. The atomic proposition m_j at n takes the value of the final verdict reached by the monitor $\mathcal{A}_j$ starting its evaluation from n. Details of the semantics are provided in Section 5. Furthermore, to evaluate reference we need the resulting oracle execution to be able to reach a final verdict, which is not always guaranteed. As such, it is important to define some of the properties of decentralized

[2]See Section 10 for details.

[3]We note that centralized monitoring of decentralized specifications makes little sense as there does not exist more than one monitor.

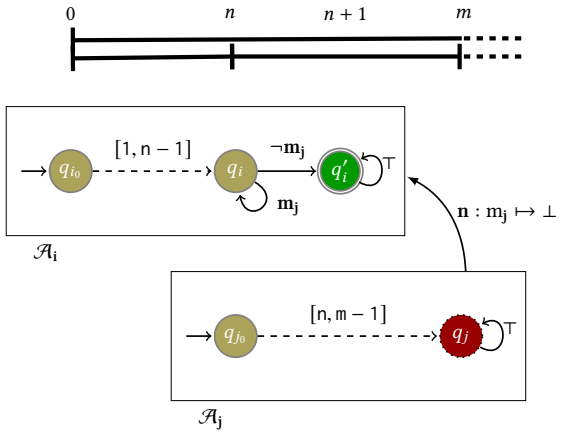


Fig. 2. Evaluating monitor references.

specifications such as *monitorability*, which indicates that a final verdict is co-reachable from any state in a given monitor (Section 6.1). We elaborate on characterizing and computing properties of decentralized specifications in Section 6.

2.3.2 Managing Partial Observations. We notice that monitor references are evaluated as if we were evaluating partial observations that we receive in the future. In fact, managing partial observations is important for decentralized monitoring in general, including decentralized monitoring of centralized specifications (Section 2.2). To that end, we introduce the *Execution History Encoding* (EHE) data structure in Section 4.2 that allows us to monitor in the presence of partial observations. The EHE is useful for two main reasons.

First, since we deal with partial observations, it is important to keep track of potential states an automaton could be in, and manage the guesses as information is eventually known, in a uniform way, this applies for *decentralized monitoring* in general. This helps us compare algorithms that rely on partial observations (Sections 7, 8 and 9).

Second, since we do not deal with loss of messages in the system, monitors are always guaranteed to receive observations eventually. Therefore, we are interested in designing the EHE so that it can replicate its state under strong eventual consistency (SEC) [48]; any two monitors eventually exchanging EHEs should have the same view of the system. This is only useful if monitors are monitoring the same automaton, thus it is mainly useful for *decentralized monitoring of centralized specifications*. Section 4.3 elaborates on the usage of EHE where multiple monitors are monitoring the same specification.

2.3.3 Monitor Placement. Being able to decide the specification and monitor placement is particularly useful for applications such as internet of things. Applications can be tailored to the computational resources and also to the proximity of various devices and the network itself. Recall Example 2.2, if more computing resources are given to the switch component than the lightbulb, then it is possible for the monitor on the switch to perform the monitoring computations, while the one on the lightbulb merely forwards observations. Alternatively, they can both forward the observations to another component which can then perform the monitoring computations. The decentralized specification determines what each monitor will check as well as the dependencies between the various monitors in the system.

2.3.4 Advantages of Using References. We mentioned in Section 2.3.1 that references are not always guaranteed to return a final verdict, and we need specific properties to ensure so. However,

references are particularly useful when synthesizing monitors from LTL as in [9], since the algorithm is doubly exponential in the number of atomic propositions. Offsetting parts of the subformula to other monitors allows us to have a lower number of atomic propositions in a given specification. Additionally, since references are treated as oracles, this allows specifications to be constructed in a modular way, allowing monitors to not even be aware of the subspecification.

Example 2.3 (Synthesizing check light). Recall the property in Example 2.1 responsible for verifying that a light switch does indeed turn a light bulb on until the switch is turned off. Suppose we have multiple rooms with multiple such lightbulbs and switches, the LTL formula parametrized by each pair can be expressed in LTL as follows: $\texttt{sc\_light}(i) \stackrel{\text{def}}{=} \mathbf{G}(\mathrm{s}_i \implies \mathbf{X}(\ell_i \,\mathbf{U}\, \neg \mathrm{s}_i))$. To verify the property across all n pairs, we formulate a property $\texttt{sc\_ok} \stackrel{\text{def}}{=} \bigwedge_{i \in [0..n]} \texttt{sc\_light}(i)$. In the case of a decentralized specification the formula will reference each monitor leading to a conjunction of n atomic propositions. However, in the case of a centralized specification, the specification needs to be written as: $\texttt{sc\_ok}^{\text{cent}} \stackrel{\text{def}}{=} \bigwedge_{i \in [0 \ldots n]} \mathbf{G}(\mathrm{s}_i \implies \mathbf{X}(\ell_i \,\mathbf{U}\, \neg \mathrm{s}_i))$, which is significantly more complex as a formula consisting of $4n$ operators (to cover the subspecification), along n conjunctions, and defined over each sensor and light bulb atomic propositions ($2n$). Given that monitor synthesis is doubly exponential, `ltl2mon` [9] requires significant resources and time to generate the minimal Moore automaton (in our case[4], it was unable to generate the monitor for $n = 3$ after a timeout of one hour).

3 COMMON NOTIONS

We introduce the dict data structure (Section 3.1.) used to build more complex data structures, and define the basic concepts for decentralized monitoring (Section 3.2).

3.1 The `dict` Data Structure

In monitoring decentralized systems, monitors typically have a state, and attempt to merge other monitor states with theirs to maintain a consistent view of the running system, that is, at no point in the execution, should two monitors receive updates that conflict with one another. In addition, we would like that any two monitors receiving the same information be in equivalent states. Therefore, we are interested in designing data structures that can replicate their state under strong eventual consistency (SEC) [48], they are known as state-based convergent replicated data-types (CvRDTs). We use a dictionary data structure (noted `dict`) as our basic building block that assigns a value to a given key. Data structure `dict` will be used to define the memory of a monitor (Section 3.2), and data structure `EHE` which encodes the execution of an automaton (Section 4.2).

We model `dict` as a partial function f. The domain of f (denoted by dom(f)) is the set of keys, while the codomain of f (denoted by codom(f)) is the set of values. `dict` supports two operations: `query` and `merge`. The `query` operation checks if a key $k \in \mathrm{dom(f)}$ and returns $\mathrm{f}(k)$. If $k \notin \mathrm{dom(f)}$, then it is undefined. The `merge` operation of a `dict` f with another `dict` g, is modeled as function composition. Two partial functions f and g are composed using operator $\dagger_{op}$ where $op : (\mathrm{dom(f)} \times \mathrm{dom(g)}) \to (\mathrm{codom(f)} \cup \mathrm{codom(g)})$ is a binary function.

$$\mathrm{f} \dagger_{op} \mathrm{g} : \mathrm{dom(f)} \cup \mathrm{dom(g)} \to \mathrm{codom(f)} \cup \mathrm{codom(g)}$$

$$\mathrm{f} \dagger_{op} \mathrm{g}(x) = \begin{cases} op(\mathrm{f}(x), \mathrm{g}(x)) & \text{if } x \in \mathrm{dom(f)} \cap \mathrm{dom(g)} \\ \mathrm{g}(x) & \text{if } x \in \mathrm{dom(g)} \setminus \mathrm{dom(f)} \\ \mathrm{f}(x) & \text{if } x \in \mathrm{dom(f)} \setminus \mathrm{dom(g)} \\ \text{undef} & \text{otherwise} \end{cases}$$

[4]On an Intel(R) Core(TM) i7-6700HQ CPU, using 16GB RAM, and running openjdk 1.8.0_172, with ltl2mon 0.0.7.

On sets of functions, $\dagger_{op}$ applies pairwise: $\biguplus^{op}\{f_1, \dots f_n\} = ((f_1 \dagger_{op} f_2) \dots f_n)$. The following two operators are used in the rest of the paper: $\dagger_2$ and $\dagger_\vee$. We define both of these operators to be commutative, idempotent, and associative to ensure SEC.

$$\dagger_2(x, x') = \begin{cases} x' & \text{if } x \prec x' \\ x & \text{otherwise} \end{cases} \qquad \dagger_\vee (x, x') = x \vee x'$$

Operator $\dagger_2$ acts as a replace function based on a total order ($\prec$) between the elements, so that it always chooses the highest element to guarantee idempotence, while $\dagger_\vee$ uses the logical *or* operator to combine elements. Respectively, we denote the associated pairwise set operators by $\biguplus^2$ and $\biguplus^\vee$.

Data structure `dict` can be composed by only using operation `merge`. The modifications never remove entries, the state of `dict` is then monotonically increasing using the order provided by `merge`. By ensuring that `merge` is idempotent, commutative, and associative we fulfill the necessary conditions [48] for our data structure to be a CvRDT (Proposition 3.1).

Proposition 3.1. *Data structure* `dict` *with operations* `query` *and* `merge` *is a CvRDT.*

3.2 Basic Monitoring Concepts

We recall the basic building blocks of monitoring. We consider the set of verdicts $\mathbb{B}_3 = \{\top, \bot, ?\}$ to denote the verdicts true, false, not reached (or inconclusive) respectively. A verdict in $\mathbb{B}_2 = \{\top, \bot\}$ is a *final* verdict. It indicates that the monitor has concluded its monitoring, and any further input will not change affect it. Abstract states of a system are represented as a set of *atomic propositions* (*AP*). A monitoring algorithm typically includes additional information such as a timestamp associated with the atomic propositions. We capture this information as an encoding of the atomic propositions (*Atoms*), this encoding is left to the monitoring algorithm to specify.

Definition 3.2 (Event). An observation is a pair in $AP \times \mathbb{B}_2$ indicating whether or not a proposition has been observed. An event is a set of observations in $2^{AP \times \mathbb{B}_2}$.

Example 3.3 (Event). We recall the example of a light switch and bulb from Example 1.1. We have $AP = \{s, \ell\}$. The event $\{\langle s, \top\rangle, \langle \ell, \bot\rangle\}$ indicates that the switch is observed to be on (i.e., the atomic proposition s is observed to be true), while the light bulb is observed to be off (i.e., the atomic proposition ℓ is observed to be false).

A decentralized monitoring algorithm requires retaining, retrieving and communicating observations. Monitoring algorithms are versatile, and may require additional information associated with atomic propositions. This information can include timestamps indicating when the atomic proposition was observed, or a component ID, to determine where the atomic proposition was observed. As such, when stored, atomic propositions are typically encoded to add this additional information by the monitoring algorithm. To abstract the additional information, and remain general, the monitors store the encoded atomic proposition (instead of the atomic proposition itself), the encoded atomic proposition is referred to as *atom*. $Expr_{Atoms}$ (resp. $Expr_{AP}$) denotes the set of Boolean expressions over *Atoms* (resp. *AP*). An encoder is a function $\text{enc} : Expr_{AP} \to Expr_{Atoms}$ that encodes the atomic propositions into atoms. In this paper, we use two encoders: idt which is the identity function (it does not modify the atomic proposition), and ts_t which adds a timestamp t to each atomic proposition. The identity encoder is mainly used with automata as its transitions are labelled by *AP*, and the timestamp encoder is used when manipulating the execution history encoding introduced in Section 4.2, as it encodes information about rounds in the expressions.

Definition 3.4 (Memory). A memory is a `dict`, and is modeled as a partial function $\mathcal{M} : Atoms \to \mathbb{B}_3$ that associates an atom to a verdict. The set of all memories is defined as *Mem*.

An event can be converted to a memory by encoding the atomic propositions to atoms, and associating their truth value: $\text{memc} : 2^{AP \times \mathbb{B}_2} \times (Expr_{AP} \to Expr_{Atoms}) \to Mem$.

Example 3.5 (Memory). We recall from Example 3.3 the event: $evt = \{\langle \text{s}, \top\rangle, \langle \ell, \bot\rangle\}$. At $t = 1$, the resulting memories using encoders idt and ts_1 are: $\text{memc}(evt, \text{idt}) = [\text{s} \mapsto \top, \ell \mapsto \bot]$, $\text{memc}(evt, \text{ts}_1) = [\langle 1, \text{s}\rangle \mapsto \top, \langle 1, \ell\rangle \mapsto \bot]$, respectively.

If we impose that *Atoms* be a totally ordered set, then two memories $\mathcal{M}_1$ and $\mathcal{M}_2$ can be merged by applying operator $\dagger_2$. The total ordering is needed for operator $\dagger_2$. This ensures that the operation is idempotent, associative and commutative. Monitors that exchange their memories and merge them have a consistent snapshot of the memory, regardless of the ordering. Since a memory is a `dict` and $\dagger_2$ is idempotent, associative, and commutative, it follows from Proposition 3.1 that a memory is a CvRDT (Corollary 3.6).

Corollary 3.6. *A memory with operation $\dagger_2$ is a CvRDT.*

In this paper, we perform monitoring by manipulating expressions in $Expr_{Atoms}$. The first operation we provide is rw, which rewrites the expression to attempt to eliminate *Atoms*.

Definition 3.7 (Rewriting an expression). An expression e is rewritten with a memory $\mathcal{M}$ using function $\text{rw} : Expr_{Atoms} \times Mem \to Expr_{Atoms}$ defined as follows:

$$
\begin{aligned}
\text{rw}(e, \mathcal{M}) = \ & \texttt{match } e \texttt{ with} \\
& \mid a \in Atoms \quad \to \begin{cases} \mathcal{M}(a) & \text{if } a \in dom(\mathcal{M}) \\ a & \text{otherwise} \end{cases} \\
& \mid \neg e' \quad \to \neg \text{rw}(e', \mathcal{M}) \\
& \mid e_1 \wedge e_2 \quad \to \text{rw}(e_1, \mathcal{M}) \wedge \text{rw}(e_2, \mathcal{M}) \\
& \mid e_1 \vee e_2 \quad \to \text{rw}(e_1, \mathcal{M}) \vee \text{rw}(e_2, \mathcal{M})
\end{aligned}
$$

Using information from a memory $\mathcal{M}$, the expression is rewritten by replacing atoms with a final verdict (a truth value in $\mathbb{B}_2$) in $\mathcal{M}$ when possible. Atoms that are not associated with a final verdict are kept in the expression. Operation rw yields a smaller formula to work with.

Example 3.8 (Rewriting). We extend the set of atomic propositions from Example 3.3 to include a motion sensor. We associate the motion sensor state with the atomic proposition pres, where if pres is observed to be $\top$, then the sensor is detecting motion. We have $AP = \{\text{s}, \ell, \text{pres}\}$. We consider the memory from Example 3.5: $\mathcal{M} = [\text{s} \mapsto \top, \ell \mapsto \bot]$; and an expression $e = (\text{s} \vee \ell) \wedge \text{pres}$. In this case, we want to check if the light or switch are on only when the motion detects presence. We have $\mathcal{M}(\text{s}) = \top$, $\mathcal{M}(\ell) = \bot$, $\mathcal{M}(\text{pres}) = ?$. Since pres is associated with $? \notin \mathbb{B}_2$ then it will not be replaced when the expression is evaluated. The resulting expression is $\text{rw}(e, \mathcal{M}) = (\top \vee \bot) \wedge \text{pres}$.

We eliminate additional atoms using Boolean logic. We denote by simplify(`expr`) the simplification of expression `expr` [5].

Example 3.9 (Simplification). Using the same setting as Example 3.8, we consider memory $\mathcal{M} = [\text{s} \mapsto \top]$ and expression $e = (\text{s} \wedge \ell) \vee (\text{s} \wedge \neg\ell)$. We have $e' = \text{rw}(e, \mathcal{M}) = (\ell \vee \neg\ell)$. We notice that rewriting e' does not yield a final verdict. However, atoms can be eliminated with simplify(e'). We finally get $\top$.

We combine both rewriting and simplification in the eval function which determines a verdict from an expression e.

[5] This is also known as The Minimum Equivalent Expression problem [12].

Definition 3.10 (Evaluating an expression). The evaluation of a Boolean expression $e \in Expr_{Atoms}$ using a memory $\mathcal{M}$ yields a verdict. Function eval : $Expr_{Atoms} \times Mem \to \mathbb{B}_3$ is defined as:

$$\text{eval}(e, \mathcal{M}) = \begin{cases} \top & \text{if simplify}(\text{rw}(e, \mathcal{M}) \Leftrightarrow \top, \\ \bot & \text{if simplify}(\text{rw}(e, \mathcal{M}) \Leftrightarrow \bot, \\ ? & \text{otherwise.} \end{cases}$$

Function eval returns the verdict $\top$ (resp. $\bot$) if the simplification after rewriting is (Boolean) equivalent to $\top$ (resp. $\bot$), otherwise it returns verdict ?.

Example 3.11 (Evaluating expressions). We recall from Example 3.8 memory $\mathcal{M} = [\text{s} \mapsto \top, \ell \mapsto \bot]$; and expression $e = (\text{s} \vee \ell) \wedge \text{pres}$. We have simplify(rw($e, \mathcal{M}$)) = simplify($(\top \vee \bot) \wedge \text{pres}$) = pres, and eval($e, \mathcal{M}$) = ? which depends on pres. We cannot emit a final verdict before observing pres.

A decentralized system is a set of components C. We assign a sequence of events to each component using a decentralized trace function.

Definition 3.12 (Decentralized trace). A decentralized trace of length n is a total function tr : $[1, n] \times C \to 2^{AP \times \mathbb{B}_2}$ (where $[1, n]$ denotes the interval of the n first non-zero natural numbers).

Function tr assigns an event to a component for a given timestamp. We denote by $\mathcal{T}$ the set of all possible decentralized traces. We additionally define function lu : $AP \to C$ to assigns an atomic proposition to a component. We assume that (1) no two components can observe the same atomic propositions[6], and (2) at least one atomic proposition is associated with a component (a component with no atomic propositions to monitor, can be simply considered excluded from the system under monitoring). Function lu is defined as $\text{lu}(ap) = c$ s.t. $\exists t \in \mathbb{N}, \exists v \in \mathbb{B}_2 : \langle ap, v \rangle \in \text{tr}(t, c)$.

We consider timestamp 0 to be associated with the initial state, therefore our traces start at 1. The length of a trace tr is denoted by |tr|. An empty trace has length 0 and is denoted by ϵ. Monitoring using LTL or finite-state automata relies on sequencing the trace. Events must be totally ordered. A timestamp indicates simply the order of the sequence of events. As such, a timestamp represents a logical time, it can be seen as a *round number*. Every round consists in a transition taken on the automaton after reading a part of the word. While tr gives us a view of what components can locally see, we reconstruct the global trace to reason about all observations. A global trace of the system is therefore a sequence of events. A global trace encompasses all observations observed locally by components. While a global trace will never be used in practice, we use it for the purpose of reasoning about the global state (Section 4.1), and ensuring the correctness of our approach (Proposition 4.10).

Definition 3.13 (Reconstructing a global trace). Given a decentralized trace tr of length n, we reconstruct the global trace using function $\rho : \left([1, n] \times C \to 2^{AP \times \mathbb{B}_2}\right) \to \left([1, n] \to 2^{AP \times \mathbb{B}_2}\right)$ defined as $\rho(\text{tr}) = evt_1 \cdot \ldots \cdot evt_n$ s.t. $\forall i \in [1, n] : evt_i = \bigcup_{c \in C} \text{tr}(i, c)$.

For each timestamp $i \in [1, n]$, we take all observations of all components and union them to get a global event. Consequently, an empty trace yields an empty global trace, $\rho(\epsilon) = \epsilon$.

Example 3.14 (Traces). Using the switch and light bulb from Example 1.1, we define multiple components. We consider a system of two components lswitch and bulb, that are associated with atomic propositions s and ℓ respectively. An example decentralized trace of the system is given by tr = $[1 \mapsto \text{lswitch} \mapsto \{\langle \text{s}, \top \rangle\}, 1 \mapsto \text{bulb} \mapsto \{\langle \ell, \top \rangle\}, 2 \mapsto \text{lswitch} \mapsto \{\langle \text{s}, \top \rangle\}, 2 \mapsto \text{bulb} \mapsto \{\langle \ell, \bot \rangle\}]$. That is, component lswitch observes proposition s to be $\top$ at both timestamps 1 and 2, while bulb observes ℓ to be $\top$ at timestamp 1 and $\bot$ at timestamp 2. The associated global trace is: $\rho(\text{tr}) = \{\langle \text{s}, \top \rangle, \langle \ell, \top \rangle\} \cdot \{\langle \text{s}, \top \rangle, \langle \ell, \bot \rangle\}$.

[6]This is not necessary, it makes the presentation clearer. For components sharing observations, we can encode their own ID in the atom to make it unique.

4 CENTRALIZED SPECIFICATIONS

We now focus on a decentralized system specified by one global automaton. We consider automata that emit 3-valued verdicts in the domain $\mathbb{B}_3$, similar to those in [9, 15] for centralized systems. Using automata with 3-valued verdicts has been the topic of a lot of the Runtime Verification literature [7, 9, 10, 15, 28], we focus on extending the approach for decentralized systems in [15] to use a new data structure called Execution History Encoding (EHE). Typically, monitoring is done by labeling an automaton with events, then playing the trace on the automaton and determining the verdict based on the reached state. We present the EHE, a data structure that encodes the necessary information from an execution of the automaton. Monitoring using EHEs ensures strong eventual consistency. We begin by defining the specification automaton used for monitoring in Section 4.1, then we present the EHE data structure, illustrate its usage for monitoring in Section 4.2, and describe its use to reconcile partial observations in Section 4.3.

4.1 Preliminaries

Specifications are similar to the Moore automata generated by [9]. We modify labels to be Boolean expressions over atomic propositions (in $Expr_{AP}$). We choose to label the transitions with Boolean expressions as opposed to events, to keep a homogeneous representation (with EHE)[7].

Definition 4.1 (Specification). The specification is a deterministic Moore automaton $\langle Q, q_0, \delta, \text{ver}\rangle$ where $q_0 \in Q$ is the initial state, $\delta : Q \times Expr_{AP} \to Q$ is the transition function and $\text{ver} : Q \to \mathbb{B}_3$ is the labeling function.

The labeling function associates a verdict with each state. We assume that by construction the state with final verdicts are sink states (as synthesized in [9]).

When using multiple automata we use labels to separate them, $\mathcal{A}_\ell = \langle Q_\ell, q_{\ell_0}, \delta_\ell, \text{ver}_\ell\rangle$. We fix $\mathcal{A}$ to be a specification automaton for the remainder of this section. For monitoring, we are interested in events (Definition 3.2), we extend δ to events, and denote it by Δ[8].

Definition 4.2 (Transition over events). Given an event evt, we build the memory $\mathcal{M} = \text{memc}(evt, \text{idt})$. Then, function $\Delta : Q \times 2^{AP \times \mathbb{B}_2} \to Q$ is defined as follows:

$$\Delta(q, evt) = \begin{cases} q' & \text{if } evt \neq \emptyset \wedge \exists q' \in Q, \exists e \in Expr_{AP} : \delta(q, e) = q' \wedge \text{eval}(e, \mathcal{M}) = \top, \\ q & \text{otherwise.} \end{cases}$$

A transition is taken only when an event contains observations (i.e., $evt \neq \emptyset$). This allows the automaton to wait on observations before evaluating, as such it remains in the same state (i.e., $\Delta(q, \emptyset) = q$). Upon receiving observations, we use $\mathcal{M}$ to evaluate each label of an outgoing transition, and determine if a transition can be taken (i.e., $\exists q' \in Q, \exists e \in Expr_{AP} : \delta(q, e) = q' \wedge \text{eval}(e, \mathcal{M}) = \top$).

To handle a trace, we extend Δ to its reflexive and transitive closure in the usual way, and note it Δ^*. For the empty trace, the automaton makes no moves, i.e., $\Delta^*(q_0, \epsilon) = q_0$.

Example 4.3 (Monitoring using expressions). Recall the monitor from Example 2.1 monitoring the light switch and bulb interaction. Let us consider the global trace from Example 3.14: $evt_0 \cdot evt_1$, with $evt_0 = \{\langle \text{s}, \top\rangle, \langle \ell, \top\rangle\}$ and $evt_1 = \{\langle \text{s}, \top\rangle, \langle \ell, \bot\rangle\}$. The resulting memory at $t = 1$ is $\mathcal{M} = \text{memc}(evt_0, \text{idt}) = [\text{s} \mapsto \top, \ell \mapsto \top]$ (see Example 3.5). The transition from q_0 to q_1 is taken since $\text{eval}(\text{s}, \mathcal{M}) = \top$. Thus we have $\Delta(q_0, evt_0) = q_1$ with verdict $\text{ver}(q_1) = ?$. We continue by repeating the process for $t = 2$. The memory is $\mathcal{M}' = \text{memc}(evt_1, \text{idt}) = [\text{s} \mapsto \top, \ell \mapsto \bot]$. The transition from

[7]Indeed, an event can be converted to an expression by the conjunction of all observations, negating the terms that are associated with the verdict $\bot$.

[8]We note that in this case, we are not using any encoding ($Atoms = AP$).

q_1 to q_2 is taken since $\text{eval}(s \wedge \neg\ell, \mathcal{M}) = \top$. Thus we have $\Delta(q_1, evt_1) = q_2$ with verdict $\text{ver}(q_2) = \bot$. We can see that for this trace, the property is violated.

Remark 1 (Properties and normalization). *We recall that the specification is a deterministic and complete automaton. Hence, there are properties on the expressions that label the transition function. For any $q \in Q$, we have:*
1) $\forall \mathcal{M} \in Mem : (\exists \langle q, e\rangle \in \text{dom}(\delta) : \text{eval}(e, \mathcal{M}) = \top) \implies (\nexists \langle q, e'\rangle \in \text{dom}(\delta) \setminus \{\langle q, e\rangle\} : \text{eval}(e', \mathcal{M}) = \top)$; and
2) the disjunction of the labels of all outgoing transitions results in an expression that is a tautology.
The first property states that for all possible memories encoded with idt *no two (or more) labels can evaluate to $\top$ at once. It results from determinism: no two (or more) transitions can be taken at once. The second property results from completeness: given any input, the automaton must be able to take a move. Furthermore, we note that for each pair of states $\langle q, q'\rangle \in Q \times Q$, we can rewrite δ such that there exists at most one expression $e \in Expr_{AP}$, such that $\delta(q, e) = q'$, without loss of generality. This is because for a pair of states, we can always disjoin the expressions to form only one expression, as it suffices that only one expression needs to evaluate to $\top$ to reach q'. By having at most one transition between any pair of states, we simplify the topology of the automaton.*

4.2 The Execution History Encoding (EHE) Data Structure

The execution of the specification automaton, is in fact, the process of monitoring, upon running the trace, the reached state determines the verdict. An execution of the specification automaton can be seen as a sequence of states $q_0 \cdot q_1 \cdot \ldots q_t \ldots$. It indicates that, for each timestamp $t \in \mathbb{N}^*$, the automaton is in the state q_t[9]. In a decentralized system, a component receives only local observations and does not necessarily have enough information to determine the state at a given timestamp. Typically, when sufficient information is shared between various components, it is possible to know the state q_t that is reached in the automaton at t (we say that the state q_t has been found, in such a case). The aim of the EHE is to construct a data structure which follows the current state of an automaton, and in case of partial information, tracks the possible states the automaton can be in. For that purpose, we need to ensure strong eventual consistency in determining the state q_t of the execution of an automaton. That is, after two different monitors share their EHE, they should both be able to find q_t for t (if there exists enough information to infer the global state), or if not enough information is available, they both find no state at all.

Execution History Encoding (EHE) is a data structure designed to encode an execution of an automaton using boolean expressions while accounting for partial observations.

Definition 4.4 (Execution History Encoding - EHE). An Execution History Encoding (EHE) of the execution of an automaton $\mathcal{A}$ is a partial function $\mathcal{I} : \mathbb{N} \times Q \to Expr_{Atoms}$.

Intuitively, for a given execution, an EHE encodes the conditions to be in a state at a given timestamp as an expression in $Expr_{Atoms}$. $\mathcal{I}(t, q)$ is an expression used to track whether the data structure automaton is in state q at t, i.e., $\mathcal{I}(t, q)$ holds iff the automaton is in state q at timestamp t. We begin by defining the EHE at timestamp $t = 0$ which indicates the initial state of the execution. For a given automaton with an initial state q_0, we know that we are indeed in the initial state at $t = 0$. As such, the initial EHE for the beginning of the execution is the function $[0 \mapsto q_0 \mapsto \top]$. For future timestamps, the EHE is extended inductively based on reachable states.

[9]We note that in the case of RV, traces are typically finite.

Table 1. A tabular representation of $\mathcal{I}^2$.

t	q	e
0	q_0	$\top$
1	q_0	$\neg\langle 1, \mathrm{s}\rangle$
	q_1	$\langle 1, \mathrm{s}\rangle$
2	q_0	$(\neg\langle 1, \mathrm{s}\rangle \wedge \neg\langle 2, \mathrm{s}\rangle) \vee (\langle 1, \mathrm{s}\rangle \wedge \neg\langle 2, \mathrm{s}\rangle)$
	q_1	$(\langle 1, \mathrm{s}\rangle \wedge \langle 2, \mathrm{s}\rangle \wedge \langle 2, \ell\rangle) \vee (\neg\langle 1, \mathrm{s}\rangle \wedge \langle 2, \mathrm{s}\rangle)$
	q_2	$\langle 1, \mathrm{s}\rangle \wedge \langle 2, \mathrm{s}\rangle \wedge \neg\langle 2, \ell\rangle$

Definition 4.5 (Constructing an EHE*).* An EHE encoding the execution till timestamp t, noted $\mathcal{I}^t$ is constructed inductively using function $\mathrm{mov} : (\mathbb{N} \times Q \to Expr_{Atoms}) \times \mathbb{N} \times \mathbb{N} \to (\mathbb{N} \times Q \to Expr)$

$$\mathcal{I}^t \stackrel{\text{def}}{=} \mathrm{mov}([0 \mapsto q_0 \mapsto \top], 0, t)$$
$$\mathrm{mov}(\mathcal{I}, t_s, t_e) \stackrel{\text{def}}{=} \begin{cases} \mathrm{mov}(\mathcal{I}', t_s + 1, t_e) & \text{if } t_s < t_e, \\ \mathcal{I} & \text{otherwise,} \end{cases}$$

with $\mathcal{I}' = \mathcal{I} \dagger_\vee \biguplus_{q' \in \mathrm{next}(\mathcal{I}, t_s)}^{\vee} \{t_s + 1 \mapsto q' \mapsto \mathrm{to}(\mathcal{I}, t_s, q', \mathrm{ts}_{t_s+1})\}$, and

$$\mathrm{to}(\mathcal{I}, t, q', \mathrm{enc}) \stackrel{\text{def}}{=} \bigvee_{\{\langle q, e'\rangle \mid \delta(q,e')=q'\}} (\mathcal{I}(t, q) \wedge \mathrm{enc}(e'))$$
$$\mathrm{next}(\mathcal{I}, t) \stackrel{\text{def}}{=} \{q' \in Q \mid \exists \langle t, q\rangle \in \mathrm{dom}(\mathcal{I}), \exists e \in Expr_{AP} : \delta(q, e) = q'\}.$$

The automaton is in the initial state at $t = 0$. We start building up $\mathcal{I}$ with the initial state and associating it with expression $\top$: $[0 \mapsto q_0 \mapsto \top]$. Then, for a given timestamp t, we use function next to check the next set of reachable states in the automaton (at $t + 1$) by looking at the outgoing transitions for all states in $\mathcal{I}$ at t (i.e., we find a state q' such that $\exists \langle t, q\rangle \in \mathrm{dom}(\mathcal{I}), \exists e \in Expr_{AP} : \delta(q, e) = q'$).

We now build the necessary expression to reach a state q' from multiple states by disjoining the transition labels using $\mathrm{to}(\mathcal{I}, t, q', \mathrm{enc})$, as it suffices to take only one such path to reach q'. Since the label consists of expressions in $Expr_{AP}$ we use an encoder (enc) to get an expression in $Expr_{Atoms}$. If an expression $\mathcal{I}(t, q)$ encodes the condition to reach q at t, and q' is reachable from q at $t + 1$ using the condition e', then it suffices to compute the conjunction.

Finally, $\mathcal{I}'$ is obtained by considering the next states and merging all their expressions with $\mathcal{I}$: $\mathcal{I}' = \mathcal{I} \dagger_\vee \biguplus_{q' \in \mathrm{next}(\mathcal{I}, t_s)}^{\vee} \{t_s + 1 \mapsto q' \mapsto \mathrm{to}(\mathcal{I}, t_s, q', \mathrm{ts}_{t_s+1})\}$. We recall from Section 3.1 that operator $\dagger_\vee$ performs the disjunction between entries, while operator $\biguplus^{\vee}$ on EHE adds expressions for given timestamps and states that are not present, and merges multiple EHEs row by row using disjunction when the entry exists. As such, an EHE is assembled for $t + 1$ by combining all expressions for reachable states at $t + 1$ using $\biguplus^{\vee}$. The assembled EHE for $t + 1$ is then combined with the EHE for t ($\mathcal{I}$) using $\dagger_\vee$, to form the EHE that contains both ($\mathcal{I}'$). We use the notation $\mathrm{rounds}(\mathcal{I})$, to denote all the timestamps that the EHE encodes, i.e., $\mathrm{rounds}(\mathcal{I}) = \{t \in \mathbb{N} \mid \langle t, q\rangle \in \mathrm{dom}(\mathcal{I})\}$. Similarly to automata notation, if multiple EHEs are present, we use a label in the subscript to identify them and their respective operations ($\mathcal{I}_\ell$ denotes the EHE of $\mathcal{A}_\ell$).

Example 4.6 (Constructing an EHE*).* We encode the execution of the automaton presented in Example 4.3. For this example, we use the encoder ts_n which appends timestamp n to an atomic proposition. We have $\mathcal{I}^0 = [0 \mapsto q_0 \mapsto \top]$. From q_0, it is possible to go to q_0 or q_1, therefore $\text{next}(\mathcal{I}^0, 0) = \{q_0, q_1\}$. To stay at q_0 at $t = 1$, we must be at q_0 at $t = 0$, and have : $\text{to}(\mathcal{I}^0, 0, q_0, \text{ts}_1) = \mathcal{I}^0(0, q_0) \wedge \neg\langle 1, \text{s}\rangle$. To move to q_1 at $t = 1$, we must be at q_0 at $t = 0$. The following condition must hold: $\text{to}(\mathcal{I}^0, 0, q_1, \text{ts}_1) = \mathcal{I}^0(0, q_0) \wedge \langle 1, \text{s}\rangle = \langle 1, \text{s}\rangle$. The encoding up to timestamp $t = 2$ is obtained with $\mathcal{I}^2 = \text{mov}(\mathcal{I}^0, 0, 2)$ and is shown in Table 1. We notice that when a state can be reached from multiple states, their expressions are disjoined. For instance, to reach q_0 at $t = 2$, we can either have stayed at q_0 at $t = 1$ and taken the loop transition or have moved to q_1, then taken the transition back to q_0 ($\neg\langle 2, \text{s}\rangle$).

Constructing an EHE with function mov is done only through merges using operator $\dagger_\vee$. By creating an arbitrary order on the states (noted $\prec^Q$) in the specification automaton, we can then compare any two entries of the EHE. We can enumerate the states of the EHE $\mathcal{I}$ as a set of tuples: $\{\langle t, q, e\rangle \mid \mathcal{I}(t, q) = e\}$. We are then able to compare any such two tuples: $\langle t, q, e\rangle \prec \langle t', q', e'\rangle$. To do so, we first check the order of timestamps (i.e. if $t \prec t'$ then $\langle t, q, e\rangle \prec \langle t', q', e'\rangle$). When the timestamps are the same ($t = t'$) then we use the order $\prec^Q$ to determine the order of the tuples. When both the timestamp and the state are the same, then merging occurs. By merging with $\dagger_\vee$ (Section 3.1) the entries not found in both are added using set union, and entries with the same timestamp and state (t, q) are disjoined ($\vee$), which is idempotent, associative, and commutative. As such, the EHE is a CvRDT.

Corollary 4.7. *An* EHE *constructed with* mov *and merged with* $\dagger_\vee$ *is a CvRDT.*

By constructing the EHE, we have for each timestamp t and each state q in the EHE an expression. Using information from the execution stored in a memory $\mathcal{M}$, if $\text{eval}(\mathcal{I}(t, q), \mathcal{M})$ is $\top$, then we know that the automaton is indeed in state q at timestamp t. Given a memory $\mathcal{M}$ which stores atoms, function sel determines if a state is reached at a timestamp t. If the memory does not contain enough information to evaluate the expressions, then the state is undef. The state q at timestamp t with a memory $\mathcal{M}$ is determined by:

$$\text{sel}(\mathcal{I}, \mathcal{M}, t) = \begin{cases} q & \text{if } \exists q \in Q : \text{eval}(\mathcal{I}(t, q), \mathcal{M}) = \top, \\ \text{undef} & \text{otherwise.} \end{cases}$$

We note that q such that $\text{eval}(\mathcal{I}(t, q), \mathcal{M}) = \top$ is unique. Since we are encoding deterministic automata, we recall from Remark 1 that when a state q is reached at a timestamp t resulting from an execution, no other state can be reached at t for the same execution. Moreover, the EHE construction using operation mov and encoder ts preserves determinism.

Proposition 4.8 (Deterministic EHE). *Given an* EHE $\mathcal{I}$ *constructed with operation* mov *using encoder* ts*, we have:*

$$\forall t \in \text{rounds}(\mathcal{I}), \forall \mathcal{M} \in \mathit{Mem}, \exists q \in Q : \\ \text{eval}(\mathcal{I}(t, q), \mathcal{M}) = \top \implies \forall q' \in Q \setminus \{q\} : \text{eval}(\mathcal{I}(t, q'), \mathcal{M}) \neq \top.$$

Determinism is preserved since, by using encoder ts, we only change an expression to add the timestamp. By construction, when there exists a state q s.t. $\text{eval}(\mathcal{I}(t, q), \mathcal{M}) = \top$, such a state is unique, since the EHE is built using a deterministic automaton. The full proof is in Appendix A.

Function verAt is a short-hand to retrieve the verdict at a given timestamp t:

$$\text{verAt}(\mathcal{I}, \mathcal{M}, t) = \begin{cases} \text{ver}(q) & \text{if } \exists q \in Q : q = \text{sel}(\mathcal{I}, \mathcal{M}, t), \\ ? & \text{otherwise.} \end{cases}$$

Example 4.9 (Monitoring with EHE*).* Consider the constructed EHE from Example 4.6 shown in Table 1. Let us consider the global trace from Example 3.14: $evt_0 \cdot evt_1$, with $evt_0 = \{\langle \mathrm{s}, \top\rangle, \langle \ell, \top\rangle\}$ and $evt_1 = \{\langle \mathrm{s}, \top\rangle, \langle \ell, \bot\rangle\}$. We create a memory with the events of the two timestamps. Let $\mathcal{M} = \mathrm{memc}(evt_0, \mathrm{ts}_1) \dagger_2 \mathrm{memc}(evt_1, \mathrm{ts}_2) = [\langle 1, \mathrm{s}\rangle \mapsto \top, \langle 1, \ell\rangle \mapsto \top, \langle 2, \mathrm{s}\rangle \mapsto \top, \langle 2, \ell\rangle \mapsto \bot,]$. It is possible to infer the state of the automaton at $t = 2$ using $\mathcal{I}^2 = \mathrm{mov}([0 \mapsto q_0 \mapsto \top], 0, 2)$ by using $\mathrm{sel}(\mathcal{I}^2, \mathcal{M}, 2)$, we evaluate:

$$\begin{aligned}
\mathrm{eval}(\mathcal{I}^2(2, q_0), \mathcal{M}) &= (\neg\langle 1, \mathrm{s}\rangle \wedge \neg\langle 2, \mathrm{s}\rangle) \vee (\langle 1, \mathrm{s}\rangle \wedge \neg\langle 2, \mathrm{s}\rangle) &&= \bot \\
\mathrm{eval}(\mathcal{I}^2(2, q_1), \mathcal{M}) &= (\langle 1, \mathrm{s}\rangle \wedge \langle 2, \mathrm{s}\rangle \wedge \langle 2, \ell\rangle) \vee (\neg\langle 1, \mathrm{s}\rangle \wedge \langle 2, \mathrm{s}\rangle)) &&= \bot \\
\mathrm{eval}(\mathcal{I}^2(2, q_2), \mathcal{M}) &= \langle 1, \mathrm{s}\rangle \wedge \langle 2, \mathrm{s}\rangle \wedge \neg\langle 2, \ell\rangle &&= \top
\end{aligned}$$

We find that q_2 is the selected state, with verdict $\mathrm{ver}(q_2) = \bot$.

While the construction of an EHE preserves the determinism found in the automaton, an important property is in ensuring that the EHE encodes correctly the execution of the automaton.

Proposition 4.10 (Soundness). *Given a decentralized trace* tr *of length n, we reconstruct the global trace* $\rho(\mathrm{tr}) = evt_1 \cdot \ldots \cdot evt_n$*, we have:* $\Delta^*(q_0, \rho(\mathrm{tr})) = \mathrm{sel}(\mathcal{I}^n, \mathcal{M}^n, n)$*, with:*

$$\begin{aligned}
\mathcal{I}^n &= \mathrm{mov}([0 \mapsto q_0 \mapsto \top], 0, n), \textit{ and} \\
\mathcal{M}^n &= \biguplus^2_{t \in [1,n]} \{\mathrm{memc}(evt_t, \mathrm{ts}_t)\}.
\end{aligned}$$

EHE is sound with respect to the specification automaton; both the automaton and EHE will indicate the same state reached with a given trace. Thus, the verdict is the same as it would be in the automaton. The proof is by induction on the reconstructed global trace ($|\rho(\mathrm{tr})|$).

Proof sketch. We first establish that both the EHE and the automaton memories evaluate two similar expressions modulo encoding to the same result. That is, for the given length i, the generated memories at $i + 1$ with encodings idt and ts_{i+1} yield similar evaluations for the same expression e. Then, starting from the same state q_i reached at length i, we assume $\Delta^*(q_0, evt_1 \cdot \ldots \cdot evt_i) = \mathrm{sel}(\mathcal{I}^i, \mathcal{M}^i, i) = q_i$ holds. We prove that it holds at $i + 1$, by building the expression (for each encoding) to reach state q_{i+1} at $i + 1$, and showing that the generated expression is the only expression that evaluates to $\top$. As such, we determine that both evaluations point to q_{i+1} being the next state. The full proof is in Appendix A.

4.3 Decentralized Monitoring with EHE

EHE provides interesting properties for decentralized monitoring. Two (or more) components sharing EHEs and merging them will be able to infer the same execution history of the automaton. That is, components will be able to aggregate the information of various EHEs, and are able to determine the reached state, if possible, or that no state was reached. Merging two EHEs of the same automaton with $\dagger_\vee$ allows us to aggregate information from two partial histories.

However, two EHEs for the same automaton contain the same expression if constructed with mov. To incorporate the memory in an EHE, we generate a new EHE that contains the rewritten and simplified expressions for each entry. To do so we define function inc to apply to a whole EHE and a memory to generate a new EHE: $\mathrm{inc}(\mathcal{I}, \mathcal{M}) = \biguplus^2_{\langle t,q\rangle \in \mathrm{dom}(\mathcal{I})} \{[\langle t, q\rangle \mapsto \mathrm{simplify}(\mathrm{rw}(\mathcal{I}(t, q), \mathcal{M}))]\}$. We note, that for a given $\mathcal{I}$ and $\mathcal{M}$, $\mathrm{inc}(\mathcal{I}, \mathcal{M})$ maintains the invariant of Proposition 4.8. We are simplifying expressions or rewriting atoms with their values in the memory which is what eval already does for each entry in the EHE. That is, $\mathrm{inc}(\mathcal{I}, \mathcal{M})$ is a valid representation of the same deterministic and complete automaton as $\mathcal{I}$. However, $\mathrm{inc}(\mathcal{I}, \mathcal{M})$ incorporates information from memory $\mathcal{M}$ in addition.

Proposition 4.11 (Memory obsolescence).

$$\forall \langle t, q\rangle \in \mathrm{dom}(\mathrm{inc}(\mathcal{I}, \mathcal{M})) : \mathrm{eval}(\mathcal{I}(t,q), \mathcal{M}) \Leftrightarrow \mathrm{eval}(\mathrm{inc}(\mathcal{I}, \mathcal{M})(t,q), []).$$

Proof. Follows directly by construction of inc and the definition of eval (which uses functions simplify and rw). □

Proposition 4.11 ensures that it is possible to directly incorporate a memory in an EHE, making the memory no longer necessary. This is useful for algorithms that communicate the EHE, as they do not need to also communicate the memory.

By rewriting the expressions, the EHEs of two different monitors receiving different observations contain different expressions. However, since they still encode the same automaton, and observations do not conflict, merging with $\dagger_\vee$ shares useful information.

Corollary 4.12. *Given an* EHE $\mathcal{I}$ *constructed using function* mov, *and two memories* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *that do not observe conflicting observations*[10], *the two* EHE*s* $\mathcal{I}_1 = \mathrm{inc}(\mathcal{I}, \mathcal{M}_1)$ *and* $\mathcal{I}_2 = \mathrm{inc}(\mathcal{I}, \mathcal{M}_2)$ *have the following properties* $\forall \langle t, q\rangle \in \mathrm{dom}(\mathcal{I}')$*:*

1) $\mathcal{I}' = \mathcal{I}_1 \dagger_\vee \mathcal{I}_2$ *is deterministic (Proposition 4.8);*

2) $\mathrm{eval}(\mathcal{I}'(t,q), []) \implies \mathrm{eval}(\mathcal{I}(t,q), \mathcal{M}_1 \dagger_2 \mathcal{M}_2)$*;*

3) $\mathrm{eval}(\mathcal{I}'(t,q), []) = \top \implies \mathrm{eval}(\mathcal{I}'(t,q), \mathcal{M}_1) = \top \wedge \mathrm{eval}(\mathcal{I}'(t,q), \mathcal{M}_2) = \top$*;*

4) $\mathrm{eval}(\mathcal{I}'(t,q), []) = \top \implies \mathrm{eval}(\mathcal{I}_1(t,q), \mathcal{M}_1) \neq \bot \wedge \mathrm{eval}(\mathcal{I}_2(t,q), \mathcal{M}_2) \neq \bot$.

The first property ensures that merging two EHEs that incorporate memories are still indeed representing a deterministic and complete automaton, this follows from Proposition 4.8 and Proposition 4.11. Since operation $\dagger_\vee$ disjoins the two expressions, and since the two expressions come from EHEs that each maintain the property, the additional disjunction will not affect the outcome of eval. The second property extends Proposition 4.11 to the merging of EHE with incorporated memories. It follows directly from Proposition 4.11, and the assumptions that the memories have no conflicts. The third property adds a stronger condition. It states that merging two EHEs with incorporated memories results in an EHE that evaluates to true, cannot evaluate to anything else with the two different memories. This follows from the second property and the fact that the memories do not have conflicting observations. Finally, the fourth property ensures that merging an EHE with an entry that evaluates to $\bot$ does not result in an entry that evaluates to $\top$. That is, if an EHE has already determined that a state is not reachable, merging it with another EHE does not result in the state being reachable. This ensures the consistency when sharing information. This property follows from the merging operator $\dagger_\vee$ which uses $\vee$ to merge entries in two EHEs. We recall that an entry in $\langle t, q\rangle \in \mathrm{dom}(\mathcal{I}')$ is constructed as: $\mathrm{eval}(\mathcal{I}_1(t,q), \mathcal{M}_1) \vee \mathrm{eval}(\mathcal{I}_2(t,q), \mathcal{M}_2)$. For $\mathrm{eval}(\mathcal{I}'(t,q), [])$ to be $\top$, either $\mathrm{eval}(\mathcal{I}_1(t,q), \mathcal{M}_1)$ or $\mathrm{eval}(\mathcal{I}_2(t,q), \mathcal{M}_2)$ has to be $\top$, if one is already $\bot$, then the other has to be $\top$. This leads to a contradiction, since both $\mathcal{I}_1$ and $\mathcal{I}_2$ encode the same deterministic automaton, as such, the automaton cannot be in two states at once.

Example 4.13 (Reconciling information). We consider the specification presented in Example 2.1, and the decentralized trace and two components: lswitch and bulb presented in Example 3.14. We recall the trace $\mathrm{tr} = [1 \mapsto \mathrm{lswitch} \mapsto \{\langle \mathrm{s}, \top\rangle\}, 1 \mapsto \mathrm{bulb} \mapsto \{\langle \ell, \top\rangle\}, 2 \mapsto \mathrm{lswitch} \mapsto \{\langle \mathrm{s}, \top\rangle\}, 2 \mapsto \mathrm{bulb} \mapsto \{\langle \ell, \bot\rangle\}]$. Furthermore, we associate respectively two monitors m_0 and m_1 with components lswitch and bulb. We focus on the timestamp at $t = 2$. The monitors can observe the propositions s and ℓ respectively and use one EHE each: $\mathcal{I}_0^2$ and $\mathcal{I}_1^2$ respectively. Their memories are respectively $\mathcal{M}_0^2 = [\langle 1, \mathrm{s}\rangle \mapsto \top, \langle 1, \mathrm{s}\rangle \mapsto \top]$ and $\mathcal{M}_1^2 = [\langle 1, \ell\rangle \mapsto \top, \langle 2, \ell\rangle \mapsto \bot]$. Table 2 shows

[10] That is, they do not associate with the same atom different truth values. This is ensured by our assumption that the system and monitors do not send wrong information.

Table 2. Reconciling information by combining EHEs. $\mathcal{I}^2$ indicates the non-rewritten EHE. The columns $[\mathcal{M}_0^2]$ and $[\mathcal{M}_1^2]$, the result of performing eval on the EHE $\dagger_\vee^2$ using memories $\mathcal{M}_0^2$ and $\mathcal{M}_1^2$ respectively. A dash (-) indicates the expression is the same as $\mathcal{I}^2$.

t	q	$\mathcal{I}^2$	$\mathcal{I}_0^2$	$\mathcal{I}_1^2$	$\dagger_\vee^2$	$[\mathcal{M}_0^2]$	$[\mathcal{M}_1^2]$
0	q_0	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
1	q_0	$\neg\langle 1, s\rangle$	$\bot$	-	-	$\bot$	?
	q_1	$\langle 1, s\rangle$	$\top$	-	$\top$	$\top$	$\top$
2	q_0	$\neg\langle 1, s\rangle \wedge \neg\langle 2, s\rangle \vee \langle 1, s\rangle \wedge \neg\langle 2, s\rangle$	$\bot$	-	-	$\bot$	?
	q_1	$\langle 1, s\rangle \wedge \langle 2, s\rangle \wedge \langle 2, \ell\rangle \vee \neg\langle 1, s\rangle \wedge \langle 2, s\rangle$	$\langle 2, \ell\rangle$	$\neg\langle 1, s\rangle \wedge \langle 2, s\rangle$	$\langle 2, \ell\rangle \vee (\neg\langle 1, s\rangle \wedge \langle 2, s\rangle)$	?	?
	q_2	$\langle 1, s\rangle \wedge \langle 2, s\rangle \wedge \neg\langle 2, \ell\rangle$	$\neg\langle 2, \ell\rangle$	$\langle 1, s\rangle \wedge \langle 2, s\rangle$	$\neg\langle 2, \ell\rangle \vee (\langle 1, s\rangle \wedge \langle 2, s\rangle)$	$\top$	$\top$

the EHEs (where $\mathcal{I}^2$ denotes the non-rewritten EHE). The columns $[\mathcal{M}_0^2]$ and $[\mathcal{M}_1^2]$ show the result of performing eval on the EHE $\dagger_\vee^2$ using memories $\mathcal{M}_0^2$ and $\mathcal{M}_1^2$ respectively.

Constructing the EHE $\mathcal{I}^2$ follows similarly from Example 4.6. We show the rewriting for both $\mathcal{I}_0^2$ and $\mathcal{I}_1^2$ respectively in the next two columns. Then, we show the result of combining the rewrites using $\dagger_\vee$. We notice initially that since s is $\bot$, m_0 could evaluate $\langle 1, s\rangle = \top$ and know that the automaton is in state q_1. However, for m_1, this is not possible until the expressions are combined. By evaluating the combination, m_1 determines that the automaton is in state q_0 at $t = 1$. We see at $t = 2$ for both q_1 and q_2 the expression resulting from combining the EHE is much weaker than the one present in each of the individual EHEs. After evaluating with the local memory, both monitors determine that the automaton is in state q_2.

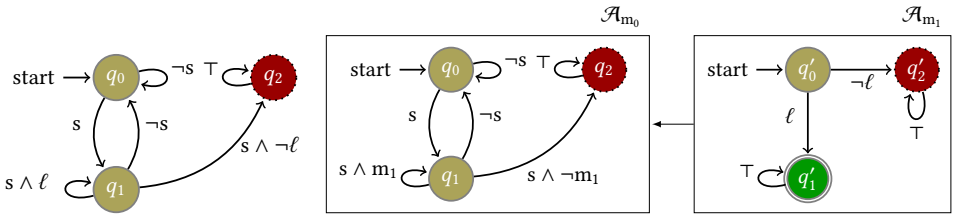
In this case, we are only looking for expressions that evaluate to $\top$. We notice that monitor m_0 can determine that q_0 is not reachable (since $\neg\langle 1, s\rangle = \bot$) while m_1 cannot, as the expression $\neg\langle 1, s\rangle$ cannot yet be evaluated to a final verdict, and thus the combination evaluates to ?. This does not affect the outcome, as we are only looking for one expression that evaluates to $\top$, since both $\mathcal{I}_0^2$ and $\mathcal{I}_1^2$ are encoding the same execution. In the future, we would like to also propagate the information about the non-reachable states by tweaking the combination of EHEs.

5 DECENTRALIZED SPECIFICATIONS

In this section, we shift the focus to a specification that is decentralized. A set of automata represent various requirements (and dependencies) for different components of a system. In this section, we define the notion of a decentralized specification and its semantics, and in Section 6, we define various properties on such specifications.

5.1 Decentralizing a Specification

We recall that a decentralized system consists of a set of components C. To decentralize the specification, instead of having one automaton, we have a set of specification automata (Definition 4.1) $\text{Mons} = \{\mathcal{A}_\ell \mid \ell \in AP_{\text{mons}}\}$, where $AP_{\text{mons}} \subseteq AP$ is a set of monitor labels. We refer to these automata as *monitors*. To each monitor, we associate a component using a function $\mathcal{L} : \text{Mons} \to C$. However, the transition labels of a monitor $\text{mon} \in \text{Mons}$ are expressions restricted to either observations local to the component the monitor is attached to (i.e., $\mathcal{L}(\text{mon})$), or references to other monitors. Transitions are labeled over $AP_{\text{mons}} \setminus \{\text{mon}\} \cup \{ap \in AP \setminus AP_{\text{mons}} \mid \text{lu}(ap) = \mathcal{L}(\text{mon})\}$. This ensures that the monitor is labeled with observations it can locally observe or depend on other



(a) Centralized. (b) Decentralized.

Fig. 3. Monitor(s) for the centralized and decentralized light switch and bulb specification presented in Example 2.1. The verdicts associated with the states are ⊥: dotted red , ⊤: double green, and ?: single yellow.

monitors. To evaluate a trace as one would on a centralized specification, we require one of the monitors to be a starting point, we refer to that monitor as the *root monitor* (rt ∈ Mons).

Definition 5.1 (Decentralized specification). A decentralized specification is a tuple $\langle AP_{\text{mons}}, \text{Mons}, C, \mathcal{L}, \text{rt}\rangle$.

We note that a centralized specification is a special case of a decentralized specification, with one component (global system, sys), and one monitor (g) attached to the sole component, i.e. $\langle\{\text{g}\}, \{\mathcal{A}_\text{g}\}, \{\text{sys}\}, [\mathcal{A}_\text{g} \mapsto \text{sys}], \mathcal{A}_\text{g}\rangle$.

As automata expressions now include references to monitors, we first define function dep : $Expr_{AP} \to$ Mons, which determines monitor dependencies. Then, we define the semantics of evaluating (decentralized) specifications with references.

Definition 5.2 (Monitor dependency). The set of monitor dependencies in an expression e is obtained by function dep : $Expr_{AP} \to$ Mons, defined as[11]: dep(*e*) = `match` *e* `with`:

$| \; id \in AP_{\text{mons}} \to \{\mathcal{A}_{id}\}$ $\quad | \; e_1 \wedge e_2 \to \text{dep}(e_1) \cup \text{dep}(e_2)$ $\quad | \; id \in AP \setminus AP_{\text{mons}} \to \emptyset$

$| \; \neg e \to \text{dep}(e)$ $\quad | \; e_1 \vee e_2 \to \text{dep}(e_1) \cup \text{dep}(e_2)$

Function dep finds all monitors referenced by expression e, by syntactically traversing it.

Example 5.3 (Decentralized specification). Figure 3b shows a decentralized light switch and bulb specification corresponding to the centralized specification in Example 2.1 (shown in Figure 3a for side-by-side comparison). We recall from Example 3.14 that the system consists of two components the light switch and bulb, labeled lswitch and bulb, respectively. We associated the components lswitch and bulb with the monitors $\mathcal{A}_{\text{m}_0}$ and $\mathcal{A}_{\text{m}_1}$, respectively. We use $\mathcal{A}_{\text{m}_0}$ as the root monitor for the decentralized specification. We consider the two atomic propositions s and ℓ can only be observed by component lswitch and bulb respectively. $\mathcal{A}_{\text{m}_0}$ depends on the verdict from m_1 and only observations local to lswitch, while $\mathcal{A}_{\text{m}_1}$ is only labeled with observations local to bulb. Given the expression $\text{s} \wedge \text{m}_1$, we have $\text{dep}(\text{s} \wedge \text{m}_1) = \{\mathcal{A}_{\text{m}_1}\}$.

5.2 Semantics of a Decentralized Specification

The transition function of the decentralized specification is similar to the centralized automaton with the exception of monitor ids.

Definition 5.4 (Semantics of a decentralized specification). Consider the root monitor $\mathcal{A}_{\text{rt}}$ and a decentralized trace tr with index $i \in [1, |\text{tr}|]$ representing the timestamps. Monitoring tr starting

[11] We note that this definition can be trivially extended to any encoding of such expressions that contains the monitor id.

from $\mathcal{A}_{\text{rt}}$ emits the verdict $\text{ver}_{\text{rt}}(\Delta'^{*}_{\text{rt}}(q_{\text{rt}_0}, \text{tr}, 1))$ where for a given monitor label ℓ:

$$\Delta'^{*}_{\ell}(q, \text{tr}, i) = \begin{cases} \Delta'^{*}_{\ell}(\Delta'_{\ell}(q, \text{tr}, i), \text{tr}, i+1) & \text{if } i < |\text{tr}| \\ \Delta'_{\ell}(q, \text{tr}, i) & \text{otherwise} \end{cases}$$

$$\Delta'_{\ell}(q, \text{tr}, i) = \begin{cases} q' & \text{if } \text{tr}(i, \mathcal{L}(\mathcal{A}_\ell)) \neq \emptyset \wedge \exists e \in Expr_{AP} : \delta_\ell(q, e) = q' \wedge \text{eval}(\text{idt}(e), \mathcal{M}) = \top \\ q & \text{otherwise} \end{cases}$$

$$\text{where} \quad \mathcal{M} = \text{memc}(\text{tr}(i, \mathcal{L}(\mathcal{A}_\ell)), \text{idt}) \dagger_2 \biguplus^{2}_{\mathcal{A}_{\ell'} \in \text{dep}(e)} \{[\ell' \mapsto \text{ver}_{\ell'}(q_{\ell'_f})]\}$$

$$\text{and} \quad q_{\ell'_f} = \Delta'^{*}_{\ell'}(q_{\ell'_0}, \text{tr}, i)$$

For a monitor $\mathcal{A}_\ell$, we determine the new state of the automaton starting at $q \in Q_\ell$, and running the trace tr from timestamp i to timestamp t by applying $\Delta'^{*}_{\ell}(q, \text{tr}, i)$. To do so, we evaluate one transition at a time using Δ'_ℓ as would Δ^{*}_{ℓ} with Δ_ℓ (see Definition 4.2). To evaluate Δ'_ℓ at any state $q' \in Q_\ell$, we need to evaluate the expressions so as to determine the next state q''. The expressions contain atomic propositions and monitor ids. For atomic propositions, the memory is constructed using $\text{memc}(\text{tr}(i, \mathcal{L}(\mathcal{A}_\ell)), \text{idt})$ which is based on the event with observations local to the component the monitor is attached to (i.e., $\mathcal{L}(\mathcal{A}_\ell)$). However, for monitor ids, the memory represents the verdicts of the monitors. To evaluate each reference ℓ' in the expression, the remainder of the trace starting from the current event timestamp i is evaluated recursively on the automaton $\mathcal{A}_{\ell'}$ from the initial state $q_{\ell'_0} \in \mathcal{A}_{\ell'}$. Then, the verdict of the monitor is associated with ℓ' in the memory.

Example 5.5 (Monitoring of a decentralized specification). We consider the decentralized specification from Example 5.3. We have the monitors $\mathcal{A}_{\text{m}_0}$ (root) and $\mathcal{A}_{\text{m}_1}$ associated to components lswitch and bulb respectively. Furthermore, we consider the decentralized trace from Example 3.14: $\text{tr} = [1 \mapsto \text{lswitch} \mapsto \{\langle \text{s}, \top \rangle\}, 1 \mapsto \text{bulb} \mapsto \{\langle \ell, \top \rangle\}, 2 \mapsto \text{lswitch} \mapsto \{\langle \text{s}, \top \rangle\}, 2 \mapsto \text{bulb} \mapsto \{\langle \ell, \bot \rangle\}]$.

To evaluate tr on $\mathcal{A}_{\text{m}_0}$ (from Figure 3b), we use $\Delta'^{*}_{\text{m}_0}(q_0, \text{tr}, 1)$. To do so, we first evaluate $\Delta'_{\text{m}_0}(q_0, \text{tr}, 1)$. In this case, the expressions only depend on the atomic proposition s, which does not depend on any other monitor. We have $\mathcal{M}^1_{\text{m}_0} = \text{memc}(\langle \text{s}, \top \rangle, \text{idt}) = [\text{s} \mapsto \top]$, and $\text{eval}(\text{s}, \mathcal{M}^1_{\text{m}_0}) = \top$. Thus, we obtain $\Delta'_{\text{m}_0}(q_0, \text{tr}, 1) = q_1$.

In q_1 at $t = 2$, we now evaluate $\Delta'_{\text{m}_0}(q_1, \text{tr}, 2)$. Transitions from q_1 are labeled with expressions that depend on m_1. Therefore, we evaluate the decentralized trace on $\mathcal{A}_{\text{m}_1}$ starting at $t = 2$ by evaluating $\Delta'^{*}_{\text{m}_1}(q'_0, \text{tr}, 2)$. We start by evaluating $\Delta'_{\text{m}_1}(q'_0, \text{tr}, 2)$. We have $\mathcal{M}^2_{\text{m}_1} = \text{memc}(\langle \ell, \bot \rangle, \text{idt}) = [\ell \mapsto \bot]$, and $\text{eval}(\neg \ell, \mathcal{M}^2_{\text{m}_1}) = \top$. Thus, we obtain $\Delta'_{\text{m}_1}(q'_0, \text{tr}, 2) = q'_2$ labeled by the verdict $\bot$. Having reached a final verdict for m_1, we can construct the memory for m_0. We have $\mathcal{M}^2_{\text{m}_0} = \text{memc}(\langle \text{s}, \top \rangle, \text{idt}) \dagger_2 [\text{m}_1 \mapsto \bot] = [\text{s} \mapsto \top, \text{m}_1 \mapsto \bot]$. Knowing that $\text{eval}(\text{s} \wedge \neg \text{m}_1, \mathcal{M}^2_{\text{m}_0}) = \top$, we conclude that the next state is $\Delta'_{\text{m}_0}(q_1, \text{tr}, 2) = q_2$. Since q_2 is labeled by verdict $\bot$, the monitoring concludes and we detect a violation of the specification.

6 PROPERTIES FOR DECENTRALIZED SPECIFICATIONS

A key advantage of using decentralized specifications is to make the association of monitors with components explicit. Since monitors have been explicitly modeled as a set of automata with dependencies between each other, we can now determine properties on decentralized specifications. In this section, we revisit the concept of *monitorability*, characterize it for automata, define it for decentralized specifications, and describe an algorithm for deciding monitorability. Furthermore, we explore *compatibility*, that is the ability of a decentralized specification to be deployed on a given architecture.

Fig. 4. A trivial non-monitorable specification.

6.1 Decentralized Monitorability

An important notion to consider when dealing with runtime verification is that of monitorability [27, 44]. In brief, monitorability of a specification determines whether or not an RV technique is applicable to a specification. That is, a monitor synthesized for a non-monitorable specification is unable to check if the execution complies or violates the specification for all possible traces. Consider the automaton shown in Figure 4, one could see that there is no state labeled with a final verdict. In this case, we can trivially see that no trace allows us to reach a final verdict. We also notice similar behavior when monitoring LTL expressions with the pattern $\mathbf{GF}(p)$ with p is an atomic proposition. The LTL expression requires that at all times $\mathbf{F}(p)$ holds $\top$, while $\mathbf{F}(p)$ requires that p eventually holds $\top$. As such, at any given point of time, we are unable to determine a verdict, since if p is $\bot$ at the current timestamp, it can still be $\top$ at a future timestamp, and thus $\mathbf{F}(p)$ will be $\top$ for the current timestamp. And if $\mathbf{F}(p)$ is $\top$ at the current timestamp, the $\mathbf{G}$ requires that it be $\top$ for all timestamps, so in the future there could exist a timestamp which falsifies it. Consequently, when monitoring such an expression, a monitor will always output ?, as it cannot determine a verdict for any given timestamp. In this section, we first characterize monitorability in terms of automata and EHE for both centralized and decentralized specifications. Then, we provide an effective algorithm to determine monitorability.

6.1.1 *Characterizing Monitorability.*

Centralized monitorability of properties. Monitorability in the sense of [44] is defined on traces. A property is monitorable if for all finite traces t (a sequence of events) in the set of all (possibly infinite) traces, there exists a continuation t' such that monitoring $t \cdot t'$ results in a true or false verdict. Informally, it can be seen as whether or not continuing to monitor the property after reading t can still yield a final verdict. We note that this definition deals with all possible traces, it establishes monitorability to be oblivious of the input trace.

Centralized monitorability in automata. We express monitorability to reach "true" or "false" verdict to the notion of reaching a *final verdict*, and associate it with automata. For automata, monitorability can be analyzed in terms of reachability and states.

Definition 6.1 (Monitorability of an automaton). Given a automaton $\mathcal{A} = \langle Q, q_0 \in Q, \delta, \text{ver} \rangle$, a state $q \in Q$ is monitorable (noted monitorable(q)) iff $\text{ver}(q') \in \mathbb{B}_2$ or $\exists q' \in Q$ such that $\text{ver}(q') \in \mathbb{B}_2$ and q' is reachable from q. Automaton $\mathcal{A}$ is said to be monitorable (noted monitorable($\mathcal{A}$)) iff $\forall q \in Q :$ monitorable(q).

Defining monitorability using reachability is consistent with [44]. After reading a finite trace t and reaching q ($q = \Delta^*(q_0, t)$), there exists a continuation t' that leads the automaton to a state q' ($q' = \Delta^*(q, t')$), such that $\text{ver}(q') \in \mathbb{B}_2$. We note that an automaton is monitorable according to this definition iff, in the automaton, all paths from the initial state q_0 lead to a state with a final verdict. As such, it is sufficient to analyze the automaton to determine monitorability irrespective of possible traces (see Section 6.1.2)[12]. We illustrate monitorability of automata in Example 6.2.

Example 6.2 (Centralized monitorability of automata.). Figure 3a illustrates the automaton that expresses the light switch and bulb specification. It is monitorable, as the states q_0, q_1, and q_2 are

[12]The expressions leading to q' must all be also satisfiable. However, satisfiability is guaranteed as our automaton is normalized, see Remark 1.

monitorable. For both q_0 and q_1, it is possible to reach q_2 labeled with the final verdict $\bot$. We note that monitorability is a necessary but not sufficient condition for termination (with a final verdict). An infinite trace consisting of repetitions of the event $\{\neg\langle s, \bot\rangle, \langle \ell, \bot\rangle\}$ never lets the automaton reach q_2. However, monitorability guarantees the possibility of reaching a final verdict. If a state q is not monitorable, we know that it is impossible to reach a final verdict from q, and can abandon monitoring.

Centralized monitorability with EHE. Reachability in automata can be expressed as well using the EHE data structure. A path from a state q to a state q' is expressed as an expression over atoms. We define paths(q, q') to return all possible paths from q to q'.

$$\text{paths}(q, q') = \{e \in Expr_{Atoms} \mid \exists t \in \mathbb{N} : \mathcal{I}^t(t, q') = e \wedge \ \mathcal{I}^t = \text{mov}([0 \mapsto q \mapsto \top], 0, t)\}$$

Each expression is derived similarly as would an execution in the EHE (Definition 4.4). We start from state q and use a logical timestamp starting at 0 incrementing it by 1 for the next reachable state. A state q is monitorable iff $\exists e_f \in \text{paths}(q, q_f)$, such that (1) e_f is satisfiable; (2) $\text{ver}(\text{simplify}(e_f)) \in \mathbb{B}_2$. The first condition ensures that the path is able to lead to the state q_f, as an unsatisfiable path will never evaluate to true. The second condition ensures that the state is labeled by a final verdict. An automaton is thus monitorable iff all its states are monitorable. We note that paths(q, q') can be infinite if the automaton contains cycles, however path expressions could be "compacted" using the pumping lemma. Using EHE we can frame monitorability as a satisfiability problem which can benefit from additional knowledge on the truth values of atomic propositions. For the scope of this paper, we focus on computing monitorability on automata in Section 6.1.2.

Decentralized monitorability. In the decentralized setting, we have a set of monitors Mons. The labels of automata include monitor ids (AP_{mons}). We recall that the evaluation of a reference $\ell \in AP_{\text{mons}}$ consists in running the remainder of the trace on $\mathcal{A}_\ell$ starting from the initial state q_{ℓ_0}. As such, for any dependency on a monitor $\mathcal{A}_\ell$, we know that ℓ evaluates to a final verdict iff monitorable$(\mathcal{A}_\ell)$. We notice that monitorability of decentralized specification is recursive, and relies on the inter-dependencies between the various decentralized specifications. This is straightforward for EHE, since a path is an expression. For a path e_f, the dependent monitors are captured in the set dep(e_f). The additional condition on the path is thus: $\forall \mathcal{A}_\ell \in \text{dep}(e_f) : \ \text{monitorable}(\mathcal{A}_\ell)$.

6.1.2 *Computing Monitorability.*

Centralized specification. We compute the monitorability of a centralized specification $\mathcal{A}$, with respect to a set of final verdicts $\mathbb{B}_2$[13]. We denote monitorability by monitorable$(\mathcal{A}, \mathbb{B}_2)$. In the remainder of the thesis we always use $\mathbb{B}_2$, thus, we write monitorable$(\mathcal{A})$. Computing monitorability consists in checking that all states of the automaton are co-reachable from states with final verdicts. As such, it relies on a traversal of the graph starting from the states that are labeled with final verdicts. To do so, we use a variation of the work-list algorithm. We begin by adding all states labeled by a final verdict to the work list. These states are trivially monitorable. Conversely, any state that leads to a monitorable state is monitorable. As such, for each element in the work list, we add its predecessors to the work list. We maintain a set of marked states (Mark), that is, states that have already been processed, so as to avoid adding them again. This ensures that cycles are properly handled. The algorithm stabilizes when no further states can be processed (i.e., the work list is empty). All marked states (Mark) are therefore monitorable. To check if an automaton is monitorable, we need all of its states to be monitorable. As such we verify that $|\text{Mark}| = |Q|$. The number of edges between any pair of states can be rewritten to be at most 1 (as explained

[13]While we use $\mathbb{B}_2$, this can be extended without loss of generality to an arbitrary set $\mathbb{B}_f$.

in Section 4.1). As such, one has to traverse the graph once, the complexity being linear in the states and edges (i.e., $O(|Q| + |\delta|)$). Hence in the worst case, an automaton forms a complete graph, and we have $\binom{|Q|}{2}$ edges. The worst case complexity is quadratic in the number of states (i.e., $O(|Q| + \frac{1}{2}|Q|(|Q| - 1))$).

Decentralized specifications. In the case of decentralized specifications, the evaluation of paths (using eval) in an automaton depends on other monitors (and thus other automata). To compute monitorability, we first build the monitor dependency set for a given monitor $\mathcal{A}_\ell$ (noted $\mathrm{MDS}(\mathcal{A}_\ell)$) associated with a monitor label ℓ.

$$\mathrm{MDS}(\mathcal{A}_\ell) = \biguplus_{\{e \in Expr_{AP} \mid \exists q, q' \in Q_\ell \colon \delta_\ell(q,e)=q'\}} \mathrm{dep}(e)$$

The monitor dependency list for a monitor contains all the references to other monitors across all paths in the given automaton ($\mathcal{A}_\ell$), by examining all the transitions. It can be obtained by a simple traversal of the automaton.

Second, we construct the monitor dependency graph (MDG), which describes the dependencies between monitors. The monitor dependency graph for a set of monitors Mons is noted $\mathrm{MDG}(\mathrm{Mons}) = \langle \mathrm{Mons}, \mathrm{DE} \rangle$ where DE is the set of edges which denotes the dependency edges between the monitors, defined as: $\mathrm{DE} = \{\langle \mathcal{A}_\ell, \mathcal{A}_{\ell'} \rangle \in \mathrm{Mons} \times \mathrm{Mons} \mid \mathcal{A}_{\ell'} \in \mathrm{MDS}(\mathcal{A}_\ell)\}$. A monitor $\mathcal{A}_{\mathrm{m}_i}$ depends on another monitor $\mathcal{A}_{\mathrm{m}_j}$ iff m_j appears in the expressions on the transitions of $\mathcal{A}_{\mathrm{m}_i}$.

Proposition 6.3 (Sufficient conditions for monitorability of decentralized specifications.). *A decentralized specification is monitorable if the two following conditions are met: (i)* $\mathrm{MDG}(\mathrm{Mons})$ *has no cycles; and (ii)* $\forall \ell \in \mathrm{Mons} : \ \mathrm{monitorable}(\mathcal{A}_\ell)$.

The first condition ensures that no cyclical dependency exists between monitors. The second condition ensures that all monitors are individually monitorable. We note, that both conditions are decidable. Furthermore, detecting cycles in a graph can be done in linear time with respect to the sum of nodes and edges, by doing a depth-first traversal with back-edge detection, or by finding strongly connected components [49]. Thus, in worst case, it is quadratic in |Mons|. Monitorability is therefore quadratic in the number of monitors and states in the largest automaton.

Example 6.4 (Decentralized monitorability of decentralized specifications.). We consider the decentralized counterpart of the light switch and bulb presented in Example 6.2. The decentralized specification is shown in Figure 3b, it introduces two monitors $\mathcal{A}_{\mathrm{m}_0}$ and $\mathcal{A}_{\mathrm{m}_1}$. The set of monitors is $\mathrm{Mons} \stackrel{\text{def}}{=} \{\mathcal{A}_{\mathrm{m}_0}, \mathcal{A}_{\mathrm{m}_1}\}$.

We compute the monitor dependency sets for each monitor. We have $\mathrm{MDS}(\mathcal{A}_{\mathrm{m}_0}) = \mathrm{dep}(\top) \cup \mathrm{dep}(\mathrm{s}) \cup \mathrm{dep}(\neg \mathrm{s}) \cup \mathrm{dep}(\mathrm{s} \wedge \mathrm{m}_1) \cup \mathrm{dep}(\mathrm{s} \wedge \neg \mathrm{m}_1) = \{\mathcal{A}_{\mathrm{m}_1}\}$, and $\mathrm{MDS}(\mathcal{A}_{\mathrm{m}_1}) = \mathrm{dep}(\top) \cup \mathrm{dep}(\ell) \cup \mathrm{dep}(\neg \ell) = \emptyset$. Using the monitor dependency sets, we construct the monitor dependency graph: $\mathrm{MDG}(\mathrm{Mons}) = \langle \mathrm{Mons}, \{\langle \mathcal{A}_{\mathrm{m}_0}, \mathcal{A}_{\mathrm{m}_1} \rangle\} \rangle$. The monitor dependency graph has no cycles, as it contains only one edge indicating the dependency of $\mathcal{A}_{\mathrm{m}_0}$ on $\mathcal{A}_{\mathrm{m}_1}$.

We now verify the monitorability of each monitor separately using centralized monitorability. Both $\mathcal{A}_{\mathrm{m}_0}$ and $\mathcal{A}_{\mathrm{m}_1}$ are monitorable as the states q_2 and q_1' or q_2' are reachable from all states.

The requirement for no cycles is sufficient but not necessary, it is possible for certain cycles to exist while the decentralized specification is still able to reach a final verdict. This stems from the fact that boolean expressions may cancel out the dependency, or dependencies can be on different timestamps (i.e., future transitions in the automaton). We illustrate a monitorable decentralized specification in *Figure 5* with two monitors that depend on each other. Regardless of the choice of the root monitor, it is possible to still avoid the dependency if one operands of the disjunction

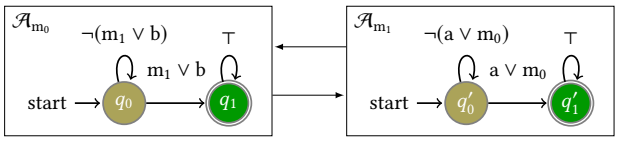


Fig. 5. A monitorable decentralized specification with cyclically dependent monitors. When observing $\langle a, \top \rangle$, and $\langle b, \top \rangle$, the disjunction cancels out the dependency.

holds true. That is, if we observe $\langle a, \top \rangle$ then it is no longer necessary to evaluate m_0, and no real dependency exists.

6.2 Compatibility

A key advantage of decentralized specifications is the ability to associate monitors to components. This allows us to associate the monitor network with the actual system architecture constraints.

The monitor network is a graph $N = \langle \text{Mons}, E \rangle$, where Mons is the set of monitors, and E representing the communication edges between monitors. The monitor network is typically generated by a monitoring algorithm during its *setup* phase (See Section 7.2). For example, N could be obtained using the construction MDG(Mons) presented in Section 6.1.2. The system is represented as another graph $S = \langle C, E' \rangle$, where C is the set of components, and E' is the set of communication channels between components.

Defining compatibility. We now consider checking for *compatibility*. Compatibility denotes whether a monitoring network can be actually deployed on the system. That is, it ensures that communication between monitors is possible when those are deployed on the components. We first consider the reachability in both the system and monitor graphs as the relations $\text{reach}_S : C \to 2^C$, and $\text{reach}_M : \text{Mons} \to 2^{\text{Mons}}$, respectively. Second, we recall that a monitor may depend on other monitors and also on observations local to a component. If a monitor depends on local observations, then it provides us with constraints on where it should be placed. We identify those constraints using the partial function $\text{cdep} : \text{Mons} \to C$. We can now formally define compatibility. Compatibility is the problem of deciding whether or not there exists a *compatible assignment*.

Definition 6.5 (Compatible assignment). A compatible assignment is a function $\text{compat} : \text{Mons} \to C$ that assigns monitors to components while preserving the following properties:

1) $\forall m_1, m_2 \in \text{Mons} : m_2 \in \text{reach}_M(m_1) \implies \text{compat}(m_2) \in \text{reach}_S(\text{compat}(m_1))$;
2) $\forall m \in \text{dom}(\text{cdep}) : \text{cdep}(m) = \text{compat}(m)$.

The first proposition ensures that reachability is preserved. That is, it ensures that if a monitor m_1 communicates with another monitor m_2 (i.e. $m_2 \in \text{reach}_M(m_1)$), then m_2 must be placed on a component reachable from where m_1 is placed (i.e. $\text{compat}(m_2) \in \text{reach}_S(\text{compat}(m_1))$). The second proposition ensures that dependencies on local observations are preserved. That is, if a monitor m depends on local observations from a component $c \in C$ (i.e. $\text{cdep}(m) = c$), then m must be placed on c (i.e. $\text{cdep}(m) = \text{compat}(m)$).

Computing compatibility. We next consider the problem of finding a *compatible assignment* of monitors to components. Algorithm 1 finds a compatible assignment for a given monitor network ($\langle \text{Mons}, E \rangle$), system ($\langle C, E' \rangle$), and an initial assignment of monitors to components (cdep). The algorithm can be broken into three procedures: procedure VERIFYCOMPATIBLE verifies that a (partial) assignment of monitors to components is compatible, procedure COMPATIBLEPROC takes as input a set of monitors that need to be assigned and explores the search space (by iterating over components),

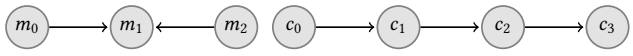
and finally, procedure COMPATIBLE performs necessary pre-computation of reachability, verifies that the constraint is first compatible, and starts the search.

We verify that an assignment of monitors to components ($s : \text{Mons} \rightarrow C$) is compatible using algorithm VERIFYCOMPATIBLE (Lines 1-8). We consider each assigned monitor ($m \in \text{dom}(s)$). Then, we constrain the set of reachable monitors from m to those which have been assigned a component ($M' = \text{reach}_M(m) \cap \text{dom}(s)$). Using M', we construct a new set of components using s (i.e., $C' = \{s(m') \in C \mid m' \in M'\}$). Set C' represents the components on which reachable monitors have been placed. Finally, we verify that the components in the set C' are reachable from where we placed m (i.e., $C' \subseteq \text{reach}_S(s(m))$). If that is not the case, then the assignment is not compatible (Line 4). To iterate over all the search space, that is, all possible assignments of monitors to components, procedure COMPATIBLEPROC (Lines 9-24) considers a set of monitors to assign (M), selects a monitor $m \in$ M (Line 13), and iterates over all possible components, verifying that the assignment is compatible (Lines 14-22). If the assignment is compatible, it iterates over the remainder of the monitors (i.e., M $\setminus \{m\}$), until it is empty (Line 16). If the assignment is not compatible, it discards it and proceeds with another component. For each monitor we seek to find at least one compatible assignment. One can see that the procedure eventually halts (as we exhaust all the monitors to assign), and is affected exponentially based on the number of monitors to assign $|\text{Mons} \setminus \text{dom}(\text{cdep})|$ (Line 31) with a branching factor determined by the possible values to assign ($|C|$, Line 14). It is important to note that the number of monitors to assign is in practice particularly small. The number of monitors to assign includes monitors that depend only on other monitors and not local observations from components, as the dependency on local observations requires that a monitor be placed on a given component (that is, it will be in dom(cdep)).

Example 6.6 (Compatibility). Figure 6 presents a simple network of 3 monitors, and a system graph of 4 components. We consider the following constraint: $\text{cdep} = [m_0 \mapsto c_0, m_2 \mapsto c_2]$. For compatibility, we must first verify that cdep is indeed a compatible (partial) assignment, then consider placing m_1 on any of the components (i.e., both properties of Definition 6.5). Procedure COMPATIBLE computes the set of reachable nodes for both the monitor network and the system. They are presented in Figure 6c and Figure 6d, respectively. We then proceed with line 28 to verify the constraint (cdep) using procedure VERIFYCOMPATIBLE. We consider both m_0 and m_2. For m_0 (resp. m_2) we generate the set (Line 3) $\{c_0\}$ (resp. $\{c_2\}$), and verify that it is indeed a subset of $\text{reach}_S(c_0)$ (resp. $\text{reach}_S(c_2)$). This ensures that the constraint is compatible. We then proceed to place m_1 by calling COMPATIBLEPROC(cdep, $\{m_1\}$, $\{c_0, c_1, c_2, c_3\}$, reach_M, reach_S). While procedure COMPATIBLEPROC will attempt all components, we will consider for the example placing m_1 on c_1. On line 15, the partial function s' will be $\text{cdep} \dagger_2 [m_1 \mapsto c_1]$. We now call VERIFYCOMPATIBLE to verify s'. We consider both m_0, m_1, and m_2. For m_0 (resp. m_1, m_2) we generate the set $\{c_0, c_1\}$ (resp. $\{c_1\}$, $\{c_2, c_1\}$). We notice that for m_0, $\{c_0, c_1\}$ is indeed a subset of $\text{reach}_S(c_0)$. This means that m_0 is able to communicate with m_1. However, it is not the case for m_2, the set $\{c_2, c_1\}$ is not a subset of $\text{reach}_S(c_2) = \{c_2, c_3\}$. The monitor m_2 will not be able to communicate with m_1 if m_1 is placed on c_1. Therefore, assigning m_1 to c_1 is incompatible. Example of compatible assignments for m_1 are c_2 and c_3 as both of those components are reachable from c_2. Procedure COMPATIBLEPROC continues by checking other components, and upon reaching c_2 or c_3 stops and returns that there is at least one compatible assignment. Therefore, the monitor network (Figure 6a) is compatible with the system (Figure 6b).

7 ANALYSIS

We aim to compare decentralized monitoring algorithms in terms of computation, communication, and memory overhead. Since the EHE and *memory* datastructures are used to abstract the behavior



(a) Monitor network. (b) System.

m_0	$\{m_0, m_1\}$
m_1	$\{m_1\}$
m_2	$\{m_2, m_1\}$

(c) reach_M.

c_0	$\{c_0, c_1, c_2, c_3\}$
c_1	$\{c_1, c_2, c_3\}$
c_2	$\{c_2, c_3\}$
c_3	$\{c_3\}$

(d) reach_S.

Fig. 6. Example Compatibility

Algorithm 1 Computing Compatibility

```
1: procedure verifyCompatible(s, reach_M, reach_S)                    ▷ Verify assignment s
2:     for each m ∈ dom(s) do                                         ▷ Consider only assigned monitors
3:         if {s(m') | m' ∈ (reach_M(m) ∩ dom(s))} ⊈ reach_S(s(m)) then   ▷ Check reachability
4:             return false
5:         end if
6:     end for
7:     return true
8: end procedure
9: procedure compatibleProc(s, M, C , reach_M, reach_S)               ▷ Explore assignments
10:    if M = ∅ then                                                  ▷ No monitors left to assign
11:        return ⟨true, s⟩                                           ▷ Successfully assigned all monitors
12:    end if
13:    m ← pick(M)                                                    ▷ Pick a monitor from those left to assign
14:    for each c ∈ C do                                              ▷ Explore assigning monitor to all possible components
15:        s' ← s †2 [m ↦ c]                                          ▷ Add assignment to the existing solution
16:        if verifyCompatible(s', reach_M, reach_S) then             ▷ Is it compatible?
17:            ⟨res, sol⟩ ← compatibleProc(s', M\{m}, C, reach_M, reach_S)   ▷ Recurse on the rest
18:            if res then                                            ▷ Found a compatible assignment for all the rest of M
19:                return ⟨res, sol⟩
20:            end if
21:        end if
22:    end for
23:    return ⟨false, []⟩                                             ▷ No compatible assignment found
24: end procedure
25: procedure compatible(⟨Mons, E⟩, ⟨C, E'⟩, cdep)
26:    reach_M ← computeReach(⟨Mons, E⟩)                              ▷ Precompute reachability
27:    reach_S ← computeReach(⟨C, E'⟩)
28:    if ¬verifyCompatible(cdep, reach_M, reach_S) then              ▷ Check constraint first
29:        return ⟨false, []⟩                                         ▷ Constraint not satisfied
30:    end if
31:    return compatibleProc(cdep, Mons \ dom(cdep), C, reach_M, reach_S)   ▷ Begin exploring
32: end procedure
```

of a monitoring algorithm. We first consider the parameters and the cost for the basic functions of the EHE and *memory* data structures in Section 7.1. We use s_E to denote the size necessary to encode an element of the set E. For example, s_{AP} is the size needed to encode an element of set AP. Then, in Section 7.2, we elaborate on the general phases of decentralized monitoring algorithms and illustrate the approach to analyze them by adapting algorithms from [15] as examples.

7.1 Data Structure Costs

We consider the cost of using a memory or an EHE. To do so, we first address the cost to store partial functions and merge them.

7.1.1 Storing Partial Functions. Since memory and EHE are partial functions, to assess their required memory storage and iterations, we consider only the elements defined in the function. The size of a partial function f, denoted $|f|$, is the size to encode all $x = f(x)$ mappings. We recall that $|\operatorname{dom}(f)|$ the number of entries in f. The size of each mapping $x = f(x)$ is the sum of the sizes $|x| + |f(x)|$. Therefore $|f| = \sum_{x \in \operatorname{dom}(f)} |x| + |f(x)|$.

7.1.2 Merging. Merging two memories or two EHEs is linear in the size of both structures in both time and space. In fact, to construct $f \dagger_{\text{op}} g$, we first iterate over each $x \in \operatorname{dom}(f)$, check whether $x \in \operatorname{dom}(g)$, and if so assign $\text{op}(f(x), g(x))$, otherwise assign $f(x)$. Finally we assign $g(x)$ to any $x \in \operatorname{dom}(g) \cap \overline{\operatorname{dom}(f)}$. This results in $|\operatorname{dom}(f \dagger_{\text{op}} g)| = |\operatorname{dom}(f) \cup \operatorname{dom}(g)|$ which is at most $|\operatorname{dom}(f)| + |\operatorname{dom}(g)|$.

7.1.3 Information delay. The main goal of the EHE data structure is to keep track of partial states of an automaton execution. Keeping track of partial states becomes unnecessary once enough information is gathered to determine which state was reached during an execution. An EHE associates an expression with a state for any given timestamp. When an expression e associated with a state q_{kn} for some timestamp t_{kn} is evaluated to $\top$, we know that the automaton is in q_{kn} at t_{kn}. We call q_{kn} a '*known*' state. The information delay δ_t is the number of timestamps needed to reach a new known state from an existing known state. That is, it is the number of timestamps in the EHE storing partial information without determining a known state. Information delay is a runtime measure, as it depends on the updates done to the EHE as it evolves with time. Given an EHE at timestamp t_{kn} such that q_{kn} is a known state for a given memory $\mathcal{M}^{t_{\text{kn}}}$ i.e. $\text{sel}(\mathcal{I}^{t_{\text{kn}}}, \mathcal{M}^{t_{\text{kn}}}, t_{\text{kn}}) = q_{\text{kn}}$. The next known timestamp is the least timestamp $t_{\text{newkn}} > t_{\text{kn}}$, such that $\text{sel}(\mathcal{I}^{t_{\text{newkn}}}, \mathcal{M}^{t_{\text{newkn}}}, t_{\text{newkn}}) \neq \text{undef}$, where $\mathcal{I}^t$ and $\mathcal{M}^t$ at some timestamp t are used to denote respectively the changes to the EHE and memory through time in the execution[14]. The information delay for this evaluation of a state is $\delta_t = t_{\text{newkn}} - t_{\text{kn}}$. While information delay needs to be computed each time a known state is reached, it is often the case that it is measured for a whole execution of an algorithm, in which case we can consider an average information delay and a maximum information delay, where we aggregate the various information delays (for reaching each known state) by computing their average and maximum. Since we know the automaton is in q_{newkn}, prior information is no longer necessary, therefore it is possible to discard all entries in $\mathcal{I}$ with $t < t_{\text{newkn}}$. Thus, it reduces the number of expressions in the EHE. This can be seen as a garbage collection strategy [48, 53] for the memory and EHE. We next show how the information delay parameter affects the size of the EHE.

7.1.4 EHE Encoding. For the EHE data structure, we consider the three functions: mov, eval, and sel[15] (see Section 4.2). Function mov depends on the topology of the automaton. We quantify it using the maximum size of the expression that labels a transition in a normalized automaton L (see Remark 1), and the number of states in the automaton ($|Q|$). From a known state, each application of mov considers all possible transitions and states that can be respectively taken and reached, for each outbound transition, the label itself is added. Therefore, the rule is expanded by L per

[14] We note that t_{newkn} is not necessarily equal to $t_{\text{kn}} + 1$, as the EHE can determine a known state by simplification, and therefore skip intermediate states. We allow skipping states as it is reasonable for LTL_3 semantics, since final verdicts do not change for all suffixes.

[15] verAt is simply a sel followed by a $O(1)$ lookup.

$$
\begin{array}{lll}
t & \mapsto q & \mapsto \top \\
\delta_t \left\{ \begin{array}{lll}
 & q_0 & \mapsto e_{10} \\
 & q_1 & \mapsto e_{11} \\
t+1 \quad \mapsto & & \vdots \\
 & q_{|Q|-1} & \mapsto e_{1(|Q|-1)} \\
 & q_0 & \mapsto e_{20} \\
t+2 \quad \mapsto & & \vdots \\
 & q_{|Q|-1} & \mapsto e_{2(|Q|-1)} \\
\vdots & & \\
 & q_0 & \mapsto e_{\delta_t 0} \\
 & q_1 & \mapsto e_{\delta_t 1} \\
t+\delta_t \quad \mapsto & & \vdots \\
 & q_{|Q|-1} & \mapsto e_{\delta_t (|Q|-1)}
\end{array} \right.
& \begin{array}{l} \\ \\ \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} |Q| \\ \\ \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} |Q| \\ \\ \\ \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} |Q| \end{array}
\end{array}
$$

Fig. 7. Size of the EHE (worst-case) with respect to information delay.

outbound state for each move beyond t_{kn}. We illustrate the expansion in Figure 7, where for each timestamp, we associated an expression with each state.

We use $\mathrm{S}(t)$ to denote the size of an expression at t timestamps after a known state. As such to reach a given state, we require a previous expression (i.e., $\mathrm{S}(t-1)$), and add the label of a given transition (of maximum size L). In the worst-case, the automaton is a fully connected graph, a state can be reached by all other states (including itself). Hence, we require the disjunction of $|Q|$ such expressions. The recurrence relation is given by: $\mathrm{S}(t) = |Q| \times (\mathrm{S}(t-1) + L)$. $\mathrm{S}(t)$ is thus a geometric series of ratio strictly greater than 1. There is a unique expression at a known time stamp and the size of such expression is 1 (since the expression is $\top$). We can then deduce that the size of the expression is exponential in the number of timestamps. An EHE contains $\delta_t \times |Q|$ expressions. In the worst case, its total size is exponential in the number of states. For $\delta_t \geq 1$, we have:

$$|\mathcal{I}^{\delta_t}| = |Q| \times \delta_t \times \left(\left(1 - \frac{L}{1-|Q|} \right) \times \frac{1-|Q|^{\delta_t}}{1-|Q|} + \delta_t \times \frac{L}{1-|Q|} \right) = \Theta\left(|Q|^{\delta_t - 1} \right)$$

For a given expression e, we use $|\mathsf{e}|$ to denote the size of e, i.e., the number of atoms in e. Given a memory $\mathcal{M}$, the complexity of function eval(e, $\mathcal{M}$) is the cost of simplify(rw(e, $\mathcal{M}$)). Function rw(e, $\mathcal{M}$) looks up each atom in e in $\mathcal{M}$ and attempts to replace it by its truth value. The cost of a memory lookup is $\Theta(1)$, and the replacement is linear in the number of atoms in e. It effectively takes one pass to syntactically replace all atoms by their values, therefore the cost of rw is $\Theta(|\mathsf{e}|)$. However, applying function simplify() requires solving the Minimum Equivalent Expression problem which is Σ_2^p-complete [12], it is exponential in the size of the expression, making it the most costly function. $|\mathsf{e}|$ is bounded by $\delta_t \times L$. Function sel() requires evaluating every expression in the EHE. For each timestamp we need at most $|Q|$ expressions, and the number of timestamps is bounded by δ_t.

7.1.5 Memory. The memory required to store $\mathcal{M}$ depends on the trace, namely the amount of observations per component. Recall that once a state is known, observations can be removed, the

number of timestamps is bounded by δ_t. The size of the memory is then:

$$\sum_{t=i}^{i+\delta_t} |\mathrm{tr}(c,t)| \times (\mathrm{s}_{\mathbb{N}} + \mathrm{s}_{AP} + \mathrm{s}_{\mathbb{B}_2}).$$

The size of the memory depends for each timestamp on the number of observations associated with the component ($|\mathrm{tr}(c,t)|$), and the size of each observation. The size of an observation is the size needed to encode the timestamp ($\mathrm{s}_{\mathbb{N}}$), the atomic proposition (s_{AP}) and the verdict ($\mathrm{s}_{\mathbb{B}_2}$).

7.2 Analyzing Existing Algorithms

We now shift the focus to the algorithms and their usage of the data structures. We first present an overview of the abstract phases performed by decentralized monitoring algorithms. We then elaborate on our approach to model their behavior. Finally, as an example, we present the analysis for each of the algorithms adapted from [15]: Orchestration (Orch), Migration (Migr), and Choreography (Chor). The algorithms contain both multiple monitors monitoring the same specification (Orch, and Migr), and a decentralization algorithm which splits one global specification to multiple subspecifications distributed on monitors (Chor). We later explore the trends provided by the analysis by benchmarking in Section 9.

7.2.1 Overview. A decentralized monitoring algorithm consists of two steps: setting up the monitoring network, and monitoring. In the first step, an algorithm initializes the monitors, defines their connections, and attaches them to the components. We represent the connections between the various monitors using a directed graph $\langle \mathrm{Mons}, E\rangle$ where $E = 2^{\mathrm{Mons}\times\mathrm{Mons}}$ defines the edges describing the sender-receiver relationship between monitors. For example, the network $\langle\{\mathrm{m}_0, \mathrm{m}_1\}, \{\langle \mathrm{m}_1, \mathrm{m}_0\rangle\}\rangle$ describes a network consisting of two monitors m_0 and m_1 where m_1 sends information to m_0. In the second step, an algorithm proceeds with monitoring, wherein each monitor processes observations and communicates with other monitors.

We consider the existing three algorithms: Orchestration, Migration and Choreography [15] adapted to use EHE. We note that these algorithms operate over a global clock, therefore the sequence of steps can be directly mapped to the timestamp. We choose an appropriate encoding of *Atoms* to consist of a timestamp and the atomic proposition ($\mathit{Atoms} = \mathbb{N} \times AP$). These algorithms are originally presented using an LTL specification instead of automata, however, it is possible to obtain an equivalent Moore automaton as described in [9].

7.2.2 Approach. A decentralized monitoring algorithm consists of one or more monitors that use the EHE and memory data structures to encode, store, and share information. By studying δ_t, we derive the size of the EHE and the memory a monitor would use (see Section 7.1) . Knowing the sizes, we determine the computation overhead of a monitor, since we know the bound on the number of simplifications a monitor needs to make ($\delta_t \times |Q|$), and we know the bounds on the size of the expression to simplify ($\delta_t \times L$). Once the cost per monitor is established, the total cost for the algorithm can be determined by aggregating the costs per monitors. This can be done by summing to compute total cost or by taking the maximum cost in the case of concurrency following the Bulk Synchronous Parallel (BSP) [52] approach.

7.2.3 Orchestration. The orchestration algorithm (Orch) consists in setting up a main monitor which will be in charge of monitoring the entire specification. However since that monitor cannot access all observations on all components, orchestration introduces one monitor per component to forward the observations to the main monitor. Therefore, for our setup, we consider the case of a main monitor m_0 placed on component c_0 which monitors the specification and $|C| - 1$ forwarding monitors that only send observations to m_0 (labeled m_k with $k \in [1, |C|]$). We consider that the

reception of a message takes at most d rounds. The information delay δ_t is then constant, $\delta_t = \mathsf{d}$. The number of messages sent at each round is $|C| - 1$, i.e., the number of forwarding monitors sending their observations. The size of a message is linear in the number of observations for the component, for a forwarding monitor labeled with m_k, the size of the message is $|\mathrm{tr}(t, c_k)| \times (\mathrm{s}_{\mathbb{N}} \times \mathrm{s}_{AP} \times \mathrm{s}_{\mathbb{B}_2})$.

7.2.4 Migration. The migration algorithm (Migr) initially consists in rewriting a formula and migrating from one or more component to other components to fill in missing observations. We call the monitor rewriting the formula the active monitor. Our EHE encoding guarantees that two monitors receiving the same information are in the same state. Therefore, monitoring with Migration amounts to rewriting the EHE and migrating it across components. Since all monitors can send the EHE to any other monitor, the monitor network is a strongly-connected graph. In Migr, the delay depends on the choice of function choose, which determines which component to migrate to next upon evaluation. By using a simple function choose, which causes a migration to the component with the atom with the smallest timestamp, it is possible to view the worst case as an expression where for each timestamp we depend on information from all components, therefore $|C| - 1$ rounds are necessary to get all the information for a timestamp ($\delta_t = |C| - 1$). We parametrize Migr by the number of active monitors at a timestamp m. The presented function choose in [15], selects at most one other component to migrate to. Therefore, after the initial choice of m, subsequent rounds can have at most m active monitors.

We illustrate Migr in Algorithm 2. The state of a migration monitor consists of a variable isActive that determines whether or not the monitor is active, and $\mathcal{I}$ that is an EHE encoding the same automaton shared by all monitors. At each round the monitor receives a timestamp t and a set of observations o. Line 2 displays the memory update with observations for that round. Lines 3 to 9 describe the reception of EHEs from other monitors. Upon receiving an EHE, the monitor state is set to active (Line 7). An active monitor will then update its EHE by first ensuring that it is expanded to the current timestamp using mov (Line 11), then rewriting and evaluating each entry (Lines 12-17). The number of entries in the EHE depends on δ_t. If any of the entries is evaluated to a final verdict (Line 14), then the verdict is found and we terminate. While the verdict is not found, the migration algorithm first removes all unnecessary entries in the EHE (Line 18). Unnecessary entries are entries for which the state is known, the last known state is only kept, all previous timestamps are removed. After removing unnecessary entries, we determine a new monitor to continue monitoring using the function choose (Lines 19-22). The initial choice of active monitors is bounded by $m \leq |C|$. Since at most $m - 1$ other monitors can be running, there can be $(m - 1)$ merges. In the worst case, the upper bound on the size of EHE is exponential in δ_t, and thus, it is exponential in the number of components. The number of messages is bounded by the number of active monitors m. The size of each message is however the size of the EHE, since Migr requires the entire EHE to be sent.

7.2.5 Choreography. Choreography (Chor) presented in [14, 15] splits the initial LTL formula into subformulas and delegates each subformula to a component. Thus Chor can illustrate how it is possible to monitor decentralized specifications. Once the subformulas are determined by splitting the main formula [16], we adapt the algorithm to generate an automaton per subformula to monitor it. To account for the verdicts from other monitors, the set of possible atoms is extended to include the verdict of a monitor identified by its id. Therefore, $Atoms = (\mathbb{N} \times AP) \cup (\mathrm{Mons} \times \mathbb{N})$. Monitoring is done by replacing the subformula by the id of the monitor associated with it. Therefore, monitors are organized in a tree, the leaves consisting of monitors without any dependencies, and dependencies building up throughout the tree to reach the main monitor that outputs the verdict. Since each monitor is in charge of evaluating a subformula, the monitors communicate the evaluation of the

[16] Details of the generation is provided in Appendix E.

Algorithm 2 Migration

```
1:  procedure Migration(t, o)
2:      M ← M †2 memc(o, ts_t)                          ▷ Add observations to memory
3:      while Received I′ do                            ▷ Received an EHE from another monitor
4:          if isActive then                            ▷ If the monitor is active, the monitor EHE has information
5:              I ← I †∨ I′                             ▷ Merge information with existing information
6:          else
7:              I ← I′; isActive ← ⊤                    ▷ Monitor becomes active after it receives an EHE
8:          end if
9:      end while
10:     if isActive then
11:         t′ ← getEnd(I); I ← mov(I, t′, t)           ▷ Build EHE up to current timestamp
12:         for each t_v ∈ dom(I) do                    ▷ Go through all EHE timestamps
13:             v ← verAt(I, M, t_v)                    ▷ Evaluate the entries associated with timestamp t_v
14:             if v ∈ B2 then                          ▷ Found a final verdict
15:                 Report v and terminate
16:             end if
17:         end for
18:         I ← dropResolved(I)                         ▷ Purge EHE of non-needed entries
19:         c_k ← choose(I)                             ▷ Determine the next component
20:         if c_k ≠ c then                             ▷ Is the next component not local
21:             isActive ← ⊥; Send I to m_k             ▷ Send to relevant monitor, stop monitoring
22:         end if
23:     end if
24: end procedure
```

formula as a verdict $\mathbb{B}_2$ when it is resolved. Furthermore, monitors may instruct other monitors to stop monitoring as they are no longer necessary. The two messages are referred to as msg_{ver} and msg_{kill}, respectively. For each monitor labeled $\ell \in AP_{\text{mons}}$ we determine the set $coref_\ell \in 2^{AP_{\text{mons}}}$ which contains the labels of monitors that send their verdicts to monitor $\mathcal{A}_\ell$. The information delay for a monitor is thus dependent on its depth in the network tree. The depth of a monitor labeled ℓ that depends on the set of monitors $coref_\ell$, is computed recursively as follows:

$$\text{depth}(\ell) = \begin{cases} 1 & \text{if monitorable}(\mathcal{A}_\ell) \wedge coref_\ell = \emptyset, \\ 1 + \max(\{\text{depth}(\ell') \mid \ell' \in coref_\ell\}) & \text{if monitorable}(\mathcal{A}_\ell) \wedge coref_\ell \neq \emptyset, \\ \infty & \text{otherwise,} \end{cases}$$

A monitor synthesized from a non-monitorable specification will never emit a verdict, therefore its depth is ∞. A leaf monitor has no dependencies, its depth is 1. Since the depth controls the information delay (δ_t), it is possible in the case of choreography to obtain a large EHE depending on the specification. In effect, the worst case the size of the EHE can be linear in the size of the trace $\delta_t = |\text{tr}|$, as it will be required to store the EHE until the end of the trace. As such properties of the specification such as monitorability (see Section 6.1) impact greatly the delay, and thus performance. In terms of communication, the number of monitors generated determines the number of messages that are exchanged. By using the naive splitting function (presented in [15]), the number of monitors depends on the size of the LTL formula. Therefore, we expect the number of messages to grow with the number of atomic propositions in the formula. By denoting $|E|$ the number of edges between monitors, we can say that the number of messages is linear in $|E|$. The size of the messages is constant, it is the size needed to encode a timestamp, id and a verdict in the case of msg_{ver}, or only the size needed to encode an id in the case of msg_{kill}.

7.2.6 Discussion. We summarize the main parameters that affect the algorithms in Table 3. This comparison could serve as a guide to choose which algorithm to run based on the environment (architectures, networks etc). For example, on the one hand, if the network only tolerates short message sizes but can support a large number of messages, then Orch or Chor is preferred over Migr. On the other hand, if we have heterogeneous nodes, as is the case in the client-server model, we

Table 3. Scalability of existing algorithms.

Algorithm	δ_t	**# Msg**	**|Msg|**
Orchestration	$\Theta(1)$	$\Theta(\|C\|)$	$\Theta(\|AP_c\|)$
Migration	$\mathcal{O}(\|C\|)$	$\mathcal{O}(m)$	$\Theta(\|Q\|^{\|C\|-2})$
Choreography	$\mathcal{O}(\text{depth(rt)} + \|\text{tr}\|)$	$\Theta(\|E\|)$	$\Theta(1)$

might want to offload the computation to one major node, in this scenario Orch would be preferable as the forwarding monitor require no computation. This choice can be further impacted by the network topology. In a ring topology for instance, one might want to consider using Migration (with $m = 1$), as using Orch might impose further delay in practice to relay all information, while in a star topology, using Orch might be preferable. In a more hierarchical network, Chor can adapt its monitor tree to the hierarchy of the network. Since we perform a worst-case analysis, we investigate the trends shown in Section 9 by simulating the behavior of the algorithms on a benchmark consisting of randomly generated specifications and traces. Furthermore, we use a real example in Section 9.2 to refine the comparison by looking at six different specifications.

8 THE THEMIS FRAMEWORK

THEMIS is a framework to facilitate the design, development, and analysis of decentralized monitoring algorithms; developed using Java and AspectJ [36] (~5700 LOC).[17] It consists of a library and command-line tools. The library provides all necessary building blocks to develop, simulate, instrument, and execute decentralized monitoring algorithms. The command-line tools provide basic functionality to generate traces, execute a monitoring run and execute a full experiment (multiple parametrized runs).

The purpose of THEMIS is to minimize the effort required to design and assess decentralized monitoring algorithms. THEMIS provides an API for monitoring and necessary data structures to load, encode, store, exchange, and process observations, as well as manipulate specifications and traces. These basic building blocks can be reused or extended to modify existing algorithms or design new more intricate algorithms. To assess the behavior of an algorithm, THEMIS provides a base set of metrics (such as messages exchanged and their size, along with computations performed), but also allows for the definition of new metrics by using the API or by writing custom AspectJ instrumentation. These metrics can be used to assess existing algorithms as well as newly developed ones. Once algorithms and metrics are developed, it is possible to use existing tools to perform monitoring runs or full experiments. Experiments are used to define sets of parameters, traces and specifications. An experiment is effectively a folder containing all other necessary files. By bundling everything in one folder, it is possible to share and reproduce the experiment[18]. After running a single run or an experiment, the metrics are stored in a database for postmortem analysis. These can be queried, merged or plotted easily using third-party tools. After completing the analysis, algorithms and metrics can be tuned so as to refine the design as necessary.

The THEMIS framework has been improved since [23, 24] to support fully distributed and multi-threaded for monitoring by adding the tool Node that acts as a runtime. One or more nodes can be deployed on a given platform. A node receives information (via commands) to deploy components, and monitors on the current platform. Each component contains one or multiple peripheries. A periphery is an input stream to the component, that generates observations. Peripheries follow a

[17] The THEMIS framework is further described and demonstrated in the tool-demonstration paper [24] and on its Website [25].
[18] Experiments provided in this paper are provided at [26], earlier experiments are provided at [25]

stream interface, and waits on a `next()` call to generate the next observations. Monitors are attached to components, and receive the observations that components receive. Thus, a node follows a publish-subscribe model. Components can be seen as topics, where a monitor registers to a topic. Peripheries produce a stream of observations for components. Peripheries can include reading traces from files, over network sockets, or be generated in a stream. Nodes can communicate with other nodes in a distributed manner, through sockets. The implementation of a node defines the high level assumptions of monitoring, for example, our round-based monitoring approach is implemented as a node. For our implementation of a node, reading peripheries to generate component observations is done in parallel. Once all peripheries have executed their `next()` call to read the next events on the stream, the observations are aggregated in an event that is sent to monitors associated with the component. Monitors execute in parallel, once all monitors have completed for a given round, the next round begins. This behavior could be altered to ignore rounds, and simply monitor as soon as information is available by using a different node implementation. Measures have been updated to be thread-safe and work at the node-level.

Furthermore, the simplification logic (operation eval) for the data structure EHE has been greatly improved to call the simplifier less and be more aggressive with the simplifications. This yields on average, a much smaller EHE, more details are provided in Appendix C.

9 COMPARING ALGORITHMS WITH THEMIS

We use THEMIS to compare the adapted versions of existing algorithms (Orch, Migr, and Chor), introduced as examples to validate the trends presented in the analysis in Section 7. Furthermore, since the analysis presented worst-case scenarios, we look at the usefulness of simulations to determine the advantages or disadvantages of certain algorithms in specific scenarios. The THEMIS tool, the data for both scenarios used in this paper, the scripts used to process the data, and the full documentation for reproducing the experiments is found at [26].

Overview of scenarios. We additionally consider a round-robin variant of Migr, Migrr, and use that for analyzing the behavior of the migration family of algorithms as it has a predictable heuristic (function choose). We compare the algorithms under two scenarios. The first scenario explores synthetic benchmarks, that is, we consider random traces and specifications. This allows us to account for different types of behavior. The second scenario explores a specific example associated with a common pattern in programming. For that, we consider a publish-subscribe system, where multiple publishers subscribe to a channel (or topic), the channel publishes events to the subscribers. We use the Chiron user interface example [3, 50], along with the specifications formalized for it [22].

Monitoring metrics. The first considered metric is that of information delay (δ_t) (Section 7.1). The information delay impacts the size of the EHE and therefore the computation, communication costs to send an EHE structure, and also the memory required to store it. To compute the average information delay (*average delay*), we first consider the timestamp difference when an EHE is resolved (i.e., it indicates a state). We sum these differences across the entire run and count the number of resolutions. As such, we acquire the average number of timestamps stored in an EHE. We notice that it is possible for delay to fall below 1, as some traces can cause some monitors to emit a verdict at the very first timestamp. By considering our analysis in Section 7, we split our metrics into two main categories: computation and communication. The EHE structure requires the evaluation and simplification of a Boolean expression which is costly (see Section 4.2). To measure computation, we can count the number of expressions evaluated (using memory lookup), and the number of calls to the simplifier. For this experiment we consider the calls to the simplifier. Since algorithms may have more than one monitor active, we consider for a given round the monitor with the most

simplifications. We sum the maximum number of simplifications per round across all the rounds, and then normalize by the number of rounds. This allows us to determine the slowest monitor per round, as other monitors are executing in parallel. Therefore, we determine the bottleneck. We refer to this metric as critical simplifications. This can be similarly done by considering the number of expressions evaluated. Since monitors can execute in parallel, we introduce *convergence* as a metric to capture load balancing across a run of length n, where:

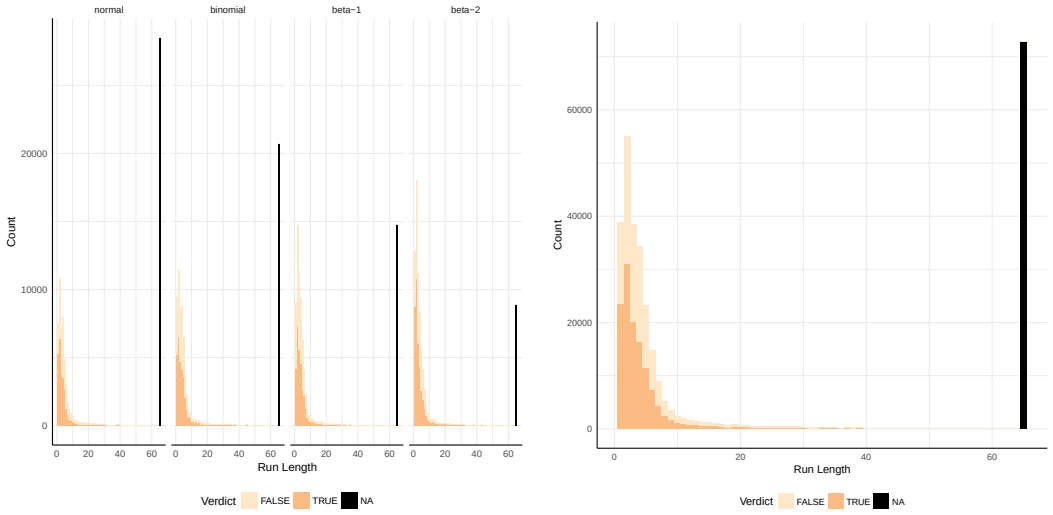
$$\text{conv}(n) = \frac{1}{n}\sum_{t=1}^{n}\left(\sum_{c\in C}\left(\frac{s_c^t}{s^t} - \frac{1}{|C|}\right)^2\right), \text{ with } s^t = \sum_{c\in C} s_c^t.$$

At a round t, we consider all simplifications performed on all components s^t and for a given component s_c^t. Then, we consider the ideal scenario where computations have been spread evenly across all components. Thus, the ideal ratio is $\frac{1}{|C|}$. We compute the ratio for each component ($\frac{s_c^t}{s^t}$), then its distance to the ideal ratio. Distances are added for all components across all rounds then normalized by the number of rounds. The higher the convergence the further away we are from having all computations spread evenly across components. Convergence can also be measured similarly on evaluated expressions. We consider communication using three metrics: number of messages, total data transferred, and the data transferred in a given message. The number of messages is the total messages sent by all monitors throughout the entire run. The data transferred consists of the total size of messages sent by all monitors throughout the entire run. Both the number of messages and the data transferred are normalized using the run length. Finally, we consider the data transferred in a given message to verify the message sizes. To do so, we normalize the total data transferred using the number of messages.

9.1 Synthetic Scenario

9.1.1 Experimental Setup. We generate the specifications as random LTL formulas using `randltl` from `Spot` [21] then converting the LTL formulae to automata using `ltl2mon` [9]. We generate traces by using the `Generator` tool in `THEMIS` which generates synthetic traces using various probability distributions (provided by COLT[19]). For all algorithms we considered the communication delay to be 1 timestamp. That is, messages sent at t are available to be received at most at $t+1$. In the case of migration, we set the active monitors to 1 ($m = 1$). For our experiment, we vary the number of components between 3 and 5, and ensure that for each number we have 100 formulae that reference all components. We were not able to effectively use a larger number of components since most formulae become sufficiently large that generating an automaton from them using `ltl2mon` fail. The generated formulae were fully constructed of atomic propositions, there were no terms containing $\top$ or $\bot$[20]. We use 200 traces of 60 events per component, we associate with each component 2 observations. Traces are generated using 4 probability distributions (50 traces for each probability distribution). The used distributions include *normal* ($\mu = 0.5, \sigma^2 = 1$), *binomial* ($n = 100, p = 0.3$), and two *beta* distributions: *beta-1* ($\alpha = 2, \beta = 5$), and *beta-2* ($\alpha = 5, \beta = 1$). The varied distributions provide different probability to assign $\top$ and $\bot$ to observations in the traces, as such we achieve varied coverage[21]. Figure 8a shows the outcome of runs for different probability distributions. We notice that by varying the distributions we obtain different distributions of verdicts across all runs for all given specifications. The trace length is chosen to be 60, based on the consideration

[19]COLT provides a set of Open Source Libraries for High Performance Scientific and Technical Computing in Java.[13]
[20]To generate formulae with basic operators, string "true=0,false=0,xor=0,M=0,W=0,equiv=0,implies=0,ap=6,X=2,R=0" is passed to `randltl`.
[21]An observation is assigned $\top$ if the generated number is strictly greater than 0.5, and is otherwise $\bot$. For the binomial distribution, we consider $p = 0.3$ the probability of obtaining $\top$.



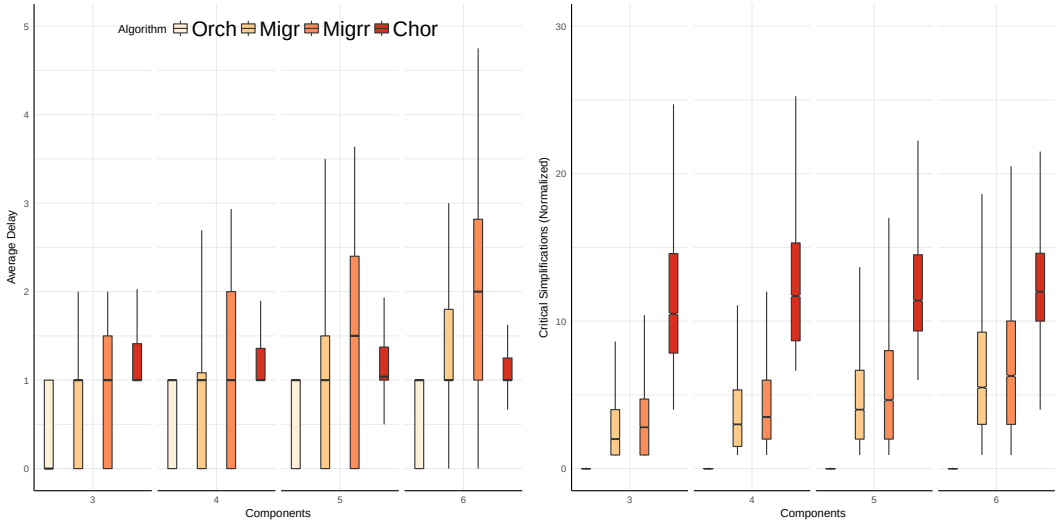
(a) Verdicts for traces based on the different probability distributions.

(b) Verdicts for all runs, across all traces of all probability distributions, using all random generated formulae.

Fig. 8. Verdicts emitted by different run lengths.

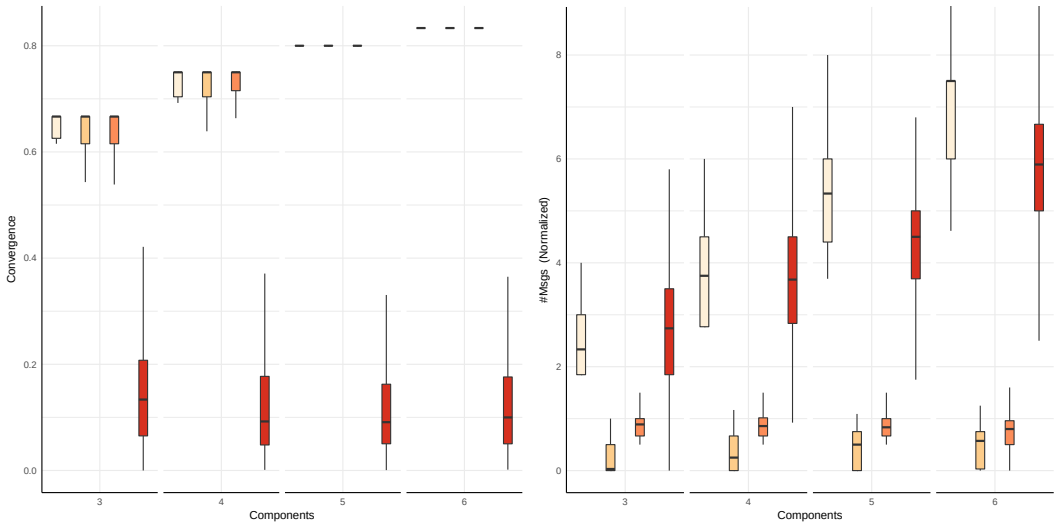
that random formulae usually cause monitor verdicts to either be returned very early or timeout (Figure 8b). The percentage of runs that lasted more than 60 timestamps and returned a final verdict is less than 0.1% of total runs. When computing sizes, we use a normalized unit to separate the encoding from actual implementation strategies. Our assumptions on the sizes follow from the bytes needed to encode data (for example: 1 byte for a character, 4 for an integer). We normalized our metrics using the length of the run, that is, the number of rounds taken to reach the final verdict (if applicable) or timeout, as different algorithms take different numbers of rounds to reach a verdict. In the case of timeout, the length of the run is 65 (length of the trace, and 5 additional timestamps to timeout).

9.1.2 Comparing Algorithms. Figures 9 and 10 present the outcome of the proposed metrics for the algorithms. We inspect the behavior of information delay in Figure 9a by computing the average information delay. As expected, orchestration never exceeds a delay of 1. For migration, the delay depends on the heuristic used, as mentioned in Section 7.2, its worst case is the number of components. Migration can still have a lower delay than orchestration in some cases (as observed for $|C| \geq 4$). This observation is due to the initial monitor placement, as in our case we chose the first component always to be where we place the main orchestration monitor (component A), while for migration, the heuristic function (choose) decides which monitor starts. As such, in a specification where the verdict can be resolved at the first timestamp, migration has an advantage. For Chor, the delay is at least 1, as the network depth affects the delay. Furthermore, we notice that the delay for Chor is not particularly affected with the number of components. We know that its worst-case will depend on traces in cases of non-monitorability, we inspect that further in Section 9.2. Figure 9b shows the average maximum computation done by a monitor for a given round. By looking at computation, we notice that Orch performs no simplifications. This is the



(a) Average delay.

(b) Critical simplifications.

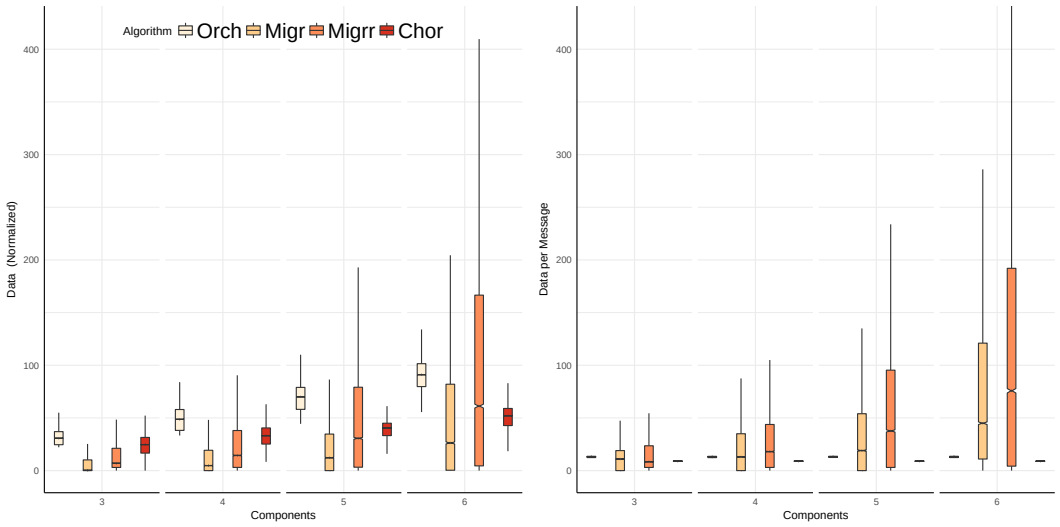


(c) Convergence.

(d) Number of messages.

Fig. 9. Comparison of delay, computation and number of messages. Algorithms are presented in the following order: Orch, Migr, Migrr, Chor.

case as expressions in the EHE do not become sufficiently complex to require simplification. We recall that for orchestration, the memories of all local observations are sent to the main monitor within one timestamp. And as such, by memory lookup, the expression is immediately evaluated without the need to simplify. We notice that for the average case, Migr performs a small number of simplifications, and Chor still executes a reasonable number of simplifications. Figure 9c shows the



(a) Total data transferred.

(b) Data per message.

Fig. 10. Data Transfer

convergence for the algorithms. Since Chor is the only algorithm that performs computations at different components in a given round, we notice that the convergence is much lower.

For communication, we first consider the number of messages transferred normalized by the length of the run. We notice that for algorithms Orch and Chor the number of messages increases with the number of components. Since Chor depends on the edges that connect monitors, it scales better with the number of components than Orch (the depth of the network is usually smaller than the number of components). In contrast, we notice that for Migr and Migrr, the number of messages is independent from the number of components, as it depends on the number of active monitors. Figure 10a presents the total data transferred normalized by the run length. We notice by examining algorithm Orch that sending all observations can be costly. Algorithms Migr and Migrr are capable of sending much less data on average, but have variable behavior, and scale poorly, we notice an increase as $|C|$ increases. Algorithm Chor performs better than Orch, and scales much better with component size. We notice that while Migr and Migrr send fewer messages than the other algorithms, and have better scaling in the number of messages transferred, they can still, in total, send more data depending on the traces and specification. We notice that the 75% quartile for Migrr still exceeds that of Orch. Since total data transferred includes both the number of messages and their sizes, we present the size of the message in Figure 10b by dividing the total data transferred by the number of messages. We observe that for Orch and Chor the size of a message is constant, not very variable and does not depend on $|C|$, while for Migr and Migrr we observe a significant increase as $|C|$ increase. We recall from Section 7.2 that the migration algorithms send the `EHE` which expressions grow exponentialy in the size of the information delay.

9.1.3 Comparing Variants. Using the same dataset, we look at another use case of `THEMIS`; that of comparing variants of the same algorithm. In this case, we focus on differences between Migr and Migrr. The heuristic of Migr improves on the round-robin heuristic of Migrr by choosing to transfer the `EHE` to the component that can observe the atomic proposition with the earliest timestamp

in the EHE (referred to as earliest obligation [15]). Using the simple heuristic, we notice a drop in the delay starting from $|C| > 4$ (Figure 9a). The simple heuristic of earliest obligation seems to reduce on average the delay of the algorithm, interestingly, it maintains a mean of 1. Furthermore, we observe a drop in both messages transferred (Figure 10a) and size of messages (Figure 10b). Consequently, this constitutes a drop in the total data transferred (Figure 10a). We note that the message size is also the size of the EHE. The drop in the number of messages sent is explained by the decision not to migrate when the soonest observation can be observed by the same component, while for Migrr, the round-robin heuristic causes the EHE to always migrate. However, this does not lead to a much lower number of simplifications (Figure 9b). Using THEMIS to compare the variants shows us that the earliest obligation heuristic reduces the size of the EHE, and thus, the size of the message, but also the number of messages sent. However, it does not seem to impact computation as the number of simplifications remains similar.

9.1.4 Discussion. The observed behavior of the simulation aligns with the initial analysis described in Section 7. We observe that the EHE presents predictable behavior in terms of size and computation. The delay presented for each algorithm indeed depends on the listed parameters in the analysis. With the presented bounds on EHE, we can determine and compare the algorithms that use it. Therefore, we can theoretically estimate the situations where algorithms might be (dis)advantaged. However, both Figures 9 and 10 show that for most metrics, we observe a large variance (as evidenced by the interquartile difference). As such, we caution that while the analysis presents trends where algorithms have the advantage, it is still necessary to address the specifics, hence the need for simulation.

9.1.5 Explaining earlier results. In [15], the authors conducted experiments to compare the various algorithms. In particular, they broke down the metrics by delay, message count, message size and number of progressions (rewrites to the formula). We focus on the first three ignoring the last, since we do not monitor by rewriting. First, the authors rank the algorithms from lowest to highest as follows: Orch, Chor, and Migr. This is consistent with our analysis and our synthetic benchmark, since Orch has a constant delay, Chor a delay depending on the depth of the network, and Migr a delay depending on the number of components. However, we note that there are (small) cases where Chor will have a worst-case delay of the size of the trace, this is not reflected in [15]. Second, the authors discuss the number of messages sent by each algorithm (assuming a round-based scheme). The lowest algorithm is Migr (with $m = 1$), followed by Chor, which is followed by Orch. We note that in our analysis (Section 7), the number of messages in migration depends on the number of active monitors, in choreography on the number of edges in the network, and in orchestration on the number of components. Considering that the monitors are organized in a DAG, there will be generally less edges than components (using the splitting strategy), this contributes to choreography outperforming orchestration. This can be seen as the authors state that exceptions allow Orch to perform better than Orch. In particular "for [..] randomly generated formulae of size 5, or when the depth of the network is greater than or equal to 3.". A larger formula and a deeper network require a more complex network organization, which could result in more edges. We recall that the experiment ranged on a number of components between 3 and 5, where not all components were referenced at all times. Third, the authors discuss the size of the messages, ranking the algorithms from short to long messages as follows: Orch, Chor, and Migr. We recall that for Chor the message size is constant, while for Orch the message size depends on the number of observations per component. Either way both are significantly smaller than Migr which sends the full monitoring information. While the analysis shows that in theory, Chor should perform better than Orch with respect to size of the message, we are unsure how the size was captured. Since in the experiments in [15], the size for Chor grows linearly with the size of the formula,

Table 4. Variation of average delay, number of messages, data transfer, critical simplifications and convergence with traces generated using different probability distributions for each algorithm. Number of components is $|C| = 6$. Table cells include the mean and the standard deviation (in parentheses).

Alg.	Trace	δ_t	#Msgs	Data	S_{crit}	$Conv_E$
Orch	normal	0.69 (0.46)	7.13 (1.38)	94.87 (18.46)	0.00 (0.00)	0.83 (0.01)
	binomial	0.69 (0.46)	7.15 (1.39)	93.10 (18.13)	0.00 (0.00)	0.83 (0.01)
	beta-1	0.70 (0.46)	6.98 (1.47)	96.11 (20.17)	0.00 (0.00)	0.83 (0.02)
	beta-2	0.69 (0.46)	6.91 (1.71)	83.37 (20.66)	0.00 (0.00)	0.82 (0.02)
Migr	normal	1.72 (1.42)	0.50 (0.32)	110.13 (276.43)	7.01 (5.01)	0.82 (0.03)
	binomial	1.67 (1.40)	0.49 (0.32)	95.44 (221.65)	6.86 (4.91)	0.82 (0.04)
	beta-1	1.82 (1.44)	0.53 (0.32)	133.15 (313.44)	7.14 (5.16)	0.82 (0.04)
	beta-2	1.53 (1.36)	0.47 (0.38)	56.48 (114.54)	5.95 (4.24)	0.82 (0.03)
Migrr	normal	2.64 (1.93)	0.70 (0.35)	177.48 (358.61)	7.50 (5.18)	0.83 (0.03)
	binomial	2.59 (1.95)	0.69 (0.36)	171.64 (318.94)	7.49 (5.21)	0.83 (0.03)
	beta-1	2.82 (1.94)	0.74 (0.34)	210.02 (452.83)	7.49 (5.23)	0.82 (0.03)
	beta-2	2.55 (2.08)	0.66 (0.41)	162.28 (287.28)	6.93 (4.90)	0.82 (0.02)
Chor	normal	2.02 (1.97)	5.92 (1.60)	52.54 (14.23)	12.68 (3.63)	0.13 (0.10)
	binomial	1.93 (1.86)	5.95 (1.61)	52.76 (14.33)	12.55 (3.70)	0.13 (0.10)
	beta-1	2.59 (4.58)	5.80 (1.64)	51.54 (14.48)	13.29 (4.33)	0.14 (0.12)
	beta-2	2.95 (7.26)	5.81 (1.79)	51.52 (15.93)	13.50 (9.91)	0.14 (0.14)

while by definition the message is defined with a fixed size (ids + verdict). We believe the encoding scheme impacts the size of the message.

9.1.6 Trace Variance. In Table 4, we examine the variance by observing metrics with respect to probability distributions used to generate the traces. To exclude the variance due to the number of components, we fix $|C| = 6$, as it provides the highest variance. For each metric, we present the mean and the standard deviation (between parentheses). All metrics are normalized by the length of the run. The metrics in order of columns are: average information delay (δ_t), average number of messages (#Msgs), total data transferred (Data), average maximum simplifications per monitor (S), and convergence based on expressions evaluated ($Conv_E$). We observe that by changing the probability distribution, the metrics vary significantly. This is particularly prominent for the information delay (especially in the case of Chor), and data transferred. We explore the differences in the algorithms in Section 9.2 by considering real examples with existing formalized specifications.

9.1.7 Impact of Network Delay. In order to assess the impact of the network delay, we fixed the number of components to three, used traces obtained using the normal distribution, and varied network delay by either delaying all messages by a constant delay, or randomizing the delay (up to a certain bound) on a per-message basis. We experiment with two values of delays: 2 and 5 rounds, and show the impact of the network delay on the information delay in Figure 11 for each monitoring algorithm. First, we verify that all verdicts for all delay values match the initial verdicts. This ensures that all runs returned sound verdicts. We note the exceptions of very few runs (2 runs) where the verdict was delayed past the timeout point and therefore could not be reached. We observe that the higher the network delay the higher the information delay. This would cause the algorithms to manipulate larger EHEs but otherwise would not affect correctness. Furthermore, when the delay is variable, we observe overall smaller resulting EHEs, as partial observations often

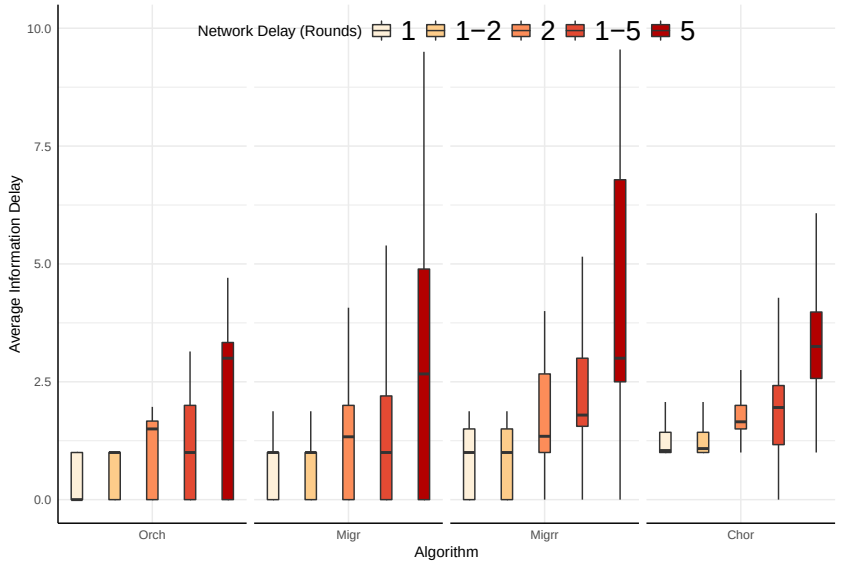


Fig. 11. Effect of network delay on information delay. Variable delay is denoted by min-max and is measured in rounds, stating that each message is randomly delayed for a value between min and max.

allow monitors to reach verdicts before all information is received, that is for an expression a ∨ b, if a is sent with a smaller delay than b and is observed to be ⊤, then the expression is resolved without waiting on b.

9.2 The Chiron User Interface

9.2.1 Overview. Moving away from synthetic benchmarks, we consider properties that apply to patterns of programs and specifications. In this section, we compare the algorithms by looking at a real example that uses the publish-subscribe pattern. To that extent, we consider the Chiron user interface example [3]. Chiron consists of artists responsible of rendering parts of a user interface, that register for various events via a dispatcher. A dispatcher receives events from an abstract data type (ADT) and forwards them to the registered artists. We chose Chiron for two practical reasons. Firstly its example source code (in ADA), and its specifications are available online [50]. The specification is completely formalized and utilizes various LTL patterns described in [1, 22]. Thus, it covers a multitude of patterns for writing specifications. Secondly, the Chiron system can be easily decomposed into various components, we consider four components, the dispatcher (A), the two artists (B,C) and the main thread (D). The main thread is concerned with observing termination of the program.

9.2.2 Experimental Setup. Table 5 lists the subset of the Chiron specification we considered. For each property, column ID references the original property name in [50], column $\mathbb{B}_3$ references the expected verdict at the end of the trace[22], and column pattern identifies the LTL pattern corresponding to the formula. We modify the Chiron example program [50] to output a trace

[22]In the case where the expected verdict is ?, the specification is designed to falsify the property, as such if no falsification is found, we will terminate with verdict ?.

Listing 1. Example Chiron Specification

```
!(notify_client_event_a1_e1 || notify_client_event_a2_e1)
  U (notify_artists_e1 ||
     []!(notify_client_event_a1_e1 || notify_client_event_a2_e1))
```

Table 5. Monitored Chiron specifications. CRC stands for Constrained Response Chain.

ID	$\mathbb{B}_3$	Pattern	Description
1	?	Absence	An artist never registers for an event if she is already registered for that event, and an artist never unregisters for an event unless she is already registered for that event.
2	?	CRC (2-1)	If an artist is registered for an event and dispatcher receives this event, it will not receive another event before passing this one to the artist.
3	$\top$	Precedence	Dispatcher does not notify any artists of an event until it receives this event from the ADT.
5	?	Absence	Dispatcher does not block ADT if no one is registered (this means that if no artists are registered for events of kind 1, dispatcher does nothing upon receiving an event of this kind from the ADT).
7*	?	CRC (3-1)	The order in which artists register for events of kind 1 is the order in which they are notified of an event of this kind by the dispatcher. In other words, if artist1 registers for event2 before artist2 does, then once dispatcher receives event2 from the ADT, it will first send it to artist1 and then to artist2.
15a	?	Universal	The program never terminates with an artist registered.
15b	$\top$	Response	An artist always unregisters before the program terminates. Given that you can't register for the same event twice, we need only check that unregisters respond to registers

of the program, and consider the specifications listed in Table 5[23]. For example, we consider the specification shown in Listing 1. It states that artists are only notified when the dispatcher receives an event. That is, the dispatcher does not send events to the artists without receiving them properly from the ADT. Since we monitor offline, we generate the trace by inserting a global monitor that contains information about all relevant atomic propositions. The program is then instrumented to notify the monitor of events. Specifications and traces are then provided as input to `THEMIS` to process with the existing algorithms. The details on the atomic propositions and their assignment to components can be found in Appendix B. We randomized the events dispatched in the Chiron example, and generated 100 traces of length 279. We targeted generating traces under 300 events. This corresponds to the ADT dispatching 91 events, with the addition of events to register, and unregister artists.

[23]We exclude specification 7 as we were unable to generate an automaton using `ltl2mon` for it. This is due to the formula either being too complex, or non-monitorable.

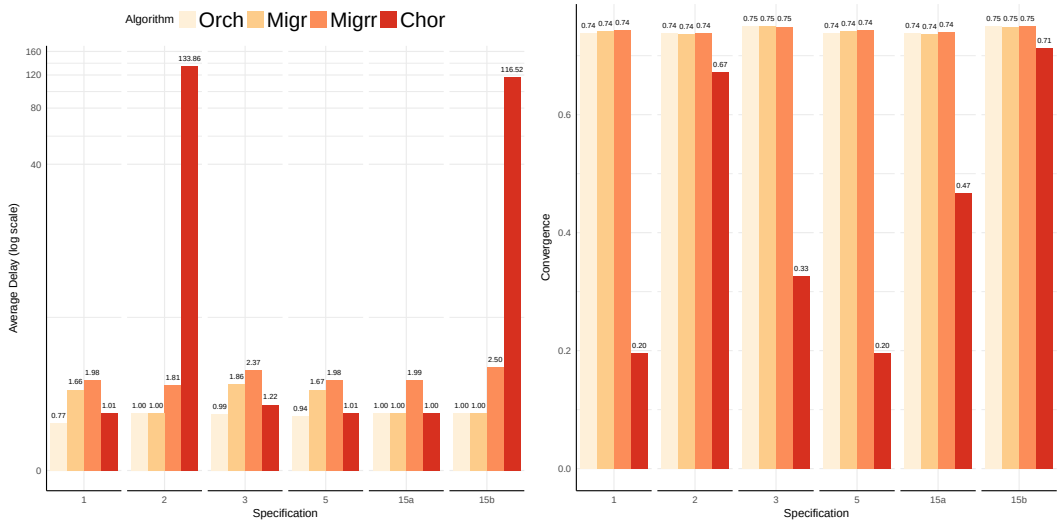
9.2.3 Comparing algorithms. Figure 12 presents the means for the metrics of average delay, convergence, and both critical and maximum simplifications[24]. We immediately notice for Chor the high average delay for specifications 2 and 15b (133.86, and 116.52, respectively). In these two cases, the heuristic to generate the monitor network for choreography has split the network inefficiently, and introduced a large delay due to dependencies. We recall that the heuristic used for choreography consider LTL formulae. For a given formula it counts the number of references to atomic propositions of a given component. The monitor tasked with monitoring the formula will then be associated with the highest component. To generate a decentralized specification, the heuristic starts with an LTL formula and splits it into two subformulas for each binary operator, then one of the subformulas is chosen to remain on the current component while the other is delegated to the component with the most references to atomic propositions. We see in this case that simply counting references and breaking ties using the lexicographical order of the component name can yield inefficient decompositions. Furthermore, we notice that while Orch maintains the lowest delay, other algorithms can still yield comparable delays. In the case of specification 15a we observe that Orch, Migr, and Chor have similar delay (1.0). While Migr may outperform Chor for specifications 2 and 15b, it is the opposite for specifications 1, 3 and 5. This highlights that the network decomposition of monitors (i.e., the setup phase) is an important consideration when designing decentralized monitoring algorithms.

Figure 12b presents the convergence (computed using the number of expressions evaluated). We see that the poor decomposition also yields imbalanced workloads on the monitors. In the case of specifications 2 and 15b, we observe a convergence of 0.67 to 0.71 for Chor, respectively. The observed convergence is comparable with that of Orch and Migr. Furthermore, it is still possible to improve on the load balance for specifications 3 and 15a, as the convergence is high (0.33 and 0.47).

Figure 12c illustrates critical simplifications, we see that Chor has a higher cost compared to Migr in terms of computation. We also notice that Migr performs better than Migrr for all specifications. The heuristic of migrating the formula based on the atomic proposition with the earliest timestamp (earliest obligation) does indeed improve computation costs. More importantly, we notice that the highest delay for Chor is for specifications 2 and 15b. To inspect that, we look at the maximum delay induced in a given monitor for an entire run, and consider the mean across all traces to obtain the worst-case maximum simplifications in Figure 12d. Indeed, we notice a peak in the maximum number of simplification in a given round for specifications 2 and 15b. Particularly, we notice that while comparable in other specifications (e.g., for specification 15a, we have 10 max simplifications for Chor as opposed to 8.64 and 10.00 for Migr and Migrr, respectively), the maximum number of simplifications for Chor increases to 2,798 (compared to 12 for Migrr), and 3,387 (compared to 16.86 for Migrr) for specifications 2 and 15b, respectively. In this particular case, we see how delay can impact the maximum number of simplifications.

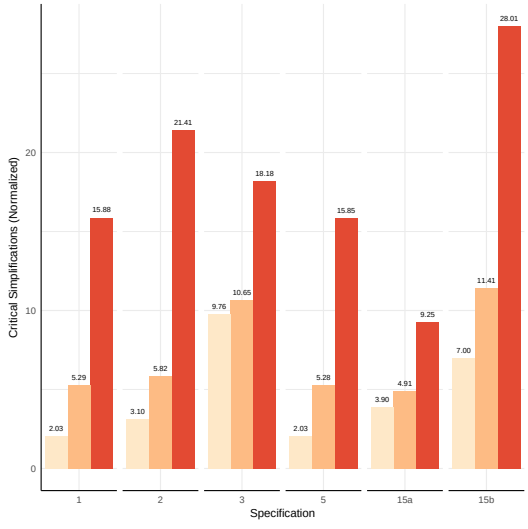
We now consider communication costs by observing the number of messages transferred in Figure 13a. We see that Migr and Migrr perform well compared to the other two algorithms, with Migr performing consistently better than Migrr. We note that the analysis of Migr indicates that the number of messages per round will be in the worst case the number of active monitors (in our case that is 1). One can see in specification 5 that Migr sends only 0.02 messages on average per round, compared to Migrr with 1.01, followed by Orch with 2.95, and finally Chor with 4.89. We see that Orch outperforms Chor in the case of specifications 1, 2, 3 and 5, where generally Orch sends 1-2 messages less. We note that this pattern is in line with the trends shown in Figure 9d. We see for $|C| = 4$ that Orch and Chor overlap, with Migrr outperforming both, and Migr outperforming

[24]We note that since we broke down the metrics per specification, we have little variation in the data, for details and standard deviations refer to Appendix D.

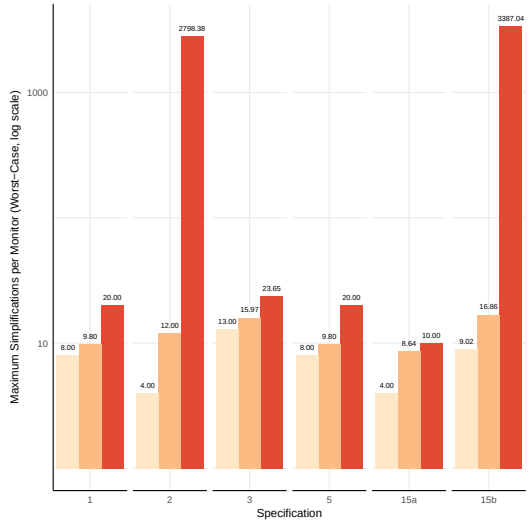


(a) Average delay.

(b) Convergence.



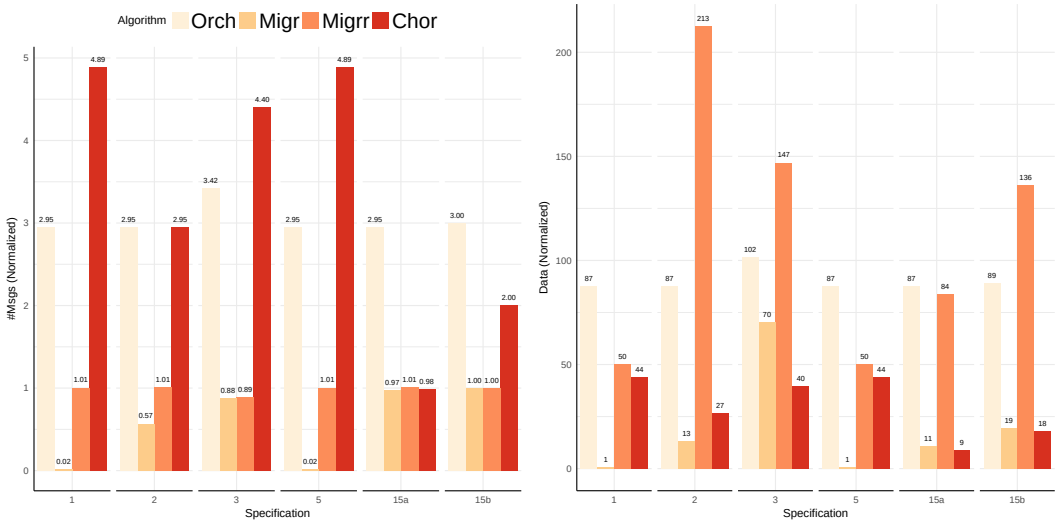
(c) Critical simplifications.



(d) Maximum simplifications (worst case).

Fig. 12. Comparison of delay, convergence and number of simplifications. Algorithms are presented in the following order: Orch, Migr, Migrr, Chor. Orch is omitted in the simplifications count as it is zero.

all other algorithms. Interestingly, we find that in the case of specification 15a in Figure 13a, we see for Chor a number of messages (0.98) slightly higher than Migr (0.97) and lower than Migrr (1.01), consistent with the lower whiskers in Figure 9d. Similarly, when considering the total data transferred in Figure 13b, we see as a trend across specifications Migr being particularly good, while still being slightly outperformed by Chor in specifications 15a and 15b. Furthermore, we notice that Migrr performs poorly and indeed sends more data than Orch in most cases, indicating



(a) Number of messages (normalized). (b) Total data transferred (normalized).

Fig. 13. Data Transfer

that a heuristic can indeed be instrumental in the success of designing the family of migration algorithms. We notice also that the trends from Figure 10a seem to apply in most cases, Orch sends a lot more data than Migr and Chor, with Migrr possibly surpassing Orch.

10 RELATED WORK

Runtime verification/monitoring is a verification technique aiming at *checking* the correctness of a system using runtime information. We refer to [38] for a comparison between runtime verification and other verification techniques.

In this section, we classify and compare with approaches to decentralized monitoring: monitoring by formula rewriting (Section 10.1), monitoring distributed systems (Section 10.2), fault-tolerant monitoring (Section 10.3), and stream-based monitoring (Section 10.4). We also refer to [33] for a recent overview.

10.1 Monitoring by Formula Rewriting

The first class of approaches consists in monitoring by LTL formula rewriting [10, 15, 45]. Given an LTL formula specifying the system, a monitor will rewrite the formula based on information it has observed or received from other monitors, to generate a formula that has to hold on the next timestamp. Typically a formula is rewritten and simplified until it is equivalent to $\top$ (true) or $\bot$ (false) at which point the algorithm terminates. Another approach [51] extends rewriting to focus on real-time systems. They use Metric Temporal Logic (MTL), which is an extension to LTL with temporal operators. This approach also covers lower bound analysis on monitoring MTL formulae. While these techniques are simple and elegant, rewriting varies significantly during runtime based on observations, thus analyzing the runtime behavior could prove difficult if not unpredictable. For example, when excluding specific syntactic simplification rules, $\mathrm{G}(\top)$ could be rewritten $\top \wedge \mathrm{G}(\top)$ and will keep growing in function of the number of timestamps. To tackle the unpredictability of rewriting LTL formulae, another approach [28] uses automata for monitoring

regular languages, and therefore (i) can express richer specifications, and (ii) has predictable runtime behavior. These approaches use a centralized specification to describe the system behavior, they perform *decentralized monitoring of a centralized specification* (Section 2.2), in a system relying on a global clock.

10.2 Monitoring Distributed Systems

Monitoring approaches to monitoring distributed systems typically consider the problem of detecting global predicates. Global predicate detection [42, 43] consists in evaluating predicates on the global state of a distributed system. Approaches performing predicate detection are capable of distributing the evaluation across a distributed system, and evaluate regular predicates which include some temporal logic predicates (such as globally $\Box$ and eventually $\diamond$). The evaluation is done online, and as such can be seen as runtime verification. In [41] the authors extend the approach beyond safety properties to monitor temporal properties in distributed systems. These techniques perform effectively *decentralized monitoring of a centralized specification* (Section 2.2), without assumptions of a global clock.

10.3 Fault-tolerant Monitoring

Another class of research focuses on handling a different problem that arises in distributed systems. In [11], monitors are subject to many faults such as failing to receive correct observations or communicate state with other monitors. Therefore, the problem handled is that of reaching consensus with fault-tolerance and is solved by determining the necessary verdict domain needed to be able to reach a consensus. To remain general, we do not impose the restriction that all monitors must reach the verdict when it is known, as we allow different specifications per monitor. Since we have heterogeneous monitors, we are not particularly interested in consensus. However, for multiple monitors tasked to monitor the same specification, we are interested in strong eventual consistency. We maintain the 3-valued verdict domain and tackle the problem from a different angle by considering the eventual delivery of messages. Similar work [7] extends the MTL approach to deal with failures by modeling knowledge gaps and working on resolving these gaps. We also highlight that the mentioned approaches [7, 10, 15], and other works [20, 46, 47] do in effect define separate monitors with different specifications, typically consisting in splitting the formula into subformulas. Then, they describe the collaboration between such monitors. However, when performing *decentralized monitoring of a centralized specification*, approaches primarily focus on presenting one global formula of the system from which they derive multiple specifications. In our approach, we generalize the notions from a centralized to a decentralized specification and separate the problem of generating multiple specifications equivalent to a centralized specification from the monitoring of a decentralized specification (Section 11).

10.4 Stream-based Monitoring

Specification languages have been developed that monitor synchronous systems as streams [17, 18]. In this setting, events are grouped as a stream, and streams are then aggregated by various operators. The output domain extends beyond the Boolean domain and encompasses types. The stream approach to monitoring has the advantage of aggregating types, as such operations such as summing, averaging or pulling statistics across multiple streams is also possible. Stream combination is thus provided by general-purpose functions, which are more complex to analyze than automata. This is similar to complex event processing where RV is a special case [34]. Specification languages such as LOLA [17] even define dependency graphs between various stream information, and have some properties like *well formed*, and *efficiently monitorable* LOLA specifications. The former ensures that dependencies in the trace can be resolved before they are needed, and the latter ensures that

the memory requirement is no more than constant with respect to the length of the trace. While streams are general enough to express monitoring, they do not address decentralized monitoring explicitly. As such, there is no explicit assignment of monitors to components and parts of the system, nor consideration of architecture. Furthermore, there is no algorithmic consideration addressing monitoring in a decentralized fashion, even-though some works such as [35] do provide multi-threaded implementations.

11 FUTURE DIRECTIONS

By introducing decentralized specifications, we separate the monitor topology from the monitoring algorithm. As such, we address future directions that result from analyzing the topology of monitors, and thus, define properties on such topologies, studying the monitoring by improving metrics, and applying decentralized specifications to the problem of runtime enforcement [27, 32].

Optimized compatibility. The first direction is to extend the notion of *compatibility* (Section 6.2) to not only decide whether or not a specification is applicable to the architecture of the system, but also use the architecture to optimize the placement. That is, one can generate a decentralized specification that balances computation to suit the system architecture, or optimize specific algorithms for specific layouts of decentralized systems (as discussed in Section 7).

Verdict equivalence. We can also compare decentralized specifications to ensure that two specifications emit the same verdict for all possible traces, we elaborate on this property as *verdict equivalence*. We consider two decentralized specifications $\mathcal{D}$ and $\mathcal{D}'$, constructed with two sets of monitors Mons and Mons′ (as per Section 5). Let the root monitors be rt, and rt′, respectively. We recall the notation from Section 4.1, for a given monitor label ℓ, q_{ℓ_0}, Δ_ℓ and ver_ℓ indicate the initial state, transition relation and the verdict function for a given monitor (automaton). One way to assess equivalence is to verify, that for all traces, both specifications yield similar verdicts. It suffices to evaluate the trace on the transition function starting from the root monitor, and check the verdict of the reached state. That is, two decentralized specifications $\mathcal{D}$ and $\mathcal{D}'$ are *verdict equivalent* iff $\forall t \in \mathcal{T} : \text{ver}_{\text{rt}}(\Delta'^{*}_{\text{rt}}(q_{\text{rt}_0}, t, 1)) = \text{ver}_{\text{rt}'}(\Delta'^{*}_{\text{rt}'}(q_{\text{rt}'_0}, t, 1))$. The verdict equivalence property establishes the basis for comparing two specifications that eventually output the same verdicts for the same traces. For all possible traces ($\forall t \in \mathcal{T}$), we first evaluate the trace on the root monitor of $\mathcal{D}$ (i.e., $\Delta'^{*}_{\text{rt}}(q_{\text{rt}_0}, t)$), and similarly we evaluate the same trace on the root monitor of $\mathcal{D}'$ (i.e., $\Delta'^{*}_{\text{rt}'}(q_{\text{rt}'_0}, t)$). The states reached for both of the automata executions need to be labeled by the same verdict. While both specifications yield the same verdict for a given trace, one could also extend this formulation to add bounds on delay.

Specification synthesis. Another interesting problem to explore is that of *specification synthesis*. Specification synthesis considers the problem of generating a decentralized specification, using various inputs. Typically, we would expect another specification as reference and possibly the system architecture. For example, given a centralized specification, we generate a decentralized specification, by splitting the specification into subspecifications and assigning the subspecifications to monitors. Generating a decentralized specification using a centralized one as reference is used in some algorithms such as choreography [15][25]. Starting from an LTL formula, the formula is split into subformulas hosted on the various components of the system (this is detailed further in Section 7.2). Given a decentralized specification $\mathcal{D}$, and a system graph $\langle C, E' \rangle$, the problem consists in generating a specification $\mathcal{D}'$. The variants of the synthesis problem depend on the properties that $\mathcal{D}'$ must have, we list (non-exhaustively) example properties:

[25]For more details see Appendix E.

(1) $\mathcal{D}'$ is monitorable (Section 6.1);
(2) $\mathcal{D}'$ is compatible with $\langle C, E' \rangle$ (Section 6.2);
(3) $\mathcal{D}'$ and $\mathcal{D}$ are verdict equivalent.

Synthesis problems could also be expanded to handle optimization techniques, with regards to specifications. The specification determines the computation and communication needed by the monitors. As such, it is possible to optimize, the size of automata, and references so as to fine-tune load and overhead for a given system architecture.

Extending THEMIS *metrics.* Moreover, one could consider creating new metrics for THEMIS to analyze more aspects of decentralized monitoring algorithms. We see that this is important, as in two specifications out of the five when using Chiron traces (Section 9.2), the choreography algorithm using a simple heuristic generated an inefficient decentralized specification. New metrics would be automatically instrumented on all existing algorithms and experiments could be easily replicated to compare them.

Weakening round constraints. In the current setting, all monitors of the decentralized specifications are required to perform their evaluation based on a global round. This condition can be weakened to a relation between only dependent monitors, instead of the whole system.

12 CONCLUSION

We present a general approach to monitoring decentralized specifications. A specification is a set of automata associated with monitors that are attached to various components. We provide a general decentralized monitoring algorithm defining the major steps needed to monitor such specifications. We make a clear distinction between the topology of monitors and the behavior of each monitor. We elaborate on two properties associated with decentralized specifications: compatibility, and monitorability. In addition, we present the EHE data structure which allows us to (i) aggregate monitor states with strong eventual consistency (ii) remain sound with respect to the execution of the monitor, and (iii) characterize the behavior of the algorithm at runtime. We then map three existing algorithms: Orchestration, Migration and Choreography to our approach using our data structures. We develop and use THEMIS to implement algorithms and analyze their behavior by designing new metrics. We implement four algorithms in THEMIS under our model and data-structures: orchestration (Orch), migration using earliest obligation (Migr), migration using round-robin (Migrr), and choreography (Chor). Using THEMIS and the designed metrics, we explore simulations of the four algorithms on two scenarios and validate the trends observed in the analysis. In the first scenario, we use the synthetic benchmark comprising of random specifications and traces. In the second scenario, we use a real example (Chiron) with existing formalized specifications.

REFERENCES

[1] Patterns Project. (1999). http://patterns.projects.cs.ksu.edu/.
[2] Patterns Project: List of specifications. (1999). http://patterns.projects.cs.ksu.edu/documentation/specifications/AFTER.raw.
[3] George S. Avrunin, James C. Corbett, Matthew B. Dwyer, Corina S. Pasareanu, and Stephen F. Siegel. 1999. *Comparing Finite-State Verification Techniques for Concurrent Software.* Technical Report.
[4] Ezio Bartocci. 2013. Sampling-based Decentralized Monitoring for Networked Embedded Systems. In *Proceedings Third International Workshop on Hybrid Autonomous Systems, HAS 2013, Rome, Italy, 17th March 2013. (EPTCS)*, Luca Bortolussi, Manuela L. Bujorianu, and Giordano Pola (Eds.), Vol. 124. 85–99. https://doi.org/10.4204/EPTCS.124.9
[5] Ezio Bartocci and Yliès Falcone (Eds.). 2018. *Lectures on Runtime Verification - Introductory and Advanced Topics.* Lecture Notes in Computer Science, Vol. 10457. Springer. https://doi.org/10.1007/978-3-319-75632-5
[6] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang.

2017. First international Competition on Runtime Verification: rules, benchmarks, tools, and final results of CRV 2014. *International Journal on Software Tools for Technology Transfer* (2017), 1–40. https://doi.org/10.1007/s10009-017-0454-5

[7] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. 2015. Failure-aware Runtime Verification of Distributed Systems. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India (LIPIcs)*, Prahladh Harsha and G. Ramalingam (Eds.), Vol. 45. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 590–603. https://doi.org/10.4230/LIPIcs.FSTTCS.2015.590

[8] Andreas Bauer and Yliès Falcone. 2016. Decentralised LTL monitoring. *Formal Methods in System Design* 48, 1-2 (2016), 46–93. https://doi.org/10.1007/s10703-016-0253-8

[9] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4 (2011), 14. https://doi.org/10.1145/2000799.2000800

[10] Andreas Klaus Bauer and Yliès Falcone. 2012. Decentralised LTL Monitoring. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings (Lecture Notes in Computer Science)*, Dimitra Giannakopoulou and Dominique Méry (Eds.), Vol. 7436. Springer, 85–100. https://doi.org/10.1007/978-3-642-32759-9_10

[11] Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. 2016. Challenges in Fault-Tolerant Distributed Runtime Verification, See [40], 363–370. https://doi.org/10.1007/978-3-319-47169-3_27

[12] David Buchfuhrer and Christopher Umans. 2008. The Complexity of Boolean Formula Minimization. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games (Lecture Notes in Computer Science)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.), Vol. 5125. Springer, 24–35. https://doi.org/10.1007/978-3-540-70575-8_3

[13] CERN. http://dst.lbl.gov/ACSSoftware/colt/. (1999). http://dst.lbl.gov/ACSSoftware/colt/.

[14] Christian Colombo and Yliès Falcone. 2014. Organising LTL Monitors over Distributed Systems with a Global Clock. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings (Lecture Notes in Computer Science)*, Borzoo Bonakdarpour and Scott A. Smolka (Eds.), Vol. 8734. Springer, 140–155. https://doi.org/10.1007/978-3-319-11164-3_12

[15] Christian Colombo and Yliès Falcone. 2016. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design* 49, 1-2 (2016), 109–158. https://doi.org/10.1007/s10703-016-0251-x

[16] Sylvain Cotard, Sébastien Faucou, Jean-Luc Béchennec, Audrey Queudet, and Yvon Trinquet. 2012. A Data Flow Monitoring Service Based on Runtime Verification for AUTOSAR. In *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012, Liverpool, United Kingdom, June 25-27, 2012*, Geyong Min, Jia Hu, Lei (Chris) Liu, Laurence Tianruo Yang, Seetharami Seelam, and Laurent Lefèvre (Eds.). IEEE Computer Society, 1508–1515. https://doi.org/10.1109/HPCC.2012.220

[17] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: Runtime Monitoring of Synchronous Systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, 166–174. https://doi.org/10.1109/TIME.2005.26

[18] Normann Decker, Philip Gottschling, Christian Hochberger, Martin Leucker, Torben Scheffel, Malte Schmitz, and Alexander Weiss. 2017. Rapidly Adjustable Non-intrusive Online Monitoring for Multi-core Systems. In *Formal Methods: Foundations and Applications*, Simone Cavalheiro and José Fiadeiro (Eds.). Springer International Publishing, Cham, 179–196.

[19] Volker Diekert and Martin Leucker. 2014. Topology, monitorable properties and runtime verification. *Theoretical Computer Science* 537 (2014), 29 – 41. https://doi.org/10.1016/j.tcs.2014.02.052 Theoretical Aspects of Computing (ICTAC 2011).

[20] Volker Diekert and Anca Muscholl. 2012. On Distributed Monitoring of Asynchronous Systems. In *Logic, Language, Information and Computation - 19th International Workshop, WoLLIC 2012, Buenos Aires, Argentina, September 3-6, 2012. Proceedings (Lecture Notes in Computer Science)*, C.-H. Luke Ong and Ruy J. G. B. de Queiroz (Eds.), Vol. 7456. Springer, 70–84. https://doi.org/10.1007/978-3-642-32621-9_5

[21] Alexandre Duret-Lutz. 2013. Manipulating LTL formulas using Spot 1.0. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13) (Lecture Notes in Computer Science)*, Vol. 8172. Springer, Hanoi, Vietnam, 442–445. https://doi.org/10.1007/978-3-319-02444-8_31

[22] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, Barry W. Boehm, David Garlan, and Jeff Kramer (Eds.). ACM, 411–420. https://doi.org/10.1145/302405.302672

[23] Antoine El-Hokayem and Yliès Falcone. 2017. Monitoring Decentralized Specifications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 125–135. https://doi.org/10.1145/3092703.3092723

[24] Antoine El-Hokayem and Yliès Falcone. 2017. THEMIS: A Tool for Decentralized Monitoring Algorithms. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17-DEMOS), Santa Barbara, CA, USA, July 2017*. https://doi.org/10.1145/3092703.3098224

[25] Antoine El-Hokayem and Yliès Falcone. THEMIS Website. (2017). https://gitlab.inria.fr/monitoring/themis.

[26] Antoine El-Hokayem and Yliès Falcone. THEMIS Article Artifact. (2018). https://gitlab.inria.fr/monitoring/themis-artifact-article.

[27] Yliès Falcone. 2010. You Should Better Enforce Than Verify. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings (Lecture Notes in Computer Science)*, Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.), Vol. 6418. Springer, 89–105. https://doi.org/10.1007/978-3-642-16612-9_9

[28] Yliès Falcone, Tom Cornebize, and Jean-Claude Fernandez. 2014. Efficient and Generalized Decentralized Monitoring of Regular Languages. In *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings (Lecture Notes in Computer Science)*, Erika Ábrahám and Catuscia Palamidessi (Eds.), Vol. 8461. Springer, 66–83. https://doi.org/10.1007/978-3-662-43613-4_5

[29] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. 2012. What can you verify and enforce at runtime? *STTT* 14, 3 (2012), 349–382. https://doi.org/10.1007/s10009-011-0196-8

[30] Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*, Manfred Broy, Doron a. Peled, and Georg Kalus (Eds.). NATO science for peace and security series, d: information and communication security, Vol. 34. ios press, 141–175. https://doi.org/10.3233/978-1-61499-207-3-141

[31] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. 2018. A Taxonomy for Classifying Runtime Verification Tools. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science)*, Christian Colombo and Martin Leucker (Eds.), Vol. 11237. Springer, 241–262. https://doi.org/10.1007/978-3-030-03769-7_14

[32] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. 2018. Runtime Failure Prevention and Reaction. See [5], 103–134. https://doi.org/10.1007/978-3-319-75632-5_4

[33] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. 2018. Runtime Verification for Decentralised and Distributed Systems. See [5], 176–210. https://doi.org/10.1007/978-3-319-75632-5_6

[34] Sylvain Hallé. 2016. When RV Meets CEP. In *Runtime Verification*, Yliès Falcone and César Sánchez (Eds.). Springer International Publishing, Cham, 68–91.

[35] Sylvain Hallé, Raphaël Khoury, and Sébastien Gaboury. 2017. Event Stream Processing with Multiple Threads. In *Runtime Verification*, Shuvendu Lahiri and Giles Reger (Eds.). Springer International Publishing, Cham, 359–369.

[36] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science)*, Jørgen Lindskov Knudsen (Ed.), Vol. 2072. Springer, 327–353. https://doi.org/10.1007/3-540-45337-7_18

[37] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. 1999. Formally specified monitoring of temporal properties. In *11th Euromicro Conference on Real-Time Systems (ECRTS 1999), 9-11 June 1999, York, England, UK, Proceedings*. IEEE Computer Society, 114–122. https://doi.org/10.1109/EMRTS.1999.777457

[38] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J. Log. Algebr. Program.* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004

[39] Martin Leucker, Malte Schmitz, and Danilo à Tellinghusen. 2016. Runtime Verification for Interconnected Medical Devices, See [40], 380–387. https://doi.org/10.1007/978-3-319-47169-3_29

[40] Tiziana Margaria and Bernhard Steffen (Eds.). 2016. *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*. Lecture Notes in Computer Science, Vol. 9953. https://doi.org/10.1007/978-3-319-47169-3

[41] Menna Mostafa and Borzoo Bonakdarpour. 2015. Decentralized Runtime Verification of LTL Specifications in Distributed Systems. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015*. IEEE Computer Society, 494–503. https://doi.org/10.1109/IPDPS.2015.95

[42] Aravind Natarajan, Himanshu Chauhan, Neeraj Mittal, and Vijay K. Garg. 2017. Efficient abstraction algorithms for predicate detection. *Theor. Comput. Sci.* 688 (2017), 24–48. https://doi.org/10.1016/j.tcs.2015.12.037

[43] Vinit A. Ogale and Vijay K. Garg. 2007. Detecting Temporal Logic Predicates on Distributed Computations. In *Distributed Computing, 21st International Symposium, DISC 2007, Proceedings (Lecture Notes in Computer Science)*, Andrzej Pelc (Ed.), Vol. 4731. Springer, 420–434. https://doi.org/10.1007/978-3-540-75142-7_32

[44] Amir Pnueli and Aleksandr Zaks. 2006. PSL Model Checking and Run-Time Verification Via Testers. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings (Lecture Notes in Computer Science)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.), Vol. 4085. Springer, 573–586. https://doi.org/10.1007/11813040_38

[45] Grigore Rosu and Klaus Havelund. 2005. Rewriting-Based Techniques for Runtime Verification. *Autom. Softw. Eng.* 12, 2 (2005), 151–197. https://doi.org/10.1007/s10515-005-6205-y

[46] Torben Scheffel and Malte Schmitz. 2014. Three-valued asynchronous distributed runtime verification. In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2014, Lausanne, Switzerland, October 19-21, 2014*. IEEE, 52–61. https://doi.org/10.1109/MEMCOD.2014.6961843

[47] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. 2004. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum (Eds.). IEEE Computer Society, 418–427. https://doi.org/10.1109/ICSE.2004.1317464

[48] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings (Lecture Notes in Computer Science)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.), Vol. 6976. Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[49] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. https://doi.org/10.1137/0201010

[50] The Chiron Team. Chiron User Interface. (1999). http://laser.cs.umass.edu/verification-examples/chiron/index.html.

[51] Prasanna Thati and Grigore Rosu. 2005. Monitoring Algorithms for Metric Temporal Logic Specifications. *Electronic Notes in Theoretical Computer Science* 113 (2005), 145 – 162. https://doi.org/10.1016/j.entcs.2004.01.029

[52] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181

[53] Gene T. J. Wuu and Arthur J. Bernstein. 1986. Efficient Solutions to the Replicated Log and Dictionary Problems. *Operating Systems Review* 20, 1 (1986), 57–66. https://doi.org/10.1145/12485.12491

A PROOFS

Proof of Proposition 4.8. The proof is by induction on the number of timestamps in the EHE, i.e., $n = |\text{rounds}(\mathcal{I})|$. Without loss of generality, we can assume the automaton being encoded is normalized (see Remark 1), that is, all shared edges between any two states are replaced by one edge which is labeled by the disjunction of their labels.

One could see that the base case only contains the initial state of an automaton, i.e., $\mathcal{I}^0 = [0 \mapsto q_0 \mapsto \top]$, and as such the proposition holds.

Let us consider $n = 2$, we have $\mathcal{I}^1 = \text{mov}(\mathcal{I}^0, 0, 1)$. To compute mov, we first consider $\text{next}(\mathcal{I}^0, 0)$ which considers all states reachable from q_0 as the only tuple in $\mathcal{I}^0$ is $\langle 0, q_0 \rangle$, i.e., $\text{next}(\mathcal{I}^0, 0) = \{q' \in Q \mid \exists e \in Expr : \delta(q_0, e) = q'\}$, we know that only one such e can evaluate to $\top$ for any memory encoded with the identity encoder (idt), since the automaton is deterministic. Let us collect all such states and their expressions as $P = \{\langle q', e \rangle \in Q \times Expr \mid \exists e \in Expr : \delta(q_0, e) = q'\}$. We note that $\mathcal{I}^1(0, q_0) = \top$ is the only entry for timestamp 0. The property holds trivially for that entry. We now consider the entries in $\mathcal{I}^1$ for timestamp 1. Each of tuple $\langle q', e \rangle \in P$ corresponds to the expression $\mathcal{I}^1(1, q')$, constructed with $\text{to}(\mathcal{I}^0, 0, q', \text{ts}_1) = \mathcal{I}^0(0, q_0) \wedge \text{ts}_1(e)$. We note that ts_1 only adds the timestamp 1 to each atomic proposition. As such, for any given memory encoded with ts_1 only one such expression can be evaluated to $\top$.

Inductive step: We assume that the property holds on $\mathcal{I}^{n-1}$ for some $n \in \mathbb{N}$, that is:
$\forall \mathcal{M} \in Mem, \forall t \in \text{rounds}(\mathcal{I}^{n-1}), \exists q \in Q : (\text{eval}(\mathcal{I}^{n-1}(t, q), \mathcal{M}) = \top) \implies (\forall q' \in Q \setminus \{q\} \implies$

$\text{eval}(\mathcal{I}^{n-1}(t,q'),\mathcal{M}) \neq \top$). Let us prove that the property holds for $\mathcal{I}^n$.

The approach is similar to that of $n=2$ using the recursive structure of the EHE to generalize. We decompose $\mathcal{I}^n$, using the definition of mov, as follows:

$$\mathcal{I}^n = \mathcal{I}^{n-1} \dagger_\vee \biguplus^{\vee}_{q' \in \text{next}(\mathcal{I}^{n-1},n)} \{n \mapsto q' \mapsto \text{to}(\mathcal{I}^{n-1}, n-1, q', \text{ts}_n)\}$$

We know that $\text{rounds}(\mathcal{I}^n) = \text{rounds}(\mathcal{I}^{n-1}) \cup \{n\}$. The induction hypothesis states that the property holds for all entries in $\mathcal{I}^{n-1}$ (i.e. for $t \in \text{rounds}(\mathcal{I}^{n-1})$), we consider the entries for timestamp n only. Since $\biguplus^{\vee}$ applies $\dagger_\vee$ on the entire set, and it is associative and commutative we consider the expression for a given state after all the merges, without consideration of order of merges. As such the states associated with timestamp n are computed using $\text{next}(\mathcal{I}^{n-1}, n)$. We have $\forall q' \in \text{next}(\mathcal{I}^{n-1}, n)$:

$$\begin{aligned}
\mathcal{I}^n(n,q') &= \text{to}(\mathcal{I}^{n-1}, n-1, q', \text{ts}_n) && \text{(Def. mov)}\\
&= \bigvee_{\{\langle q,e'\rangle \mid \delta(q,e')=q'\}} (\mathcal{I}^{n-1}(n-1,q) \wedge \text{ts}_n(e')) && (1)
\end{aligned}$$

(1) follows from the definition of to. If we examine the disjunction we notice using the induction hypothesis that there can only be a unique $q_\text{u} \in Q$ with $\mathcal{I}^{n-1}(n-1,q_\text{u})$ that evaluates to $\top$ at timestamp $n-1$. As such, the conjunction can only hold for one such q_u. Consequently, we can rewrite (1) by simplifying the disjunction and considering only states reachable from q_u, as the rest cannot evaluate to $\top$. Let us collect all such states and expressions in the set $P_\text{u} = \{\langle q', e'\rangle \mid q' \in \text{next}(\mathcal{I}^{n-1}, n) \wedge \exists e' \in Expr_{AP} : \delta(q_\text{u}, e') = q'\}$. The only entries that can still evaluate to $\top$ are:

$$\begin{aligned}
\forall \langle q', e'\rangle \in P_\text{u} : \mathcal{I}^n(n,q') &= \mathcal{I}^{n-1}(n-1,q_\text{u}) \wedge \text{ts}_n(e')\\
&= \text{ts}_n(e')
\end{aligned}$$

Since the automaton is deterministic, we know that we have one unique expression e_u that can evaluate to $\top$, given any memory encoded with idt. Since ts_n only adds the timestamp n to the atomic propositions without changing the expression, we deduce that only $\text{ts}_n(e_\text{u})$ evaluates to $\top$. As such, there is a unique expression that can evaluate to $\top$ for any given memory encoded with ts_n. Furthermore, we know that the expression has only been encoded with ts_n so when memories encoded with different timestamps or encoders are merged, they do not affect the evaluation of $\text{ts}_n(e_\text{u})$. As such, we have a unique entry $\mathcal{I}^n(n,q'_\text{u})$ s.t. $\delta(q_\text{u}, e_\text{u})$ that can evaluate to $\top$. Therefore:

$$\begin{aligned}
\forall \mathcal{M} \in Mem, \forall t \in \text{rounds}(\mathcal{I}^n), \exists q \in Q :&\\
(\text{eval}(\mathcal{I}^n(t,q),\mathcal{M}) = \top) &\implies\\
(\forall q' \in Q : q' \neq q \implies \text{eval}(\mathcal{I}^n(t,q'),\mathcal{M}) \neq \top)&
\end{aligned}$$

□

Lemma A.1 (Evaluation modulo encoding). *Given a trace* tr *of length i and a reconstructed global trace* $\rho(\text{tr}) = evt_1 \cdot \ldots \cdot evt_i$*, we consider two memories* $\mathcal{M}^i_\mathcal{A}$ *and* $\mathcal{M}^i$ *generated under different encodings. We consider* $\mathcal{M}^i_\mathcal{A} = \text{memc}(evt_i, \text{idt})$*, and* $\mathcal{M}^i = \biguplus^2_{t\in[1,i]}\{\text{memc}(evt_t, \text{ts}_t)\}$*. We show that*

an expression encoded using different encodings evaluates the same for the memories, that is:

$$\forall e \in Expr_{AP} : \text{eval}(\text{idt}(e), \mathcal{M}^i_{\mathcal{A}}) \Leftrightarrow \text{eval}(\text{ts}_i(e), \mathcal{M}^i).$$

Proof of Lemma A.1. We first note that for the first evaluation $\text{eval}(\text{idt}(e), \mathcal{M}^i_{\mathcal{A}})$, we rely only on the event evt_i since $\mathcal{M}^i_{\mathcal{A}} = \text{memc}(evt_i, \text{idt})$. This is not the case for $\text{eval}(\text{ts}_i(e), \mathcal{M}^i)$ as $\mathcal{M}^i = \biguplus^2_{t\in[1,i]}\{\text{memc}(evt_t, \text{ts}_t)\}$. However, we notice that for the second evaluation we evaluate the expression $\text{ts}_i(e)$, that is, where the expression where all atomic propositions have been encoded by the timestamp i. Therefore, let us denote the memory with timestamp i by $\mathcal{M}' = \text{memc}(evt_i, \text{ts}_i)$. We can rewrite $\mathcal{M}^i$ as follows:

$$\begin{aligned} \mathcal{M}^i &= \text{memc}(evt_i, \text{ts}_i) \quad \dagger_2 \biguplus^2_{t\in[1,k]}\{\text{memc}(evt_k, \text{ts}_k)\}, \\ &= \mathcal{M}' \qquad\qquad\quad \dagger_2 \biguplus^2_{t\in[1,k]}\{\text{memc}(evt_k, \text{ts}_k)\}. \end{aligned}$$

We know that all entries $\langle k, a\rangle \in \text{dom}(\mathcal{M}^i)$ with $k < i$ do not affect at all the evaluation of an expression encoded with ts_i. As such, we have:

$$\forall e \in Expr_{AP} : \text{eval}(\text{ts}_i(e), \mathcal{M}^i) \Leftrightarrow \text{eval}(\text{ts}_i(e), \mathcal{M}')$$

We now show that the two memories $\mathcal{M}^i_{\mathcal{A}}$ and $\mathcal{M}'$ contain simply an encoding of the same atomic propositions. By construction, we have the following:

$$\begin{aligned} \forall a \in \text{dom}(\mathcal{M}^i_{\mathcal{A}}) : &\quad \langle i, a\rangle \in \text{dom}(\mathcal{M}^i) \wedge\ \mathcal{M}^i_{\mathcal{A}}(a) = \mathcal{M}'(\langle i, a\rangle), \\ \forall \langle i, a'\rangle \in \text{dom}(\mathcal{M}') : &\quad a' \in \text{dom}(\mathcal{M}^i_{\mathcal{A}}) \wedge\ \mathcal{M}'(\langle i, a'\rangle) = \mathcal{M}^i_{\mathcal{A}}(a'). \end{aligned}$$

As such, we have: $\forall e \in Expr_{AP} : \text{eval}(\text{idt}(e), \mathcal{M}^i_{\mathcal{A}}) \Leftrightarrow \text{eval}(\text{ts}_i(e), \mathcal{M}') \Leftrightarrow \text{eval}(\text{ts}_i(e), \mathcal{M}^i)$.

□

Proof of Proposition 4.10. Given a trace tr of length i and a reconstructed global trace $\rho(\text{tr}) = evt_1 \cdot \ldots \cdot evt_i$, the proof is done by induction on the length of the trace $|\rho(\text{tr})|$. We omit the label ℓ for clarity.

Base case: $|\rho(\text{tr})| = 0, \rho(\text{tr}) = \epsilon$

$$\Delta^*(q_0, \epsilon) = q_0 = \text{sel}(\mathcal{I}^0, [\,], 0)$$

$$\mathcal{I}^0 = \text{mov}([0 \mapsto q_0 \mapsto \top], 0, 0) = [0 \mapsto q_0 \mapsto \top]$$

We only have expression $\top$ which is mapped to q_0 at $t = 0$. Expression $\top$ requires no memory to be evaluated.

Inductive step: We assume that the property holds for a trace of length i for some $i \in \mathbb{N}$, that is $\Delta^*(q_0, evt_1 \cdot \ldots \cdot evt_i) = \text{sel}(\mathcal{I}^i, \mathcal{M}^i, i) = q_i$. Let us prove that the property holds for any trace of length $i + 1$.

We now consider the transition functions in the automaton:

$$\begin{aligned} q_{i+1} &= \Delta^*(q_0, evt_1 \cdot \ldots \cdot evt_{i+1}) \\ &= \Delta(\Delta^*(q_0, evt_1 \cdot \ldots \cdot evt_i), evt_{i+1}) && \text{(Def. 4.2)} \\ &= \Delta(q_i, evt_{i+1}) && \text{(Hyp.)} \\ &\Leftrightarrow \exists e \in Expr_{AP} : \\ &\quad \delta(q_i, expr) = q_{i+1} \wedge \text{eval}(e, \mathcal{M}^{i+1}_{\mathcal{A}}) = \top && (1) \end{aligned}$$

We note that, since the automaton is deterministic, there is a unique q_{i+1} such that $q_{i+1} = \Delta(q_i, evt_{i+1})$.

We now consider the EHE operations to reach q_{i+1} from q_i.

$$
\begin{aligned}
& q_i = \text{sel}(\mathcal{I}^i, \mathcal{M}^i, i) \\
& \quad \Leftrightarrow e = \mathcal{I}^i(i, q_i) \text{ with } \text{eval}(e, \mathcal{M}^i) = \top && (2) \\
& \qquad \wedge \forall q_i' \in Q : q_i' \neq q_i \implies \text{eval}(\mathcal{I}^i(i, q_i')) \neq \top && (\text{Prop. } 4.8) \\
& \quad \Leftrightarrow \text{to}(\mathcal{I}^i, i, q_{i+1}, \text{ts}_{i+1}) = \top && (3)
\end{aligned}
$$

(3) From the induction hypothesis, we know that $\mathcal{I}^i(i, q_i) = \top$, thus:

$$
\begin{aligned}
& \text{to}(\mathcal{I}^i, i, q_{i+1}, \text{ts}_{i+1}) \\
& \quad = \bigvee_{\{\langle q, e' \rangle | \delta(q, e') = q'_{i+1}\}} (\mathcal{I}^i(i, q) \wedge \text{ts}_{i+1}(e')) \\
& \quad = \bigvee_{\{\langle q, e'' \rangle | \delta(q, e'') = q'_{i+1} \wedge q \neq q_i\}} (\mathcal{I}^i(i, q) \wedge \text{ts}_{i+1}(e'')) \\
& \qquad \vee \bigvee_{\{\langle q_i, e''' \rangle | \delta(q_i, e''') = q'_{i+1}\}} (\text{ts}_{i+1}(e''')).
\end{aligned}
$$

We split the disjunction to consider the expressions that only come from state q_i, we now show that one such expression evaluates to $\top$. We know from (1), that one such expression can be taken in the automaton:

$$
\begin{aligned}
& \exists e \in Expr_{AP} : \delta(q_i, e) = q_{i+1} \wedge \text{eval}(e, \mathcal{M}_{\mathcal{A}}^{i+1}) = \top && (1) \\
& \qquad \Leftrightarrow \text{eval}(\text{ts}_{i+1}(e), \mathcal{M}^{i+1}) = \top && (4) \\
& \qquad \Leftrightarrow \text{to}(\mathcal{I}^i, i, q_{i+1}, \text{ts}_{i+1}) = \top && (5)
\end{aligned}
$$

(4) is obtained using Lemma A.1 and $\text{idt}(e) = e$.
(5) follows from the disjunction.
Using the same approach, we can show that $\forall q' \in \text{next}(\mathcal{I}^i, i) : q' \neq q_{i+1} \implies \text{to}(\mathcal{I}^i, i, q', \text{ts}_{i+1}) \neq \top$, since the first part of the conjunction does not evaluate to $\top$, and we know that the second part cannot evaluate to $\top$ by (2).
Finally, $\text{to}(\mathcal{I}^i, i, q_{i+1}, \text{ts}_{i+1}) = \top$ iff $\text{sel}(\mathcal{I}^{i+1}, \mathcal{M}^{i+1}, i+1) = q_i$.

□

B CHIRON SYSTEM ATOMIC PROPOSITIONS

We broke down the Chiron system based on analysis of the examples provided in [22, 50], using the various specifications rewritten in [2]. Table 6 displays various associations we used to generate our traces and events. Column **C** assigns an ID to the component. Column **Name** lists the logical module of the system we considered as a component. Column **Original (AP)** lists the atomic proposition provided by the authors of Chiron, and then edited by [22]. Column **AP** maps the atomic proposition to our traces. Column **Comments** includes comments on the atomic propositions.

Table 6. Chiron Atomic Propositions and Components.

C	Name	Original (AP)	AP	Comments
A	Dispatcher	registered_event_a1_e1	a0	True after an artist has completed registration
		registered_event_a1_e2	a1	
		registered_event_a2_e1	a2	
		registered_event_a2_e2	a3	
		notify_a1_e1	a4	True when starting to dispatch an event to an artist
		notify_a1_e2	a5	
		notify_a2_e1	a6	
		notify_a2_e2	a7	
		lst_sz0_e1	a8	Tracking the size of the list (state)
		lst_sz0_e2	a9	
		lst_gt2_e1	a10	
		lst_gt2_e2	a11	
B	Artist1	notify_client_a1_e1	b0	Artist receives a notification
		notify_client_a1_e2	b1	
		register_event_a1_e1	b2	Artist requests to register for event
		register_event_a1_e2	b3	
		unregister_event_a1_e1	b4	Artist requests to unregister
		unregister_event_a1_e2	b5	
C	Artist2	notify_client_a2_e1	c0	See Artist1
		notify_client_a2_e2	c1	
		register_event_a2_e1	c2	
		register_event_a2_e2	c3	
		unregister_event_a2_e1	c4	
		unregister_event_a2_e2	c5	
D	Main	term	d0	Main program terminates

C CHANGES IN THEMIS

Figure 14 shows the data transferred for the migration algorithms, which is associated with the size of the EHE. We reduced the size of the EHE by making calls to the simplifier only for complex simplifications, and implement the basic Boolean simplification while traversing the expression to replace atomic propositions by looking up the memory (in the operation rw). Since we have less calls, we apply a more aggressive simplification that is more costly[26], but also reduces the size of the expressions. The x-axis indicates the algorithm's variant and the number of components, where Migr (resp. Migrr) stands for earliest the variant obligation (resp. round-robin). The y-axis is presented in logarithmic scale. We notice a significant drop in the size of EHE, dropping from 769 in the ISSTA'17 version for Migrr-5 to 73.12.

D DETAILED COMPARISON

Tables 7 and 8 present the detailed comparison for the synthetic scenario and Chiron, respectively. The metrics presented are (in order of columns): average information delay (δ_t), normalized average number of messages (#Msgs), normalized data transferred (Data), maximum simplifications done by any given monitor per run, averaged across all runs (S_{max}), normalized critical simplifications

[26]We use `ltlfilt` from [21] with `-boolean-to-isop` to rewrite Boolean subformulas as irredundant sum of products.

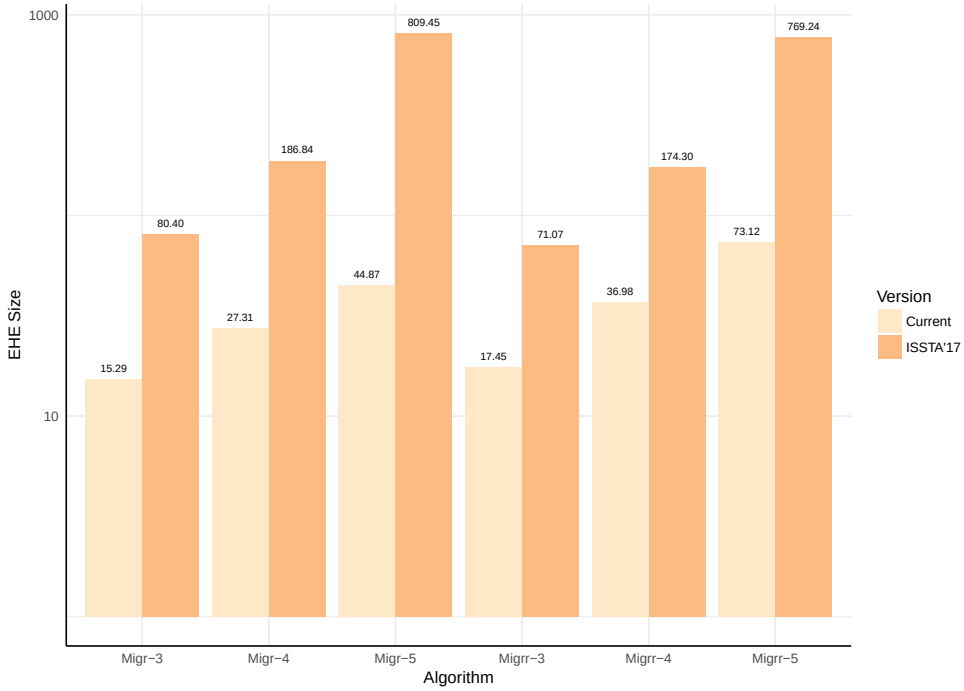


Fig. 14. Size of EHE.

Table 7. Synthetic Benchmark. Cells contains mean and standard deviation in parentheses.

Alg.	$\|C\|$	δ_t	#Msgs	Data	S_{max}	S_{crit}	$Conv_E$
Orch	3	0.48 (0.50)	2.44 (0.61)	31.84 (8.07)	0.00 (0.00)	0.00 (0.00)	0.65 (0.02)
	4	0.53 (0.50)	3.85 (0.94)	50.05 (12.39)	0.00 (0.00)	0.00 (0.00)	0.74 (0.02)
	5	0.64 (0.48)	5.30 (1.16)	69.20 (15.55)	0.00 (0.00)	0.00 (0.00)	0.79 (0.02)
	6	0.69 (0.46)	7.04 (1.50)	91.86 (20.02)	0.00 (0.00)	0.00 (0.00)	0.83 (0.02)
Migr	3	0.58 (0.58)	0.27 (0.32)	8.46 (15.32)	4.72 (4.41)	3.08 (2.66)	0.65 (0.02)
	4	0.71 (0.67)	0.32 (0.34)	17.45 (35.87)	6.10 (6.17)	4.03 (3.75)	0.73 (0.03)
	5	0.96 (0.71)	0.43 (0.34)	30.41 (56.68)	7.41 (6.18)	4.97 (3.76)	0.79 (0.03)
	6	1.19 (0.86)	0.50 (0.34)	98.80 (244.94)	10.09 (8.32)	6.74 (4.87)	0.82 (0.04)
Migrr	3	0.76 (0.69)	0.78 (0.33)	14.51 (18.40)	5.62 (4.99)	3.51 (2.93)	0.65 (0.02)
	4	1.02 (0.90)	0.76 (0.36)	31.76 (51.55)	7.64 (7.16)	4.58 (4.04)	0.74 (0.03)
	5	1.39 (1.04)	0.75 (0.35)	62.83 (91.89)	9.70 (7.88)	5.70 (4.25)	0.79 (0.03)
	6	1.72 (1.19)	0.70 (0.37)	180.35 (360.25)	12.56 (9.76)	7.35 (5.14)	0.82 (0.03)
Chor	3	1.47 (1.99)	2.79 (1.10)	24.98 (9.85)	60.22 (242.88)	12.27 (6.55)	0.16 (0.12)
	4	1.36 (1.52)	3.84 (1.23)	34.36 (10.94)	44.71 (184.05)	12.95 (5.98)	0.13 (0.12)
	5	1.41 (1.55)	4.63 (1.37)	41.17 (12.16)	44.06 (223.15)	12.68 (6.06)	0.12 (0.11)
	6	1.29 (1.38)	5.87 (1.66)	52.09 (14.77)	38.35 (215.27)	13.01 (6.01)	0.13 (0.12)

(S_{crit}), and convergence based on expressions evaluated ($Conv_E$). For more details on the metrics, see Section 9.

Table 8. Metrics for Chiron traces. Cells contains mean and standard deviation in parentheses.

Alg.	Spec	δ_t	#Msgs	Data	S_{max}	S_{crit}	$Conv_E$
Orch	1	0.77 (0.42)	2.95 (0.00)	87.46 (0.00)	0.00 (0.00)	0.00 (0.00)	0.74 (0.00)
	2	1.00 (0.00)	2.95 (0.00)	87.46 (0.00)	0.00 (0.00)	0.00 (0.00)	0.74 (0.00)
	3	0.99 (0.10)	3.42 (0.02)	101.62 (1.00)	0.00 (0.00)	0.00 (0.00)	0.75 (0.00)
	5	0.94 (0.24)	2.95 (0.00)	87.46 (0.00)	0.00 (0.00)	0.00 (0.00)	0.74 (0.00)
	15a	1.00 (0.00)	2.95 (0.00)	87.46 (0.00)	0.00 (0.00)	0.00 (0.00)	0.74 (0.00)
	15b	1.00 (0.00)	3.00 (0.01)	88.90 (0.16)	0.00 (0.00)	0.00 (0.00)	0.75 (0.00)
Migr	1	1.66 (0.03)	0.02 (0.00)	0.52 (0.00)	8.00 (0.00)	2.03 (0.00)	0.74 (0.00)
	2	1.00 (0.00)	0.57 (0.00)	13.09 (0.10)	4.00 (0.00)	3.10 (0.01)	0.74 (0.00)
	3	1.86 (0.00)	0.88 (0.01)	70.23 (1.00)	13.00 (0.00)	9.76 (0.05)	0.75 (0.00)
	5	1.67 (0.00)	0.02 (0.00)	0.52 (0.00)	8.00 (0.00)	2.03 (0.00)	0.74 (0.00)
	15a	1.00 (0.00)	0.97 (0.00)	10.71 (0.04)	4.00 (0.00)	3.90 (0.00)	0.74 (0.00)
	15b	1.00 (0.00)	1.00 (0.00)	19.36 (0.35)	9.02 (2.00)	7.00 (0.03)	0.75 (0.00)
Migrr	1	1.98 (0.00)	1.01 (0.01)	50.19 (0.33)	9.80 (1.37)	5.29 (0.07)	0.74 (0.00)
	2	1.81 (0.02)	1.01 (0.01)	212.55 (3.17)	12.00 (0.00)	5.82 (0.05)	0.74 (0.00)
	3	2.37 (0.03)	0.89 (0.02)	147.00 (3.97)	15.97 (0.22)	10.65 (0.09)	0.75 (0.01)
	5	1.98 (0.01)	1.01 (0.00)	50.16 (0.30)	9.80 (1.35)	5.28 (0.06)	0.74 (0.00)
	15a	1.99 (0.01)	1.01 (0.01)	83.80 (0.34)	8.64 (0.94)	4.91 (0.01)	0.74 (0.00)
	15b	2.50 (0.01)	1.00 (0.00)	136.05 (0.27)	16.86 (0.35)	11.41 (0.01)	0.75 (0.00)
Chor	1	1.01 (0.00)	4.89 (0.00)	44.02 (0.00)	20.00 (0.00)	15.88 (0.32)	0.20 (0.01)
	2	133.86 (0.17)	2.95 (0.00)	26.52 (0.00)	2798.38 (211.11)	21.41 (0.74)	0.67 (0.02)
	3	1.22 (0.04)	4.40 (0.13)	39.64 (1.16)	23.65 (0.87)	18.18 (0.93)	0.33 (0.02)
	5	1.01 (0.00)	4.89 (0.00)	44.02 (0.00)	20.00 (0.00)	15.85 (0.33)	0.20 (0.01)
	15a	1.00 (0.00)	0.98 (0.00)	8.84 (0.00)	10.00 (0.00)	9.25 (0.07)	0.47 (0.01)
	15b	116.52 (1.19)	2.00 (0.00)	18.00 (0.00)	3387.04 (316.16)	28.01 (1.13)	0.71 (0.00)

E CHOREOGRAPHY SETUP PHASE

Choreography as presented in [15] splits the initial LTL formula into subformulas and delegates each subformula to a monitor on a component. Thus choreography presents a complicated *setup* phase. In this section, we present the *setup* phase. As such, we present the generation of the decentralized specification from a start LTL formula.

Choreography begins by taking the main formula, then deciding to split it into subformulas. Each monitor will monitor the subformula, notify other monitors of its verdict, and when needed *respawn* . Recall from the definition of Δ' (see Definition 5.4), that monitoring is recursively applied to the remainder of a trace starting at the current event. That is, initially we monitor from e_0 to e_n and then from e_1 to e_n and so forth. To do so, it is necessary to reset the state of a monitor appropriately, this process is called in [15] a *respawn*. Once the subformulas are determined, we generate an automaton per subformula to monitor it. Then, we construct the network of monitors in the form of a tree, in which the root is the main monitor. Verdicts for each subformula are then propagated in the hierarchy until a verdict can be reached by the root monitor.

A choreography monitor is a tuple $\langle id, \mathcal{A}_{\varphi_{id}}, ref_{id}, coref_{id}, respawn_{id} \rangle$ where:

- id denotes the monitor unique identifier (label);
- $\mathcal{A}_{id}$ the automaton that monitors the subformula;
- $ref_{id} : 2^{\text{Mons}}$ the monitors that this monitor should notify of a verdict;
- $coref_{id} : 2^{\text{Mons}}$ the monitors that send their verdicts to this monitor;

- $respawn_{id} : \mathbb{B}_2$ specifies whether the monitor should *respawn*;

To account for the verdicts from other monitors, the set of possible atoms is extended to include the verdict of a monitor identified by its id. Therefore, $Atoms = (\mathbb{N} \times AP) \cup (\text{Mons} \times \mathbb{N})$. Monitoring is done by replacing the subformula by the id of the monitor associated with it.

Before splitting a formula, it is necessary to determine the component that hosts its monitor. The component score is computed by counting the number of atomic propositions associated with a component in the subformula.

$$
\begin{aligned}
\text{scor}(\varphi, c) : LTL \times C \to \mathbb{N} & \\
= \ & \texttt{match } \varphi \texttt{ with} \\
| \ a \in AP \quad & \to \begin{cases} 1 & \text{if lu}(a) = c \\ 0 & \text{otherwise} \end{cases} \\
| \ \text{op } \phi \quad & \to \text{scor}(\phi, c) \\
| \ \phi \text{ op } \phi' \quad & \to \text{scor}(\phi, c) + \text{scor}(\phi', c)
\end{aligned}
$$

The chosen component is determined by the component with the highest score, using $\text{chc} : LTL \to C$:

$$\text{chc}(\varphi) = \underset{c \in C}{\text{argmax}}(\text{scor}(\varphi, c))$$

In order to setup the network of monitors, firstly the LTL expression is split into subformulas and the necessary monitors are generated to monitor each subformula. The tree of monitors is generated by recursively splitting the formula at the binary operators. We present the *setup* phase as a tree traversal of the LTL formula to generate the monitor network, merging nodes at each operator, which is a different flavor of the generation procedure in [15]. Given the two operands, we choose which operands remains in the host component, and (if necessary) which would be placed on a different component. Therefore, we add the constraint that at least one part of the LTL expression must still remain in the same component. Given two formulas φ and φ' and an initial base component c_b we determine the two components that should host φ and φ' with the restriction that one of them is c_b:

$$
\begin{aligned}
c_1 = \text{chc}(\varphi), \quad & c_2 = \text{chc}(\varphi') \\
s_1 = \text{scor}(\varphi, c_b), \quad & s_2 = \text{scor}(\varphi', c_b) \\
\text{split}(\varphi, \varphi', c_b) = & \begin{cases} \langle c_b, c_b \rangle & \text{if } c_1 = c_2 = c_b \\ \langle c_1, c_b \rangle & \text{if } (c_1 \neq c_b) \\ & \qquad \wedge (c_2 = c_b \vee s_2 > s_1) \\ \langle c_b, c_2 \rangle & \text{otherwise} \end{cases}
\end{aligned}
$$

Algorithm 3 displays the procedure to split the formula. For each binary operator, we determine which of the operands needs to be hosted in a new component. The result is a tuple: $\langle root, N, E \rangle$ where:

- *root* is the root of the tree;
- N is the set of generated monitor data;
- E is the set of edges between the monitors.

Monitor data is a pair $\langle id, spec \rangle$ that represents the id of the monitor and the formula that it monitors.

- First, chc determines the `host` component where the root monitor resides.
- Second, the AST of the LTL formula is traversed using netx, which splits on binary operators.

Algorithm 3 Setting up the monitor tree

```
1:  procedure NET_CHOR(φ, C, M)
2:      id ← 0
3:      c_h ← chc(φ)
4:      ⟨root, mons, edges⟩ ← netx(φ, id, c_h)
5:      return ⟨{root} ∪ mons, edges⟩
6:  end procedure
7:  procedure netx(φ, id_c, c_h)
8:      if φ ∈ AP then
9:          m ← ⟨φ, id_c⟩
10:         return ⟨m, ∅, ∅⟩
11:     else if φ matches op e then                          ▷ Unary Operator
12:         o ← netx(e, id_c, c_h)
13:         m ← ⟨op o.f, id_c⟩
14:         return ⟨m, o.N, o.E⟩
15:     else if φ matches e op e′ then
16:         ⟨c_1, c_2⟩ ← split(e, e′, c_h, M)
17:         if c_1 = c_2 = c_h then                          ▷ No Split
18:             l ← netx(e, id_c, c_h)
19:             r ← netx(e′, id_c, c_h)
20:             m ← ⟨l.f op r.f, id_c⟩
21:             return ⟨m, l.N ∪ r.N, l.E ∪ r.E⟩
22:         else if c_1 = c_h then                           ▷ Split Right Branch
23:             id_n ← newid()
24:             l ← netx(e, id_c, c_h)
25:             r ← netx(e′, id_n, c_2)
26:             m ← ⟨l.f op ⟨id_n⟩, id_c⟩
27:             return ⟨m, (l.N ∪ r.N ∪ r.root), (l.E ∪ r.E ∪ {⟨id_n, id_c⟩})⟩
28:         else                                             ▷ Split Left Branch
29:             id_n ← newid()
30:             l ← netx(e, id_n, c_1)
31:             r ← netx(e′, id_c, c_h)
32:             m ← ⟨⟨id_n⟩ op r.f, id_c⟩
33:             return ⟨m, (l.N ∪ r.N ∪ l.root), (l.E ∪ r.E ∪ {⟨id_n, id_c⟩})⟩
34:         end if
35:     end if
36: end procedure
```

- If both formulae can be monitored with the same monitor it does not split.
- Otherwise

(1) We recurse on the side kept, to further split the formula;
(2) We recurse on the side split, with a new *host* and *id*;
(3) We merge the subnetworks by:
  (a) Generating the host monitor with the formula resulting from the recursion;
  (b) Connecting the split branch's root monitor to the current host monitor;
  (c) Adding the split branch's root monitor to the set of additional monitors;
  (d) Merging the set of additional monitors and edges from both branches.

Once the monitor data tree is created, monitors are created accordingly, generating an automaton for the subformula, where some of its atomic propositions have been replaced with monitor ids. Each monitor is initialized with the refs and corefs set based on the edges setup.

Remark 2 (Compacting the network). *The monitor network can further be compacted as follows; monitors with the same subformula are merged into one, and their refs and corefs will be the result of the set union. However, one or more merged monitor will have to replace all occurrences of the id of the other monitors in all subformulas of all monitors.*