Monitoring Decentralized Specifications

Antoine El-Hokayem
Univ. Grenoble Alpes, Inria, CNRS,
Laboratoire d’Informatique de Grenoble
F-38000 Grenoble, France
antoine.el-hokayem@univ-grenoble-alpes.fr

Yliès Falcone
Univ. Grenoble Alpes, Inria, CNRS,
Laboratoire d’Informatique de Grenoble
F-38000 Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr

ABSTRACT
We define two complementary approaches to monitor decentralized systems. The first relies on those 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 decentralized monitorability, and present a general algorithm for monitoring decentralized specifications. We map three existing algorithms to our approaches and provide a framework for analyzing their behavior. Lastly, we introduce our tool, which is a framework for designing such decentralized algorithms, and simulating their behavior.

KEYWORDS
Runtime Verification, Monitoring, Decentralized Specification, Monitorability, Eventual Consistency

ACM Reference format:
DOI: 10.1145/3092703.3092723

1 INTRODUCTION
Runtime Verification (RV) [15, 18] is a lightweight formal method which consists in verifying that a run of a system is correct wrt 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 blackbox 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 the compliance of the system to the specification. RV techniques have been used for instance in the context of decentralized automotive [8] and medical [19] systems. In both cases, RV is used to verify correct communication 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 trying to scale existing approaches to decentralized systems, that is, systems with multiple components having no central observation point.

Challenges. Several algorithms have been designed [4, 5, 7, 13] and used [1] 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 [4, 7, 22] 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 depending on the monitor. 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 [14, 17, 21]) for centralized specifications exist [3, 9, 14], but do not extend to decentralized specifications. For example by splitting an LTL formula ad-hoc, it is possible to obtain a non-monitorable subformula1 which interferes with the completeness of a monitoring algorithm.

Contributions. In this paper, we tackle the presented challenges using two complementary approaches. We first lay out the basic blocks, by introducing our basic data structure, and the basic notions of monitoring with expressions in Sec. 3. Then, we present our first approach, a middle ground between rewriting and automata evaluation by introducing the Execution History Encoding (EHE) data structure in Sec. 4. We restrict the rewriting to boolean expressions, 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 [25]. In Sec. 5, we shift the focus on studying decentralized specifications and their properties, we define their semantics, interdependencies and study their monitorability. We aim at abstracting the high-level steps of decentralized monitoring. By identifying these steps, we elaborate a

---

1We use the example from [7]. GF(a) ∧ ¬(GF(a)) is monitorable, but its subformulae are both non-monitorable.
general decentralized monitoring algorithm. We view a decentralized system as a set of components \( C \). We associate \( n \) monitors to these components with the possibility of two or more monitors being associated to a component. We see a decentralized specification as a set of \( n \) finite-state automata with specific properties. Each automaton is associated with a monitor. Therefore, we generalize monitoring algorithms to multiple monitors. Therefore, we present a general decentralized monitoring algorithm that uses two high level steps: setup and monitor. Monitoring a centralized system can be seen as a special case with one component, one specification, and one monitor. Additionally, 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. In Sec. 6, 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 Sec. 7, 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. In Sec. 8, we use THEMIS to create new metrics related to load-balancing and our data structures, and to experimentally verify our analysis. Finally, we conclude and present future work in Sec. 9.

2 RELATED WORK

Several approaches have been taken to handle decentralized monitoring focusing on different aspects of the problem. The first class of approaches consists in monitoring by LTL formula rewriting [4, 7, 22]. Given an LTL formula specifying the system, a monitor will rewrite the formula based on information it has observed or received from other monitors. 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 [26] 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, \( \neg (G(T)) \) could be rewritten \( \top \land G(T) \) and will keep growing in function of the number of timestamps. To tackle the unpredictability of rewriting LTL formulae, another approach [13] uses automata for monitoring regular languages, and therefore (i) can express richer specifications, and (ii) has more predictable runtime behavior. This approach however focuses on a centralized specification.

Another class of research focuses on handling a different problem that arises in distributed systems. In [5], 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 monitors that 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 eventual delivery of messages. Similar work [2] 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 [2, 4, 7], and other works [10, 23, 24] do in effect introduce a decentralized specification. These approaches define separate monitors with different specifications, typically consisting of splitting the formula into subformulae. Then, they describe the collaboration between such monitors. However, they 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.

3 COMMON NOTIONS

We begin by introducing the \( \text{dict} \) data structure used to build more complex data structures in Sec. 3.1. Then, we introduce the basic concepts for decentralized monitoring in Sec. 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. We would like in addition, 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) [25], they are known as state-based convergent replicated data-types (CvRDTs). We use a dictionary data structure \( \text{dict} \) as our basic building block that maps a key to a value. \( \text{dict} \) supports two operations: \( \text{query} \) and \( \text{merge} \). The merge operation is the only operation that modifies \( \text{dict} \). The modifications never remove entries, the state of \( \text{dict} \) is then monotonically increasing. By ensuring that \( \text{merge} \) is idempotent, commutative, and associative we fulfill the necessary conditions for our data structure to be a CvRDT.

**PROPOSITION 3.1.** Data structure \( \text{dict} \) with operations \( \text{query} \) and \( \text{merge} \) is a CvRDT.

We model \( \text{dict} \) as a partial function \( f \). The keys are the domain of \( f \), i.e., \( \text{dom}(f) \) and values are mapped to each entry of the domain. The \( \text{query} \) operation checks if a key \( k \in \text{dom}(f) \) and returns \( f(k) \). If \( k \notin \text{dom}(f) \), then it is undefined. The \( \text{merge} \) operation of a \( \text{dict} \) \( f \) with another \( \text{dict} \) \( g \), is modeled as function composition. Two partial functions \( f \) and \( g \) are composed using operator \( \uparrow_{op} \) where \( op : (\text{dom}(f) \times \text{dom}(g)) \rightarrow \text{dom}(f) \).
We recall the basic building blocks of monitoring. We consider the set of outcomes \( B = \{ \top, \bot, ? \} \) to denote the verdicts true, false, not reached (or inconclusive) respectively. A verdict from \( B \) is a final verdict. Given a set of atomic propositions \( AP \), we define an encoding of the atomic propositions as \( Atoms \), this encoding is left to the monitoring algorithm to specify. \( Expr \) (resp. \( Expr_{AP} \)) denotes the set of boolean expressions over \( Atoms \) (resp. \( AP \)). When omitted, \( Expr \) refers to \( Expr_{AP} \). An encoder is a function \( enc : Expr_{AP} \rightarrow Expr_{Atoms} \) that encodes the atomic propositions into atoms. In this paper, we use two encoders: \( \text{idt} \) which is the identity function (it does not modify the atomic proposition), and \( ts \) which adds a timestamp \( t \) to each atomic proposition. A decentralized monitoring algorithm requires retaining, retrieving and communicating observations.

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

**Example 3.3 (Event).** The event \( \{(a, \top), (b, \bot)\} \) over \( AP = \{a, b\} \) indicates that the proposition \( a \) has been observed to be true, while \( b \) has been observed to be false.

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

Events are commonly stored in a monitor memory with some encoding (e.g., adding a timestamp). An event can be converted to a memory by encoding the atomic propositions to atoms, and associating their truth value:

\[
memc : 2^{AP \times B} \times (Expr_{AP} \rightarrow Expr_{Atoms}) \rightarrow Mem.
\]

**Example 3.5 (Memory).** Let \( e = \{(a, \top), (b, \bot)\} \) be an event at \( t = 1 \), the resulting memories using our encoders are:

\[
\begin{align*}
memc(e, \text{idt}) &= \{(a \mapsto \top, b \mapsto \bot)\}, \\
memc(e, ts) &= \{(1, a) \mapsto \top, (1, b) \mapsto \bot\}.
\end{align*}
\]

If we impose that \( Atoms \) be a totally ordered set, then two memories \( M_1 \) and \( M_2 \) can be merged by applying the operator \( \triangledown \). The total ordering is needed for the operator \( \triangledown \). 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 memory is a dict and \( \triangledown \) is idempotent, associative, and commutative, it follows from Prop. 3.1 that it is a CvRDT.

**Corollary 3.6.** A memory with operation \( \triangledown \) is a CvRDT.

In this paper, we perform monitoring by manipulating expressions in \( Expr \). The first operation we provide is \( rw \), which rewrites the expression to attempt to eliminate \( Atoms \).

**Definition 3.7 (Rewriting).** An expression \( expr \) is rewritten with a memory \( M \) using \( rw(expr, M) \) defined as follows:

\[
\begin{align*}
\text{rw} : Expr \times Mem &\rightarrow Expr \\
\text{rw}(expr, M) &= \text{match expr with} \\
| a &\in Atoms \rightarrow \{ M(a) \text{ if } a \in \text{dom}(M) \} \\
| \neg e &\rightarrow \text{rw}(e, M) \\
| e_1 \land e_2 &\rightarrow \text{rw}(e_1, M) \land \text{rw}(e_2, M) \\
| e_1 \lor e_2 &\rightarrow \text{rw}(e_1, M) \lor \text{rw}(e_2, M)
\end{align*}
\]

Using information from a memory \( M \), the expression is rewritten by replacing atoms with a final verdict (a truth value in \( B \)) in \( M \) when possible. Atoms that are not associated with a final verdict are kept in the expression. The operation \( rw \) yields a smaller formula to work with and repeatedly evaluate.

**Example 3.8 (Rewriting).** We consider \( M = \{a \mapsto \top, b \mapsto \bot\} \) and \( e = (a \lor b) \land c \). We have \( M(a) = \top, M(b) = \bot, M(c) = ? \). Since \( c \) is associated with \( ? \notin B \) then it will not be replaced. The resulting expression is \( \text{rw}(e, M) = (\top \lor \bot) \land c \).

We eliminate additional atoms using boolean logic. We denote by \( \text{simplify(expr)} \) the simplification of formula \( expr \).

**Example 3.9 (Simplification).** Consider \( M = \{a \mapsto \top\} \) and \( e = (a \land b) \lor (a \land \neg b) \). We have \( e' = \text{rw}(e, M) = (b \lor \neg b) \). Atoms can be eliminated with \( \text{simplify}(e') \). We finally get \( \top \).

We combine both rewriting and simplification in the eval function which determines a verdict from an expression \( expr \).

**Definition 3.10 (Evaluating an expression).** The evaluation of an boolean expression \( expr \in Expr \) using a memory \( M \) yields a verdict.

\[
\text{eval}(expr, M) = \begin{cases} 
\top & \text{if } e' \mapsto \top \\
\bot & \text{if } e' \mapsto \bot \\
? & \text{otherwise}
\end{cases}
\]

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

\footnotesize{This is also known as The Minimum Equivalent Expression problem [6].}
A decentralized system is as set of components $C$, we associate a sequence of events to each component using a decentralized trace function.

**Definition 3.12 (Decentralized trace).** A decentralized trace is a function $tr : \mathbb{N} \times C \rightarrow 2^{AP \times B_2}$.

Function $tr$ assigns an event to a component for a given timestamp. We additionally define function $lu : AP \rightarrow C$ to associate an observation to a component\(^3\).

$$lu(ap) = c \text{ s.t. } \exists t \in \mathbb{N}, \exists e \in B_2 : (ap, e) \in tr(t, c).$$

We consider timestamp 0 to be associated with the initial state, therefore our traces start at 1. Thus, a finite trace of length $n$ is a function $tr$ with a domain $[1, n]$. An empty trace has length 0 and is denoted by $\emptyset$. While $tr$ gives us a view of what components can locally see, we reconstruct the global trace to reason about all observations.

**Definition 3.13 (Reconstructing a global trace).** Given a decentralized trace $tr$ of length $n$, the global trace $\rho(tr) = e_1 \cdot \ldots \cdot e_n$ is s.t. $\forall i \in [1, n] : e_i = \bigcup_{c \in C} tr(i, c)$.

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, $\rho(\emptyset) = \emptyset$.

## 4 CENTRALIZED SPECIFICATIONS

We now focus on a decentralized system specified by one global automaton. The automaton is similar to automata defined for monitoring LTL\(_3\). This has been the topic of a lot of the Runtime Verification literature, we focus on adapting the approach 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 EHE, a data structure that encodes the necessary information from an execution of the automaton, and ensures strong eventual consistency. We begin by defining the specification automaton used for monitoring in Sec. 4.1, then we present the EHE data structure, illustrate how it can be used for monitoring in Sec. 4.2, and describe its use when partial observations are available in Sec. 4.3.

### 4.1 Preliminaries

Specifications are similar to the Moore automata generated by \cite{3}. We modify labels to be boolean expressions over atomic propositions (in $Expr_{AP}$).

**Definition 4.1 (Specification).** The specification is a deterministic Moore automaton $(Q, q_0 \in Q, \delta, \text{ver})$ where $\delta : Q \times Expr_{AP} \rightarrow Q$ is the transition function and $\text{ver} : Q \rightarrow B_3$ is the labeling function.

---

\(^3\)We assume that (1) no two components can observe the same atomic propositions, and (2) a component has at least one observation at all times (a component with no observations to monitor, can be simply considered excluded from the system under monitoring).

---

We choose to label the transitions with boolean expressions as opposed to events, to keep a homogeneous representation\(^4\). The labeling function associates a verdict with each state. The specification is a complete automaton, only one label will be equivalent to $\top$, and for a given state $q \in Q$ if we disjunct the labels of all outgoing transitions, the expression should be equivalent to $\top$. When using multiple automata we use the subscript notation to separate them, $A_i = (Q_i, q_{i0}, \delta_i, \text{ver}_i$. The semantics of the specification are given by a function $\Delta^5$.

**Definition 4.2 (Semantics of the specification).** Given $A_i$, a state $q \in Q_i$, and an event $e$, we build the memory $M = \text{mem}(e, i, dt)$. We have:

$$\Delta_i(q, e) = \begin{cases} \{ q' | \exists q' \in Q_i : go(q, q', M) \text{ if } e \neq \emptyset \} & \text{otherwise} \\ \emptyset \quad \text{if } \forall q \in Q_i : \text{eval}(e, M, q) \downarrow \end{cases}$$

Using $M$ we evaluate each label of an outgoing transition, and determine if a transition can be taken on $e$. In the case of an empty event ($e = \emptyset$), we return the same state. To handle a trace, we extend $\Delta_i$ to its reflexive and transitive closure in the usual way, and note it $\Delta_i^*$.

**Example 4.3 (Monitoring using expressions).** We consider $Atoms = AP = \{a, b\}$ and the specification in Fig. 1, we seek to monitor $F(a \lor b)$. The automaton consists of two states: $q_0$ and $q_1$ associated respectively with the verdicts $\bot$ and $\top$. We consider at $t = 1$ the event $e = ((a, \top), (b, \bot))$. The resulting memory is $M = [a \mapsto \top, b \mapsto \bot]$ (see Ex. 3.5). The transition from $q_0$ to $q_1$ is taken since $\text{eval}(a \lor b, M) = \top$. Thus we have $\Delta(q_0, e) = q_1$ with verdict $\text{ver}(q_1) = \top$.

### 4.2 Execution History Encoding

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 $q_0 : q_1 : \ldots : q_t : \ldots$. It indicates that for each timestamp $t \in [0, \infty[$ the automaton is in the state $q_t$. 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 main motivation behind designing the EHE encoding is 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.

\(^4\)Indeed, an event can be converted to an expression by the conjunction of all observations, negating the terms that are associated with the verdict $\bot$.

\(^5\)We note that in this case, we are not using any encoding ($Atoms = AP$).
Definition 4.4 (Execution History Encoding - EHE). An Execution History Encoding (EHE) of the execution of an automaton $\mathcal{A}_1$ is a partial function $I_1 : \mathbb{N} \times Q \rightarrow \text{Expr}$.

For a given execution, we encode the conditions to be in a state at a given timestamp as an expression in $\text{Expr}$. $I_1(t, q) = e$ indicates that the automaton is in state $q$ at $t$ if $\text{eval}(e, M) = \top$ given a memory $M$. Since we are encoding deterministic automata, we assume that when a state $q$ is reachable at $t$, no other state is reachable at $t$ (i.e., $\exists q' \in Q_1 : \text{eval}(I_1(t, q')) = \top \implies \forall q' : q' \neq q \implies \text{eval}(I_1(t, q')) \neq \top$).

To compute $I_1$ for a timestamp range, we will next define some (partial) functions: $\text{sel}_t$, $\text{verAt}_t$, $\text{next}_t$, $\text{to}_t$, and $\text{mov}_t$. The purpose of these functions is to extract information from $I_1$ at a given timestamp, which we can use to recursively build $I_1$ for future timestamps. Given a memory $M$ which stores atoms, the function $\text{sel}_t$ 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 undefined. The state $q$ at timestamp $t$ with a memory $M$ is determined by:

$$\text{sel}_t(I_1, M, t) = \begin{cases} q & \text{if } \exists q' \in Q_1 : \text{eval}(I_1(t, q'), M) = \top \\ \text{undef} & \text{otherwise} \end{cases}$$

Function $\text{verAt}_t$ is a short-hand to retrieve the verdict at $t$:

$$\text{verAt}_t(I_1, M, t) = \left\{ \begin{array}{ll} \text{ver}_t(q) & \text{if } \exists q' \in Q_1 : q = \text{sel}_t(I_1, M, t) \\ ? & \text{otherwise} \end{array} \right.$$  

The automaton is in the first state at $t = 0$. We start building up $I_1$ with the initial state and associating it with expression $\top$: $I_1 = [0 \rightarrow q_0 \rightarrow \top]$. Then, we check the next possible states in the automaton; for timestamp $t$, we look at the states in $I_1(t)$ and check for the possible states at $t + 1$ using $\delta_1$.

$$\text{next}_t(I_1, t) = \{ q' \mid \exists q \in I_1(t, q), \exists e : \delta_1(q, e) = q' \}$$

We now build the necessary expression to reach $q'$ from multiple states $q$ by disjoining the transition labels. Since the label consists of expressions in $\text{Expr}_\mathcal{A}$ we use an encoder to get an expression in $\text{Expr}_\text{Atoms}$. To get to the state $q'$ at $t + 1$ from $q$ we conjunct the condition to reach $q$ at $t$.

$$\text{to}_t(I_1, t, q', f) = \bigvee \{ (I_1(t, q) \land \text{enc}(e)) \mid (q, e') \in \delta(q, e') \}$$

By considering the disjunction, we cover all possible paths to reach a given state. Updating the conditions for the same state on the same timestamp is done by disjoining the conditions.

$$\text{mov}_t(I_1, t_s, t_e) = \left\{ \begin{array}{ll} \text{mov}_t(I_1', t_s + 1, t_e) & \text{if } t_s < t_e \\ I_1' & \text{otherwise} \end{array} \right.$$  

with: $I_1' = I_1 \uplus \bigcup \{ t_s + 1 \rightarrow q' \rightarrow \text{to}_t(I_1, t_s, q', t_{s+1}) \mid q' \in \text{next}_t(I_1, t_s) \}$.

Finally, $I_1'$ is obtained by considering the next states and merging all their expressions to $I_1$. We omit the $\mathcal{A}$ subscript when we have one automaton, and denote the encoding up to a timestamp $t$ as $I^t$.

**Example 4.5 (Monitoring with EHE).** We encode the execution of the automaton presented in Ex. 4.3. We have $I^0 = [0 \rightarrow q_0 \rightarrow \top]$. From $q_0$, it is possible to go to $q_0$ or $q_1$, therefore $\text{next}(I^0, 0) = \{ q_0, q_1 \}$. To move to $q_1$ at $t = 1$, we must be at $q_0$ at $t = 0$. The following condition must hold: $\text{to}(I^0, 0, q_1, t_{s_1}) = I^0(q_0) \land (t = 1)$, which is obtained with $I^2 = \text{mov}(I^0, 0, 2)$ and is shown in Table 1. We consider the same event as in Ex. 3.5 at $t = 1$, $e = \{(a, \top), (b, \bot)\}$. Let $M = \text{mem}(e, t_{s_1}) = \{(1, a) \mapsto \top, (1, b) \mapsto \bot\}$. It is possible to infer the state of the automaton after computing only $I^1 = \text{mov}(I^0, 0, 1)$ by using $\text{sel}(I^1, M, 1)$, we evaluate:

$$\text{eval}(I^1(q_0, q_1), M) = (\neg(a, 1) \land (1, b)) \Leftrightarrow (1, a) \lor (1, b) = \top$$

We find that $q_1$ is the selected state, with verdict $\text{ver}(q_1) = \top$.

**Proposition 4.6 (Soundness).** Given a decentralized trace $\tau$ of length $n$, we reconstruct the global trace $\bar{\tau} = \rho(\tau) = e_0 \cdot \ldots \cdot e_n$, we have:

$$\Delta^*(q_0, \bar{\tau}) = \text{sel}(I^n, M^n, n)$$  

with $I^n = \text{mov}([0 \rightarrow q_0 \rightarrow \top], 0, n)$  

$M^n = \{ e_i^2 \mid e_i \in [1, n], \text{mem}(e_i, t_{s_1}) \}$

The proposition asserts that EHE is sound wrt the specification automaton; given the same trace, both will indicate the same state reached. Thus, the verdict is the same as it would be in the automaton. The proof is by induction on the length of the reconstructed global trace and is in Appendix A.

### 4.3 Reconciling Execution History

We also note that EHE provides interesting properties for decentralized monitoring. Merging two EHEs of the same automaton with $\gamma_\mathcal{A}$ allows us to aggregate information from two partial histories. For the same execution of the automaton and a timestamp $t$, if we have two encodings $I(t, q) = e$ and $I'(t, q) = e'$, then we know that the automaton is in $q$ at $t$ iff either $e$ or $e'$ evaluates to $\top$ (Def. 4.4). Therefore, the new expression $e'' = e \lor e'$ can be an effective way to reconcile information from two encodings. The memory $M$ can be embedded in an expression $e$ by simply using $\text{rw}(e, M)$ (Def. 3.7). Thus, by rewriting expressions and combining EHEs, it is possible to reconcile multiple partial observations of an execution.

**Example 4.7 (Reconciling information).** We consider the specification $F(a \land b)$ (Fig. 2), and two components: $c_0$ and $c_1$ monitored by $m_0$ and $m_1$ respectively. The monitors can observe the propositions $a$ and $b$ respectively and use one EHE each: $I_0$ and $I_1$ respectively. Their memories are respectively $M_0 = [(1, a) \mapsto \top]$ and $M_1 = [(1, b) \mapsto \bot]$. Table 2 shows the EHEs at $t = 1$. Constructing the EHE follows similarly from Ex. 4.5. We show the rewriting for both $I_0$ and $I_1$ respectively in the next two columns. Then, we show the result of combining the rewrites using $\gamma_\mathcal{A}$. We notice initially that since $b$...
is $\bot$, $m_1$ could evaluate $\neg(1, a) \lor \top = \top$ and know that the automaton is in state $q_0$. However, for $m_0$, this is not possible until the expressions are combined. By evaluating the combination $(\bot \lor \neg(1, b)) \lor \top = \top$, $m_0$ determines that the automaton is in state $q_0$. In this case, we are only looking for expressions that evaluate to $\top$. We notice that the monitor $m_1$ can determine that $q_1$ is not reachable (since $(1, a) \land \bot = \bot$) while $m_0$ cannot ($(1, b)$). This does not affect the outcome, as we are only looking for one expression that evaluates to $\top$, since both $I_0$ and $I_1$ are encoding the same execution.

**Corollary 4.8.** An EHE with operation $\uparrow_\lor$ is a CvRDT.

## 5 DECENTRALIZED SPECIFICATIONS

In this section, we shift the focus to a specification that is decentralized. A set of automata represent various requirements for different components of a system. In Sec. 5.1, we define the notion of a decentralized specification and its semantics, and in Sec. 5.2, we tackle the monitorability of such specification.

### 5.1 Decentralizing the Specification

To decentralize the specification, we consider a set of monitor labels $\text{Mons} = \{m_0, \ldots, m_{n-1}\}$. Each monitor label $m_k$ is associated with a specification automaton $\mathcal{A}_k$ (Definition 4.1) and a component $C_k \subseteq C$ (with $k \in [0, n-1]$).

**Definition 5.1 (Monitor dependency).** The set of monitor ids in an expression $e$ is denoted by $\text{dep}(e)$.

$$\text{dep}(e) = \text{match } e \text{ with}
\begin{align*}
| \text{id} \in \text{Mons} &\rightarrow \{\text{id}\} | \text{e}_1 \land e_2 &\rightarrow \text{dep}(e_1) \cup \text{dep}(e_2) \\
| \neg e &\rightarrow \text{dep}(e) | e_1 \lor e_2 &\rightarrow \text{dep}(e_1) \cup \text{dep}(e_2)
\end{align*}$$

The dep function finds all monitors which the expression $e$ references by syntactically traversing it.

**Example 5.2 (Decentralized specification).** Figure 3 illustrates one decentralized specification, that monitors the same specification as Ex. 4.3. It consists of 2 monitors $m_0$ and $m_1$, with automata $\mathcal{A}_0$ and $\mathcal{A}_1$ respectively. We consider two atomic propositions $a_0$ and $b_0$ which can be observed by component $c_0$ and $c_1$ respectively. Monitor $m_0$ (resp. $m_1$) is attached to component $c_0$ (resp. $c_1$). We notice that $\mathcal{A}_0$ depends on the verdict from $m_1$ and only observations local to $c_0$, while $\mathcal{A}_1$ is only labeled with observations local to $c_1$. Given $e = m_1 \land a_0$, we have $\text{dep}(e) = \{m_1\}$.

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

**Definition 5.3 (Semantics of the decentralized specification automaton).** Consider the monitor id $m_k$, with the specification automaton $\mathcal{A}_k$, a state $q \in Q_k$, a decentralized trace of length $t$ with $i \in [0, t]$.

$$\Delta_k^\ast(q, tr, i, t) = \begin{cases} \Delta_k^\ast(q, tr, i), i + 1, t) & \text{if } i < t \\ \Delta_k^\ast(q, tr, i) & \text{otherwise} \end{cases}$$

$$M = \text{memc(tr(i, c_k), idt)} = \begin{cases} 2 & \text{if } tr(i, c_k) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

$$q_{ij} = \Delta_k^\ast(q_{j0}, tr, i, t)$$

For a monitor $m_k$, we determine the new state of the automaton starting at $q \in Q_k$, and running the trace $tr$ from timestamp $i$ to $t$ by applying $\Delta_k^\ast(q, tr, i, t)$. To do so, we evaluate one transition at a time using $\Delta_k^\ast$ as would $\Delta_\top^\ast$ with $\mathcal{A}_k$ (see Def. 4.2). To evaluate $\Delta_k^\ast$ at any state $q' \in Q_k$, 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 used $\text{memc(tr(i, c_k), idt)}$ which is based on the event with observations local to $c_k$. However, for monitor ids, the memory represents the verdicts of the monitors. To evaluate an id $m_j$, the remainder of the trace starting from the current event timestamp $i$ is evaluated recursively on the automaton $\mathcal{A}_j$ from the initial state $q_{j0} \in \mathcal{A}_j$. Then, the verdict of the monitor is associated with $m_j$ in the memory.

**Example 5.4 (Monitoring the decentralized specification).** Consider monitors $m_0$ and $m_1$ associated to components $c_0$ and $c_1$ respectively and the trace $tr = [1 \mapsto c_0 \mapsto \{(a, \bot), 1 \mapsto c_1 \mapsto \{(b, \bot), 2 \mapsto c_0 \mapsto \{(a, \bot), 2 \mapsto c_1 \mapsto \{(b, \top)}]]$. To evaluate $tr$ on $\mathcal{A}_0$ (from Fig. 3), we use $\Delta_0^\ast(q_{j0}, tr, 1, 2)$. To do so, we first evaluate $\Delta_k^\ast(q_{j0}, tr, 1)$. We notice that the expressions depend on $m_1$, therefore we need to evaluate $\Delta_1^\ast(q_{j1}, tr, 1, 2)$. All expressions have no monitor labels, thus we construct $M_1^\ast = \text{memc}((b, \bot), \text{idt}) = \{b \mapsto \bot\}$, and notice that $\text{eval}(-b, M_1^\ast) = \top$ and therefore it can move to state $q_{j2}$.
associated with verdict \(\perp\). Notice that \(\Delta'_1(A'_1(q_{ts}, tr, 1), tr, 2) = q_{ts}\) with \(ver_1(q_{ts}) = \perp\). We can construct \(M'_2 = mmecc((a, \perp), idt)\), \(t_2\) \(\{m_1 \mapsto \perp\} = [a \mapsto \perp, m_1 \mapsto \perp]\). We then have \(\text{eval}(\neg m_1 \land \neg a_0, Mem'_2) = \top\) and \(A'_9\) is in state \((m_0, q_0)\). By doing the same for \(t = 2\), we obtain \(Mem'_2 = [a \mapsto \perp, m_1 \mapsto \top]\), then we evaluate \(\text{eval}(m_1 \land \neg a_0) = \top\). This indicates that \(\Delta'_0(A'_0(q_{th}, tr, 1), tr, 2) = q_{th}\) and the final verdict is \(\top\).

**Remark 1 (Compliance/Violation).** Importantly, we do not define a global verdict for the system. We do not impose that a main monitor be present, or that the monitors be organized in a tree or list topology. We leave the choice of the topology and dependencies to the algorithm in question.

### 5.2 Decentralized Monitorability

We next consider the monitorability of decentralized specifications by introducing the notion of path expression.

**Path expression.** A path from a state \(q_s\) to a state \(q_e\) is expressed as an expression over atoms. We define paths \((q_s, q_e)\) to return all possible paths from \(q_s\) to \(q_e\).

**Definition 5.5 (Path expressions).**

\[
\text{paths}(q_s, q_e) = \left\{ \exists t \in \mathbb{N}, \exists \text{expr} : I^t(t, q_e) = \text{expr} \land I^t = \text{mov}(0 \mapsto q_s \mapsto \top, 0, t) \right\}
\]

The expression is derived similarly as would an execution in the EHE (Def. 4.4). Instead of executing from the initial state \(q_0\), we start from state \(q_s\) and use a logical timestamp starting at 0 incrementing it by 1 for the next reachable state.

**Decentralized monitorability.** Decentralized monitorability for a given automaton \(A_k\) is determined by looking at the paths that reach a state associated with a final verdict. However, since paths depend on other monitors, then it must also extend recursively to those monitors.

**Definition 5.6 (Decentralized monitorability).** A spec \(A_k\) is monitorable, noted monitorable\((A_k)\), iff \(\forall q \in Q, \exists q_f \in Q, \exists e_f \in \text{paths}(q, q_f)\), such that (1) \(e_f\) is satisfiable; (2) \(ver_1(\text{simplify}(q_f))) \in E_2\); (3) \(\forall m \in \text{dep}(e_f)\): monitorable\((A_l)\).

The first condition ensures that the path can be taken (i.e., there exists a trace that satisfies it)

6This is ensured if the automaton is generated from LTL [3].

### 6 ANALYSIS

We compare decentralized monitoring algorithms in terms of computation, communication and memory overhead. We first consider the parameters and the cost for the basic functions of the EHE. Then, we adapt the existing algorithms to use EHE and analyze their behavior. We use \(s_k\) to denote the size necessary to encode \(e\). For example, \(s_{AP}\) is the size needed to encode \(AP\).

#### 6.1 Data Structure Costs

**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 each \(f(x) = y\) mapping. We denote \(|\text{dom}(f)|\) the number of entries in \(f\). The size of each entry \(f(x) = y\) is the sum of the sizes \(|x| + |y|\). Therefore \(|f| = \sum_{x \in \text{dom}(f)} |x| + |f(x)|\).

**Merging.** Merging two memories or two EHEs, is linear in the size of both structures in both time and space. In fact to construct \(f' = f \cup g\), we first iterate over each \(x \in \text{dom}(f)\), check if \(x \in \text{dom}(g)\), and if so assign \(f'(x) = \text{op}(f(x), g(x))\), otherwise assign \(f'(x) = f(x)\). Finally we assign \(f'(y) = g(y) \forall y \in \text{dom}(g) : y \notin \text{dom}(f)\). This results in \(|\text{dom}(f')| = |\text{dom}(f) \cup \text{dom}(g)|\) which is at most \(|\text{dom}(f)| + |\text{dom}(g)|\).

**Information delay.** EHE associates an expression with a state for any given timestamp. When an expression \(\text{expr}\) associated with a state \(q_{kn}\) for some timestamp \(t_{kn}\) is \(\top\), we know that the automaton is in \(q_{kn}\) at \(t_{kn}\). We call \(q_{kn}\) a ‘known’ (or stable) state. Since we know the automaton is in \(q_{kn}\), prior information is no longer necessary, therefore it is possible to discard all \(t < t_{kn}\) in \(I\). We parametrize the number of timestamps needed to reach a new known state from an existing known state as the information delay \(\delta_i\). This can be seen as a garbage collection strategy [25, 28] for the memory and EHE.

**EHE encoding.** For the EHE data structure we are interested in three functions: move, eval, and \(s_1\). Function move depends on the topology of the automaton, we quantify it using the maximum size of the expression that labels a transition \(L\), the maximum size of outbound transitions from a state that share the same destination state \(P\), and the number of states in the automaton \(|Q|\). From a known state each move 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 \(L\) for each \(P\) for each move beyond \(t_{kn}\). For each timestamp, we need for each state an expression, the maximum size of the EHE is therefore:

\[
|I^{\delta_i}| = |Q^{\delta_i}| \sum_{t} LP = \delta_i |Q| LP.
\]

For a given expression \(\text{expr}\), we use \(|\text{expr}|\) to denote the size of \(\text{expr}\), i.e., the number of atoms in \(\text{expr}\). Given a memory \(M\), the complexity of function \(\text{eval}(\text{expr}, M)\) is the cost of \(\text{simplify}(\text{rw}(\text{expr}, M))\). Function \(\text{rw}(\text{expr}, M)\) looks up each atom in \(\text{expr}\) in \(M\) and attempts to replace it by its truth-value. The cost of a memory lookup is \(O(1)\), and the replacement is
linear in the number of atoms in \text{expr}. It effectively takes one pass to syntactically replace all atoms by their values, therefore the cost of \text{rw} is \(O(|\text{expr}|)\). However, \text{simplify()} requires solving the Minimum Equivalent Expression problem which is \(\Sigma_2^P\) complete [6], it is exponential in the size of the expression, making it the most costly function. \(|\text{expr}|\) is bounded by \(\delta_t \text{LP}\). Function \text{sel()} requires evaluating every expression in the \text{EHE}. For each timestamp we need at most \(|Q|\) expressions, and the number of timestamps is bounded by \(\delta_t\).

Memory. The memory required to store \(\text{Mons}\) depends on the trace, namely the amount of observations per component. We note that once a state is known, observations can be removed, the number of timestamps is bounded by \(\delta_t\). The size of the memory is then: \(\sum_{t=1}^{t^*} |\text{tr}(c,t)| \times (s_{\text{N}} \times s_{\text{AP}} \times s_{\text{B}})\).

6.2 Analyzing Existing Algorithms

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 the way their connections, and attaches them to the components. We represent the connections between the various monitors using a directed graph \((\text{Mons}, E)\) where \(E = 2^{\text{Mons}} \times \text{Mons}\) defines the edges describing the sender-receiver relationship between monitors. For example, the network \((\{m_0, m_1\}, \{(m_1, m_0)\})\) 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 [7] adapted to use \text{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 \text{Atoms} to consist of a timestamp and the atomic proposition \((\text{Atoms} = N \times \text{AP})\). 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 [3].

\text{Approach.} A decentralized monitoring algorithm consists of one or more monitors that use the \text{EHE} and memory data structures to encode, store, and share information. By studying \(\delta_t\), we derive the size of the \text{EHE} and the memory a monitor would use. 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 (\(\delta_t|Q|\)), and we know the bounds on the size of the expression to simplify (\(\delta_t \text{LP}\)). 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) [27] approach.

\text{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 \(\delta_i\) is then constant, \(\delta_i = d\). The number of messages sent at each round is \(|C| - 1\), i.e., the number of forwarding monitors sending their observations. The size of the message is linear in the number of observations for the component, for a forwarding monitor \(k\), the size of the message is \(MS_k = |\text{tr}(t, c_k)| \times (s_{\text{N}} \times s_{\text{AP}} \times s_{\text{B}})\).

\text{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 \text{EHE} encoding guarantees that two monitors receiving the same information are in the same state. Therefore, monitoring with Migration amounts to rewriting the \text{EHE} and migrating it across components. Since all monitors can send the \text{EHE} to any other monitor, the monitor network is a strongly-connected graph. In Migr, the delay depends on the choice of a choose function, which determines which component to migrate to next upon evaluation. By using a simple 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 Migration by the number of active monitors at a timestamp \(m\). The presented choose in [7], chooses at most one other component to migrate to. Therefore, after the initial choice of \(m\), subsequent rounds can have at most \(|C|\) active monitors. The initial choice of active monitors is bounded by \(m \leq |C|\). Since at most \(|C| - 1\) other monitors can be running, there can be \((m - 1)\) merges. The size of the resulting \text{EHE} is \(m \times |\text{tr}^t_{\text{q}}| = m(|C| - 1)^2|Q|\text{LP}\). In the worst case, the upper bound on the size of \text{EHE} is \((|C| - 1)^2|Q|\text{LP}\). The number of messages is bounded by the number of active monitors \(m\). The size of each message is however the size of the \text{EHE}, since Migr requires the entire \text{EHE} to be sent.

\text{Choreography.} Choreography (Chor) presented in [4, 7] splits the initial LTL formula into subformulae and delegates each subformula to a component. Thus Chor can illustrate how it is possible to monitor decentralized specifications. Once the subformulae are determined by splitting the main formula, 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 the monitor identified by its id. Therefore, \text{Atoms} = (N \times \text{AP}) \cup \text{(Mons \times 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 leafs consisting of monitors without any dependencies, and dependencies building up throughout the tree to reach the main monitor that outputs the verdict. The information delay for a monitor is thus dependent on its depth in the network tree. A monitor that is not monitorable will never emit a verdict, therefore its depth is \(\infty\). A leaf monitor
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. 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.

8 EVALUATION AND DISCUSSION

We use THEMIS to compare multiple existing algorithms and validate the behavior of the EHE data structure. We use the adapted existing three algorithms: Orch, Migr, and Chor presented in Sec. 6. We additionally consider a round-robin variant of Migr, Migr and use that for analyzing the behavior of the migration family of algorithms as it has a predictable choose. We conduct a study to confirm the analysis in Sec. 6 and explore the situations that best suit the algorithms.

Experimental setup. We generate the specifications as random LTL formulas using randltl from Spot [11] then converting the LTL formulae to automata using 1tm [3]. We generate traces by using the GeneratorTrace tool in THEMIS which generates synthetic traces by creating random events using a normal probability distribution. For all algorithms we considered the communication delay to be 1 timestamp. That is, messages sent at \( t \) are available to be received on \( t + 1 \). In the case of both Migration variants, we set the active monitors to 1 (\( m = 1 \)). For our experiment 200 traces of 100 events per component, we associate with each component 2 observations. We vary the number of components between 3 and 5, and ensure that for each number we have at least 1,000 formulae that references all components. We were not able to effectively use a larger number of components since the formula becomes sufficiently large that generating an automaton from it becomes unfeasible. The generated formulae were fully constructed of atomic propositions, there were no terms containing \( \perp \) or \( \top \). When computing sizes, we use a normal unit to separate the encoding from actual implementation strategies, our assumptions on the sizes is similar to the number of 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 runs taken to reach the final verdict (if applicable) or timeout, as different algorithms take different number of rounds to reach a verdict.

Monitoring metrics. The first considered metric is that of information delay \( \delta_i \). \( \delta_i \) 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. By considering our analysis in Sec. 6, we split our metrics into two main categories: communication and computation. We consider communication using two metrics: number of messages and size 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\textsubscript{ever}, or only the size needed to encode an id in the case of msg\textsubscript{kill}.

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 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.

Table 3: Scalability of Existing Algorithms.

| Algorithm     | \( \delta_i \) | # Mag | |Msg| |
|---------------|----------------|-------|-------|
| Orchestration | \( O(1) \)     | \( O(|C|) \) | \( O(AP_c) \) |
| Migration     | \( O(|C|) \) | \( O(m) \) | \( O(m|C|^2) \) |
| Choreography  | \( O(depth(0)) \) | \( O(|E|) \) | \( O(1) \) |

has no dependencies, its depth is 1. 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 [7]), 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\textsubscript{ever}, or only the size needed to encode an id in the case of msg\textsubscript{kill}.
data communicated. The number of messages is the total messages sent by all monitors throughout the entire run. The data communicated consists of the total size of messages sent by all monitors throughout the entire run. Both number of messages and the data transferred are normalized using the run length. The EHE structure requires the evaluation and simplification of a boolean expression which is costly (see Sec. 4.2). Thus, we count the number of simplier calls as a measure of computation required by the monitor. Alternatively, it can be seen as the number of formula evaluations conducted by the monitor. We measure simplifications per round as a normalized value, by taking the total number of simplifications performed by all monitors throughout the run and dividing by the run length. The simplifications per round describes the total computations done by all monitors. We introduce this metric to measure concurrency. Since monitors can execute in parallel, we are interested in the monitor with the most amount of simplifications to perform in a given round, since it acts as a bottleneck for concurrency. In addition, to capture load balancing, we introduce convergence where:

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

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.

**Results.** We present the results in Table 4, we show the algorithm followed by the number of components, and an average of our metrics across all runs. The column \#S denotes the normalized number of simplifications while \#S/Mon denotes the normalized number of simplifications per monitor. We notice that Orch maintains the lowest delay, followed by Migrr and Chor. In addition, we notice that Migrr has an average \( \delta_t \), increases with \(|C|\). We note that Migrr data transfer is an estimation of the size of the EHE, as we have one active monitor sending the EHE per round. This gives us a good estimate of the growth the size of EHE with \(|C|\) for Migrr. The size of EHE determines the computation required, we see that for higher \( \delta_t \), \#S increases. Since Chor is the only algorithm running multiple monitors we can see that it effectively spreads the computation across components, making its slowest monitor \((\#S/Mon)\) perform a bit worse than Orch, but with reasonable scalability wrt \(|C|\). On the one hand, Migrr sends a small and balanced \#Msgs that does not depend on \(|C|\), in exchange, its data transfer is much bigger as it has to send the entire EHE. On the other hand, we observe that both Orch and Chor maintain a smaller number of data than Migrr but a higher \#Msgs. We notice that in terms of data, Chor outperforms Orch. This is consistent with the observation that the size of messages for Chor is constant while, in the case of Orch, it scales with the number of observations per component.

| Alg. | \(|C|\) | \( \delta_t \) | \#Msgs | Data | \#S | \#S/Mon | Conv |
|------|--------|--------------|-------|------|-----|---------|------|
| Chor | 3      | 2.37         | 2.02  | 18.05| 15.27| 6.63    | 0.18 |
|      | 4      | 2.49         | 2.54  | 22.62| 18.22| 6.79    | 0.20 |
|      | 5      | 2.37         | 3.08  | 27.18| 21.29| 6.95    | 0.22 |
| Migrr| 3      | 1.02         | 0.36  | 49.46| 4.80 | 4.80    | 1.00 |
|      | 4      | 1.38         | 0.41  | 128.26| 5.67| 5.67    | 1.00 |
|      | 5      | 2.28         | 0.57  | 648.86| 9.40| 9.40    | 1.00 |
| Migrr| 3      | 1.09         | 0.86  | 58.02| 5.00 | 5.00    | 1.00 |
|      | 4      | 1.49         | 0.85  | 144.62| 5.91| 5.91    | 1.00 |
|      | 5      | 2.32         | 0.83  | 684.81| 9.60| 9.60    | 1.00 |
| Orch | 3      | 0.63         | 1.68  | 21.01| 4.13 | 4.13    | 1.00 |
|      | 4      | 0.65         | 2.43  | 30.42| 4.11 | 4.11    | 1.00 |
|      | 5      | 0.81         | 3.04  | 38.51| 5.55 | 5.55    | 1.00 |

**9 CONCLUSIONS AND FUTURE WORK**

We present a general approach to decentralized monitoring of 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. In addition, we present the EHE data structure which allows us to (i) aggregate monitor states with strong eventual consistency, (ii) remain sound wrt the execution of the monitor, and (iii) characterize the behavior of the algorithm at runtime. We then show how we can map the three existing algorithms: Orchestration, Migration and Choreography to our approach and modify them to use our data structures. We develop and use THEMIS to implement algorithms, design new metrics, and analyze the behavior of these algorithms.

We can now explore new directions of decentralized monitoring. One is to study algorithms that generate from a centralized specification, an equivalent decentralized one. Another direction is the design of new algorithms for combining monitors for specific parts of the systems. One can now separate the problem of the topology and dependencies of the monitors from the monitoring procedure. 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 Sec. 6). Moreover, one could consider creating new metrics for THEMIS to analyze more aspects of decentralized monitoring algorithms. New metrics would be automatically instrumented on all existing algorithms and experiments could be easily replicated to compare them. Finally, we shall consider new settings for runtime enforcement [12] frameworks: (i) decentralized runtime enforcement of centralized specifications and (ii) (decentralized) runtime enforcement of decentralized specifications.
REFERENCES


A PROOF

Proof of Prop. 4.6. Given a trace tr of length i and a reconstructed global trace \( \rho(tr) = q_0 \cdot \ldots \cdot q_i \), the proof is done by induction on the length of the trace \( |\rho(tr)| \).

Base case: \( |\rho(tr)| = 0, \rho(tr) = \emptyset \)

\[ \Delta(q_0, \emptyset) = \Delta(q_0, \emptyset) = q_0 = \text{sel}(\rho(0), [0]) \]

\[ \rho(0) = \text{mov}([0 \rightarrow q_0 \rightarrow \top], 0, \emptyset) = [0 \rightarrow q_0 \rightarrow \top] \]
We only have the expression \( \top \) which is mapped to \( q_0 \) at \( t = 0 \), \( \top \) requires no memory to evaluate.

Inductive step: We assume \( \Delta^i(q_0, e_0 \cdot \ldots \cdot e_i) = \text{sel}(I^i, M^i, i) = q_i \), we show it holds for length \( i + 1 \)

We first examine the content of the memories at \( i + 1 \) we have for the EHE: \( M^{i+1} = M^i \oplus \text{memc}(e_{i+1}, ts_{i+1}) \) we have:

**Lemma A.1.** \( \text{eval(idt}(e), M^{i+1}_{\mathcal{A}}) \Leftrightarrow \text{eval}(ts_{i+1}(e), M^{i+1}_{\mathcal{A}}) \).

In the case of the automaton, the memory is \( M^{i+1}_{\mathcal{A}} = \text{memc}(e_{i+1}, \text{idt}) \). We note that the encoding of the expression is different, still, the memories contain the same event modulo encoding at \( i + 1 \). That is, by construction, both the following hold:

\[
\forall a \in \text{dom}(M^{i+1}_{\mathcal{A}}) : (i + 1, a) \in \text{dom}(M^{i+1}) \\
\land M^{i+1}_{\mathcal{A}}(a) = M^{i+1}(i + 1, a) \\
\forall (i + 1, a') \in \text{dom}(M^{i+1}) : a' \in \text{dom}(M^{i+1}) \\
\land M^{i+1}_{\mathcal{A}}((i + 1, a')) = M^{i+1}(a')
\]

Therefore, for a given expression \( e \) evaluating \( e \) with both encodings with the appropriate memory for the encoding yield the same verdict.

We now consider the transition functions in the automaton:

\[
q_{i+1} = \Delta^i(q_0, e_0 \cdot \ldots \cdot e_{i+1}) \\
= (\Delta(\Delta^i(q_0, e_0 \cdot \ldots \cdot e_i), e_{i+1}) \text{ (Def. 4.2)}) \\
= \Delta(q_i, e_{i+1}) \text{ (Induction Hypothesis)} \\
\Leftarrow \exists \text{expr} : \delta(q_i, \text{expr}) = q_{i+1} \land \text{eval(expr, } M^{i+1}_{\mathcal{A}}) = \top \tag{1}
\]

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

We now consider the EHE operations to reach \( q_{i+1} \) from \( q_i \):

\[
q_i = \text{sel}(I^i, M^i, i) \\
\Rightarrow e = I^i(i, q_i) \text{ with eval}(e, M^i) = \top \tag{2} \\
\land \forall q'_i \in Q : q'_i \neq q_i \Rightarrow \text{eval}(I^i(i, q'_i)) \neq \top \\
\Rightarrow \text{to}(I^i, i, q_{i+1}, ts_{i+1}) = \top \tag{3}
\]

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

\[
\text{to}(I^i, i, q_{i+1}, ts_{i+1}) \\
= \bigvee \{ (q, e') \mid \delta(q, e', q'_{i+1}) \}
\]

\[
= \bigvee \{ (q, e') \mid \delta(q, e', q'_{i+1}, \land q \neq q_i) \\
\lor (ts_{i+1}(e'')) \}
\]

We split the disjunction to consider the expressions that only come from the 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:

\[
\exists \text{expr} : \delta(q_i, \text{expr}) = q_{i+1} \land \text{eval(expr, } M^{i+1}_{\mathcal{A}}) = \top \tag{1}
\]

\[
\Rightarrow \text{eval}(ts_{i+1}(\text{expr}), M^{i+1}) = \top \tag{4}
\]

\[
\Rightarrow \text{to}(I^i, i, q_{i+1}, ts_{i+1}) = \top \tag{5}
\]

(4) is obtained using Lemma A.1 and \( \text{idt(expr) = expr} \).

(5) follows from the disjunction.

Using the same approach, we can show that \( \forall q' \in \text{next}(I^i, i) : q' \neq q_{i+1} \Rightarrow \text{to}(I^i, i, q', ts_{i+1}) \neq \top \), 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, \( \text{to}(I^i, i, q_{i+1}, ts_{i+1}) = \top \iff \text{sel}(I^{i+1}, M^{i+1}, i + 1) = q_i \).

### B SIZE OF EHE

In Sec. 6.1, we mentioned that the size of EHE depends on the specification:

\[
|I^{\mathcal{D}_1}| = |\mathcal{Q}_{\mathcal{L}P}|
\]

To capture the size of an EHE, we look at the size of the message sent by a Migration algorithm with only 1 active monitor \( (m = 1) \). To minimize the variability we consider Migrr since it will migrate on each timestamp when the state is not resolved, sending the entire EHE. Fig. 4 shows the size of the message sent by Migrr in function of \( \mathcal{L}P \) in the experiment described in Sec. 8. Since the size is also affected by information delay, we group the information as to observe the variation wrt \( \mathcal{L}P \). We notice that indeed the size increases with the size of the labels and number of transition pairs.