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:
Antoine El-Hokayem and Yliès Falcone. 2017. Monitoring Decentralized Specifications . In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 12 pages.
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 subformula[1] 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

[1] We use the example from [7]: $\mathbf{GF}(a) \wedge \neg(\mathbf{GF}(a))$ is monitorable, but its subformulae are both non-monitorable.

ISSTA'17, Santa Barbara, CA, USA

DOI: 10.1145/3092703.3092723

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 $\perp$ (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, $\mathbf{G}(\top)$ could be rewritten $\top \wedge \mathbf{G}(\top)$ 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 `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 `dict` as our basic building block that maps a key to a value. `dict` supports two operations: `query` and `merge`. The `merge` operation is the only operation that modifies `dict`. The modifications never remove entries, the state of `dict` is then monotonically increasing. By ensuring that `merge` is idempotent, commutative, and associative we fulfill the necessary conditions for our data structure to be a CvRDT.

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

We model `dict` as a partial function f. The keys are the domain of f, i.e., $\mathrm{dom}(\mathrm{f})$ and values are mapped to each entry of the domain. The `query` operation checks if a key $k \in \mathrm{dom}(\mathrm{f})$ and returns $\mathrm{f}(k)$. If $k \notin \mathrm{dom}(\mathrm{f})$, then it is undef. 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 $op : (\mathrm{dom}(f) \times \mathrm{dom}(g)) \rightarrow$

$(\mathrm{codom}(f) \cup \mathrm{codom}(g))$ is a binary function.

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

On sets of functions, $\dagger_{op}$ applies pairwise: $\biguplus^{op}\{\mathrm{f}_1, \ldots \mathrm{f}_\mathrm{n}\} = ((\mathrm{f}_1 \dagger_{op} \mathrm{f}_2) \ldots \mathrm{f}_\mathrm{n})$ 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 with the addition of a total order ($\prec$) between the elements, so that it always chooses the highest element to guarantee idempotence, while $\dagger_\vee$ is simply the logical *or* operator. Respectively, we denote the pairwise set operators as $\biguplus^2$ and $\biguplus^\vee$.

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 from $\mathbb{B}_2 = \{\top, \bot\}$ 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. $Expr_{Atoms}$ (resp. $Expr_{AP}$) denotes the set of boolean expressions over *Atoms* (resp. AP). When omitted, *Expr* refers to $Expr_{Atoms}$. An encoder is a function $\mathrm{enc} : Expr_{AP} \to Expr_{Atoms}$ 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. A decentralized monitoring algorithm requires retaining, retrieving and communicating observations.

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). The event $\{\langle a, \top\rangle, \langle b, \bot\rangle\}$ 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 $\mathcal{M} : Atoms \to \mathbb{B}_3$ 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: $\mathrm{memc} : 2^{AP \times \mathbb{B}_2} \times (Expr_{AP} \to Expr_{Atoms}) \to Mem$.

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

$$
\begin{aligned}
\mathrm{memc}(e, \mathrm{idt}) &= [a \mapsto \top, b \mapsto \bot], \\
\mathrm{memc}(e, \mathrm{ts}_1) &= [\langle 1, a\rangle \mapsto \top, \langle 1, b\rangle \mapsto \bot].
\end{aligned}
$$

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 the operator $\dagger_2$. The total ordering is needed for the 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 memory is a `dict` and $\dagger_2$ is idempotent, associative, and commutative, it follows from Prop. 3.1 that it is a CvRDT.

Corollary 3.6. *A memory with operation $\dagger_2$ 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 $\mathcal{M}$ using $\mathrm{rw}(\texttt{expr}, \mathcal{M})$ defined as follows:

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

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. The operation rw yields a smaller formula to work with and repeatedly evaluate.

Example 3.8 (Rewriting). We consider $\mathcal{M} = [a \mapsto \top, b \mapsto \bot]$; and $e = (a \vee b) \wedge c$. We have $\mathcal{M}(a) = \top$, $\mathcal{M}(b) = \bot$, $\mathcal{M}(c) = ?$. Since c is associated with $? \notin \mathbb{B}_2$ then it will not be replaced. The resulting expression is $\mathrm{rw}(e, \mathcal{M}) = (\top \vee \bot) \wedge c$.

We eliminate additional atoms using boolean logic. We denote by $\mathrm{simplify}(\texttt{expr})$ the simplification of formula `expr` [2].

Example 3.9 (Simplification). Consider $\mathcal{M} = [a \mapsto \top]$ and $e = (a \wedge b) \vee (a \wedge \neg b)$. We have $e' = \mathrm{rw}(e, \mathcal{M}) = (b \vee \neg b)$. Atoms can be eliminated with $\mathrm{simplify}(e')$. 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 $\texttt{expr} \in Expr$ using a memory $\mathcal{M}$ yields a verdict. $\mathrm{eval} : Expr \times Mem \to \mathbb{B}_3$:

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

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 ?.

[2]This is also known as The Minimum Equivalent Expression problem [6].

Example 3.11 (Evaluating expressions). Consider $\mathcal{M} = [a \mapsto \top, b \mapsto \bot]$ and $e = (a \lor b) \land c$. We have $\text{simplify}(\text{rw}(e, \mathcal{M})) = \text{simplify}((\top \lor \bot) \land c) = c$, and $\text{eval}(e, \mathcal{M}) = ?$ which depends on c: we cannot emit a final verdict before observing c.

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 $\text{tr} : \mathbb{N} \times C \to 2^{AP \times \mathbb{B}_2}$.

Function tr assigns an event to a component for a given timestamp. We additionally define function $\text{lu} : AP \to C$ to associate an observation to a component[3].
$\text{lu}(ap) = c$ 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. 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 $\rho(\text{tr}) = e_1 \cdot \ldots \cdot e_n$ is s.t. $\forall i \in [1, n] : e_i = \bigcup_{c \in C} \text{tr}(i, c)$.

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

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 [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 $\langle Q, q_0 \in Q, \delta, \text{ver} \rangle$ where $\delta : Q \times Expr_{AP} \to Q$ is the transition function and $\text{ver} : Q \to \mathbb{B}_3$ 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).

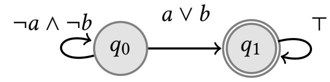

Figure 1: Representing $\text{F}(a \lor b)$

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, $\mathcal{A}_i = \langle Q_i, q_{i_0}, \delta_i, \text{ver}_i \rangle$. The semantics of the specification are given by a function Δ[5].

Definition 4.2 (Semantics of the specification). Given $\mathcal{A}_i$, a state $q \in Q_i$, and an event e, we build the memory $\mathcal{M} = \text{memc}(e, \text{idt})$. We have:

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

$$\text{go}(q, q', \mathcal{M}) \Leftrightarrow \exists e \in Expr_{AP} : \delta(q, e) = q' \land \text{eval}(e, \mathcal{M}) = \top$$

Using $\mathcal{M}$ 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 Δ_i to its reflexive and transitive closure in the usual way, and note it Δ_i^*.

Example 4.3 (Monitoring using expressions). We consider $Atoms = AP = \{a, b\}$ and the specifiction in Fig. 1, we seek to monitor $\text{F}(a \lor b)$. The automaton consists of two states: q_0 and q_1 associated respectively with the verdicts ? and $\top$. We consider at $t = 1$ the event $e = \{\langle a, \top \rangle, \langle b, \bot \rangle\}$. The resulting memory is $\mathcal{M} = [a \mapsto \top, b \mapsto \bot]$ (see Ex. 3.5). The transition from q_0 to q_1 is taken since $\text{eval}(a \lor b, \mathcal{M}) = \top$. 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 $q_0 \cdot q_1 \cdot \ldots q_t \cdot q_{t+1} \ldots$. 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 $\mathcal{A}_i$ is a partial function $\mathcal{I}_i : \mathbb{N} \times Q \to Expr$.

For a given execution, we encode the conditions to be in a state at a given timestamp as an expression in *Expr*. $\mathcal{I}_i(t, q) = e$ indicates that the automaton is in state q at t if $\text{eval}(e, \mathcal{M})$ is $\top$ given a memory $\mathcal{M}$. 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., $\exists q \in Q_i : \text{eval}(\mathcal{I}_i(t, q)) = \top \implies \forall q' \in Q_i : q' \neq q \implies \text{eval}(\mathcal{I}_i(t, q')) \neq \top$).

To compute $\mathcal{I}_i$ for a timestamp range, we will next define some (partial) functions: sel_i, verAt_i, next_i, to_i, and mov_i. The purpose of these functions is to extract information from $\mathcal{I}_i$ at a given timestamp, which we can use to recursively build $\mathcal{I}_i$ for future timestamps. Given a memory $\mathcal{M}$ which stores atoms, the function sel_i 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:

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

Function verAt_i is a short-hand to retrieve the verdict at t:

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

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

$$\text{next}_i(\mathcal{I}_i, t) = \{q' \mid \exists q \in \mathcal{I}_i(t, q), \exists e : \delta_i(q, e) = q'\}$$

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 $Expr_{AP}$ we use an encoder to get an expression in $Expr_{Atoms}$ To get to the state q' at $t + 1$ from q we conjunct the condition to reach q at t.

$$\text{to}_i(\mathcal{I}_i, t, q', f) = \bigvee_{\{\langle q, e' \rangle \mid \delta(q, e') = q'\}} (\mathcal{I}_i(t, q) \wedge \text{enc}(e'))$$

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.

$$\text{mov}_i(\mathcal{I}_i, t_s, t_e) = \begin{cases} \text{mov}_i(\mathcal{I}'_i, t_s + 1, t_e) & \text{if } t_s < t_e \\ \mathcal{I}_i & \text{otherwise} \end{cases}$$

with: $\mathcal{I}'_i = \mathcal{I}_i \dagger_\vee \biguplus^{\vee}_{q' \in \text{next}_{\mathcal{A}}(\mathcal{I}_i, t_s)} \{t_s + 1 \mapsto q' \mapsto \text{to}_i(\mathcal{I}_i, t_s, q', \text{ts}_{t_s+1})\}$.

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

Example 4.5 (Monitoring with EHE). We encode the execution of the automaton presented in Ex. 4.3. 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 move to q_1 at $t = 1$, we must be at q_0 at $t = 0$. The following condition must hold: $\text{to}(\mathcal{I}^0, 0, q_1, \text{ts}_1) = \mathcal{I}^0(0, q_0) \wedge (\langle 1, a \rangle \vee \langle 1, b \rangle) = \langle 1, a \rangle \vee \langle 1, b \rangle$. 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 consider the same event as in Ex. 3.5 at $t = 1$, $e = \{\langle a, \top \rangle, \langle b, \bot \rangle\}$. Let $\mathcal{M} = \text{memc}(e, \text{ts}_1) = [\langle 1, a \rangle \mapsto \top, \langle 1, b \rangle \mapsto \bot]$. It is possible to infer the state of the automaton after computing only $\mathcal{I}^1 = \text{mov}(\mathcal{I}^0, 0, 1)$ by using $\text{sel}(\mathcal{I}^1, \mathcal{M}, 1)$, we evaluate:

$$\begin{aligned} \text{eval}(\mathcal{I}^1(1, q_0), \mathcal{M}) &= \neg\langle 1, a \rangle \wedge \neg\langle 1, b \rangle &&= \bot \\ \text{eval}(\mathcal{I}^1(1, q_1), \mathcal{M}) &= \langle 1, a \rangle \vee \langle 1, b \rangle &&= \top \end{aligned}$$

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

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

t	q	expr
0	q_0	$\top$
1	q_0	$\neg\langle 1, a \rangle \wedge \neg\langle 1, b \rangle$
1	q_1	$\langle 1, a \rangle \vee \langle 1, b \rangle$
2	q_0	$(\neg\langle 1, a \rangle \wedge \neg\langle 1, b \rangle) \wedge (\neg\langle 2, a \rangle \wedge \neg\langle 2, b \rangle)$
2	q_1	$((\langle 1, a \rangle \vee \langle 1, b \rangle) \vee ((\neg\langle 1, a \rangle \wedge \neg\langle 1, b \rangle) \wedge (\langle 2, a \rangle \vee \langle 2, b \rangle)))$

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

$$\begin{aligned} & \Delta^*(q_0, \bar{e}) &&= \text{sel}(\mathcal{I}^n, \mathcal{M}^n, n) \\ \text{with} \quad & \mathcal{I}^n &&= \text{mov}([0 \mapsto q_0 \mapsto \top], 0, n) \\ & \mathcal{M}^n &&= \biguplus^2_{t \in [1,n]} \{\text{memc}(e_t, \text{ts}_t)\} \end{aligned}$$

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 $\dagger_\vee$ 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 $\mathcal{I}(t, q) = e$ and $\mathcal{I}'(t, q) = e'$, 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 $e'' = e \vee e'$ can be an effective way to reconcile information from two encodings. The memory $\mathcal{M}$ can be embedded in an expression e by simply using $\text{rw}(e, \mathcal{M})$ (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 $\mathbf{F}(a \wedge b)$ (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: $\mathcal{I}_0$ and $\mathcal{I}_1$ respectively. Their memories are respectively $\mathcal{M}_0 = [\langle 1, a \rangle \mapsto \top]$ and $\mathcal{M}_1 = [\langle 1, b \rangle \mapsto \bot]$. Table 2 shows the EHEs at $t = 1$. Constructing the EHE follows similarly from Ex. 4.5. We show the rewriting for both $\mathcal{I}_0$ and $\mathcal{I}_1$ respectively in the next two columns. Then, we show the result of combining the rewrites using $\dagger_\vee$. We notice initially that since b

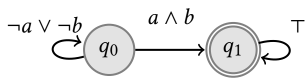

Figure 2: Representing $\mathrm{F}(a \wedge b)$

Table 2: Reconciling information

t	q	$\mathcal{I}_0$	$\mathcal{I}_1$	$\dagger_\vee$
0	q_0	$\top$	$\top$	$\top$
1	q_0	$\bot \vee \neg\langle 1, b\rangle$	$\neg\langle 1, a\rangle \vee \top$	$\top$
1	q_1	$\top \wedge \langle 1, b\rangle$	$\langle 1, a\rangle \wedge \bot$	$\langle 1, b\rangle$

is $\bot$, m_1 could evaluate $\neg\langle 1, a\rangle \vee \top = \top$ 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 $(\bot \vee \neg\langle 1, b\rangle) \vee \top = \top$, 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 $\langle 1, a\rangle \wedge \bot = \bot$) while m_0 cannot ($\langle 1, b\rangle$). This does not affect the outcome, as we are only looking for one expression that evaluates to $\top$, since both $\mathcal{I}_0$ and $\mathcal{I}_1$ are encoding the same execution.

Since `EHE` is a `dict` mapping a pair in $\mathbb{N} \times Q$ to an expression in *Expr*, and since the combination is done using $\dagger_\vee$, which is idempotent, commutative and associative, it follows from Prop. 3.1 that `EHE` is a CvRDT. Two components receiving the same `EHE` and merging them will be able to infer the same execution history of the automaton.

Corollary 4.8. *An* `EHE` *with operation* $\dagger_\vee$ *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 $\mathrm{Mons} = \{\mathrm{m}_0, \ldots, \mathrm{m}_{n-1}\}$. Each monitor label m_k is associated with a specification automaton $\mathcal{A}_k$ (Definition 4.1) and a component $c_k \in C$ (with $k \in [0, n-1]$). However, the transition labels of the automaton is restricted to either observations local to c_k or references to other monitors. Transitions are labeled over $Atoms_k = \mathrm{Mons} \setminus \{\mathrm{m}_k\} \cup \{ap \in AP \mid \mathrm{lu}(ap) = c_k\}$. This ensures that the automaton is labeled with observations it can locally observe or depends on other monitors.

Definition 5.1 (Monitor dependency). The set of monitor ids in an expression e is denoted by dep(e).

$\mathrm{dep}(e) = \texttt{match}\ e\ \texttt{with}$:

| $id \in \mathrm{Mons} \rightarrow \{id\}$ | $e_1 \wedge e_2 \rightarrow \mathrm{dep}(e_1) \cup \mathrm{dep}(e_2)$
| $\neg e \rightarrow \mathrm{dep}(e)$ | $e_1 \vee e_2 \rightarrow \mathrm{dep}(e_1) \cup \mathrm{dep}(e_2)$

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

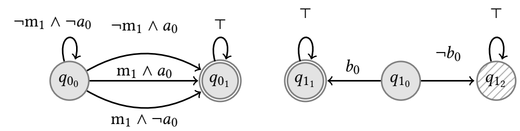

Figure 3: Representing $\mathrm{F}(a_0 \vee b_0)$ **(decentralized)**

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 $e = m_1 \wedge a_0$, we have $\mathrm{dep}(e) = \{m_1\}$.

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 tr of length t with $i \in [0, t]$.

$$\Delta_k'^*(q, \mathrm{tr}, i, t) = \begin{cases} \Delta_k'^*(\Delta_k'(q, \mathrm{tr}, i), i+1, t) & \text{if } i < t \\ \Delta_k'(q, \mathrm{tr}, i) & \text{otherwise} \end{cases}$$

$$\Delta_k'(q, \mathrm{tr}, i) = \begin{cases} q' \mid \begin{array}{l} \exists e : \delta_k(q, e) = q' \\ \wedge\, \mathrm{eval}(e, \mathcal{M}) = \top \end{array} & \text{if } \mathrm{tr}(i, c_k) \neq \emptyset \\ q & \text{otherwise} \end{cases}$$

$$\mathcal{M} = \mathrm{memc}(\mathrm{tr}(i, c_k), \mathrm{idt}) \dagger_2 \biguplus_{\mathrm{m}_j \in \mathrm{dep}(e)}^{2} \{[\mathrm{m}_j \mapsto \mathrm{ver}_j(q_{j_f})]\}$$

$$q_{j_f} = \Delta_j'^*(q_{j_0}, \mathrm{tr}, i, t)$$

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 timestamp t by applying $\Delta_k'^*(q, \mathrm{tr}, i, t)$. To do so, we evaluate one transition at a time using Δ_k' as would Δ_k^* with Δ_k (see Def. 4.2). To evaluate Δ_k' 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 using $\mathrm{memc}(\mathrm{tr}(i, c_k), \mathrm{idt})$ 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 $q_{j_0} \in \mathcal{A}_j$. 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 $\mathrm{tr} = [1 \mapsto c_0 \mapsto \{\langle a, \bot\rangle\}, 1 \mapsto c_1 \mapsto \{\langle b, \bot\rangle\}, 2 \mapsto c_0 \mapsto \{\langle a, \bot\rangle\}, 2 \mapsto c_1 \mapsto \{\langle b, \top\rangle\}]$. To evaluate tr on $\mathcal{A}_0$ (from Fig. 3), we use $\Delta_0'^*(q_{0_0}, \mathrm{tr}, 1, 2)$. To do so, we first evaluate $\Delta_0'(q_{0_0}, \mathrm{tr}, 1)$. We notice that the expressions depend on m_1, therefore we need to evaluate $\Delta_1'^*(q_{1_0}, \mathrm{tr}, 1, 2)$. All expressions have no monitor labels, thus we construct $\mathcal{M}_1^1 = \mathrm{memc}(\langle b, \bot\rangle, \mathrm{idt}) = [b \mapsto \bot]$, and notice that $\mathrm{eval}(\neg b, \mathcal{M}_1^1) = \top$ and therefore it can move to state q_{1_2}

associated with verdict $\bot$. Notice that $\Delta'_1(\Delta'_1(q_{1_0}, \text{tr}, 1), \text{tr}, 2) = q_{1_2}$ with $\text{ver}_1(q_{1_2}) = \bot$. We can construct $\mathcal{M}_0^1 = \text{memc}(\langle a, \bot \rangle, \text{idt}) \dagger_2 [\text{m}_1 \mapsto \bot] = [a \mapsto \bot, \text{m}_1 \mapsto \bot]$. We then have $\text{eval}(\neg \text{m}_1 \wedge \neg a_0, Mem_0^1) = \top$) and $\mathcal{A}_0$ is in state (m_0, q_0). By doing the same for $t = 2$, we obtain $Mem_0^2 = [a \mapsto \bot, \text{m}_1 \mapsto \top]$, we then evaluate $\text{eval}(\text{m}_1 \wedge \neg a_0) = \top$. This indicates that $\Delta'_0(\Delta'_0(q_{0_0}, \text{tr}, 1), \text{tr}, 2) = q_{0_1}$ 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 $\text{paths}(q_s, q_e)$ to return all possible paths from q_s to q_e.

Definition 5.5 (Path expressions).

$$\text{paths}(q_s, q_e) = \left\{ \texttt{expr} \;\middle|\; \begin{array}{l} \exists t \in \mathbb{N}, \exists \texttt{expr} : \mathcal{I}^t(t, q_e) = \texttt{expr} \\ \wedge\, \mathcal{I}^t = \text{mov}([0 \mapsto q_s \mapsto \top], 0, t) \end{array} \right\}$$

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 $\mathcal{A}_k$ 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 $\mathcal{A}_k$ is monitorable, noted $\text{monitorable}(\mathcal{A}_k)$, iff $\forall q \in Q_{\mathcal{A}_k} \exists q_f \in Q_{\mathcal{A}_k} \exists e_f \in \text{paths}(q, q_f)$, such that (1) e_f is satisfiable; (2) $\text{ver}_k(\text{simplify}(q_f)) \in \mathbb{B}_2$; (3) $\forall \text{m}_j \in \text{dep}(e_f)$: $\text{monitorable}(\mathcal{A}_j)$.

The first condition ensures that the path can be taken (i.e., there exists a trace that satisfies it)[6]. The second condition ensures that the state is labeled with a final verdict. And the third condition ensures that the atomic propositions that depend on other monitors can be evaluated. Additionally, q is said to be a *monitorable state*, and if $\text{ver}_k(q_f) = \bot$ (resp. $\top$), then we say that q can be negatively (resp. positively) determined. An automaton is monitorable iff we can from any state reach a state that can output a final verdict.

Remark 2 (Monitoring Completeness). *Verdicts must be eventually reported, so that they can be included in the memory to evaluate the expressions with monitor ids. A memory with a missing verdict generates the verdict ?, this could cause an expression to never evaluate (see Def. 3.10). Monitorability guarantees that a path expression to a final verdict can be evaluated. This however can require a timestamp t much higher than the length of the current trace and it ensures at least one monitor will terminate. Therefore, Δ_k^* may not terminate.*

[6] This 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_e 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 $f' = f \dagger_{\text{op}} g$, 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 timestmap. When an expression 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 $t < t_{\text{kn}}$ in $\mathcal{I}$. We parametrize the number of timestamps needed to reach a new known state from an existing known state as the information delay δ_t. 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: mov, eval, and sel[7]. Function mov 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 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 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:

$$|\mathcal{I}^{\delta_t}| = \delta_t |Q| \sum_{1}^{\delta_t} LP = \delta_t^2 |Q| LP.$$

For a given expression expr, we use $|\texttt{expr}|$ to denote the size of expr, i.e., the number of atoms in expr. Given a memory $\mathcal{M}$, the complexity of function $\text{eval}(\texttt{expr}, \mathcal{M})$ is the cost of $\text{simplify}(\text{rw}(\texttt{expr}, \mathcal{M}))$. Function $\text{rw}(\texttt{expr}, \mathcal{M})$ looks up each atom in expr 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

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

linear in the number of atoms in expr. It effectively takes one pass to syntactically replace all atoms by their values, therefore the cost of rw is $\theta(|\text{expr}|)$. However, simplify() requires solving the Minimum Equivalent Expression problem which is Σ_2^p-complete [6], it is exponential in the size of the expression, making it the most costly function. $|\text{expr}|$ is bounded by $\delta_t LP$. 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.

Memory. The memory required to store $\mathcal{M}$ 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 δ_t. The size of the memory is then: $\sum_{t=i}^{i+\delta_t} |\text{tr}(c,t)| \times (\text{s}_{\mathbb{N}} \times \text{s}_{AP} \times \text{s}_{\mathbb{B}_2})$.

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 $\langle \text{Mons}, E\rangle$ where $E = 2^{\text{Mons}} \times \text{Mons}$ defines the edges describing the sender-receiver relationship between monitors. For example, the network $\langle\{\text{m}_0, \text{m}_1\}, \{\langle \text{m}_1, \text{m}_0\rangle\}\rangle$ 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 [7] 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 ($Atoms = \mathbb{N} \times AP$). 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 [3].

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. 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 ($\delta_t LP$). 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.

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, $\delta_t = \text{d}$. 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 $MS_k = |\text{tr}(t, c_k)| \times (\text{s}_{\mathbb{N}} \times \text{s}_{AP} \times \text{s}_{\mathbb{B}_2})$.

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 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 m active monitors. 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. The size of the resulting EHE is $m \times |\mathcal{I}_t^{\delta}| = m(|C| - 1)^2|Q|LP$. In the worst case, the upper bound on the size of EHE is $(|C| - 1)^3|Q|LP$. The number of messages is bounded by the number of active monitors m. The size of each message is however the size of the EHE, since Migr requires the entire EHE to be sent.

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 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. 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 ∞. A leaf monitor

Table 3: Scalability of Existing Algorithms.

Algorithm	δ_t	# Msg	|Msg|
Orchestration	$\Theta(1)$	$\Theta(\|C\|)$	$O(AP_c)$
Migration	$O(\|C\|)$	$O(m)$	$O(m\|C\|^2)$
Choreography	$O(\text{depth}(0))$	$\Theta(\|E\|)$	$\Theta(1)$

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_{ver}, or only the size needed to encode an id in the case of msg_{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.

7 THE THEMIS FRAMEWORK

THEMIS is a tool to facilitate the design, development, and analysis of decentralized monitoring algorithms; developed using Java and AspectJ [16] (~5700 LOC). It consists of a library and command-line tools. The library provides all necessary building blocks to develop, simulate, and instrument 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 overhead of designing and assessing 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. 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, Migrr 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 `ltl2mon` [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 $\top$ or $\bot$.[8] When computing sizes, we use a normalized 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 rounds 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 δ_t. δ_t 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

[8] To generate formulae with basic operators, string `false=0,true=0,xor=0,M=0,W=0,equiv=0,implies=0,ap=6,X=2,R=0` is passed to `randltl`.

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 simplifier 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:

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

At a round t, we consider all simplifications performed on all components $\bar{s}^t$ 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 ($\frac{s_c^t}{\bar{s}^t}$), 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 δ_t that 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 Migr. The size of EHE determines the computation required, we see that for higher δ_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

Table 4: Decentralized Monitoring Metrics

Alg.	$\lvert C\rvert$	δ_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
Migr	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	646.86	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

Chor is constant while, in the case of Orch, it scales with the number of observations per component.

Discussion. The observed behavior of the simulation confirms the initial analysis described in Sec. 6. 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.

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

[1] 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. DOI: http://dx.doi.org/10.4204/EPTCS.124.9

[2] 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. DOI: http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2015.590

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

[4] 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. DOI: http://dx.doi.org/10.1007/978-3-642-32759-9_10

[5] Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. 2016. Challenges in Fault-Tolerant Distributed Runtime Verification, See [20], 363–370. DOI: http://dx.doi.org/10.1007/978-3-319-47169-3_27

[6] 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. DOI: http://dx.doi.org/10.1007/978-3-540-70575-8_3

[7] 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. DOI: http://dx.doi.org/10.1007/s10703-016-0251-x

[8] 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. DOI: http://dx.doi.org/10.1109/HPCC.2012.220

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

[10] 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. DOI: http://dx.doi.org/10.1007/978-3-642-32621-9_5

[11] 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. DOI: http://dx.doi.org/10.1007/978-3-319-02444-8_31

[12] 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. DOI: http://dx.doi.org/10.1007/978-3-642-16612-9_9

[13] 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. DOI: http://dx.doi.org/10.1007/978-3-662-43613-4_5

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

[15] 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. DOI: http://dx.doi.org/10.3233/978-1-61499-207-3-141

[16] 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. DOI: http://dx.doi.org/10.1007/3-540-45337-7_18

[17] 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. DOI: http://dx.doi.org/10.1109/EMRTS.1999.777457

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

[19] Martin Leucker, Malte Schmitz, and Danilo à Tellinghusen. 2016. Runtime Verification for Interconnected Medical Devices, See [20], 380–387. DOI: http://dx.doi.org/10.1007/978-3-319-47169-3_29

[20] 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. DOI: http://dx.doi.org/10.1007/978-3-319-47169-3

[21] 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. DOI: http://dx.doi.org/10.1007/11813040_38

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

[23] 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. DOI: http://dx.doi.org/10.1109/MEMCOD.2014.6961843

[24] 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. DOI: http://dx.doi.org/10.1109/ICSE.2004.1317464

[25] 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. DOI: http://dx.doi.org/10.1007/978-3-642-24550-3_29

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

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

[28] 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. DOI: http://dx.doi.org/10.1145/12485.12491

A PROOF

Proof of Prop. 4.6. Given a trace tr of length i and a reconstructed global trace $\rho(\mathrm{tr}) = e_0 \cdot \ldots \cdot e_i$, the proof is done by induction on the length of the trace $|\rho(\mathrm{tr})|$.

Base case: $\rho(\mathrm{tr})| = 0, \rho(\mathrm{tr}) = \emptyset$

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

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

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 $\Delta^*(q_0, e_0 \cdot \ldots \cdot e_i) = \mathrm{sel}(\mathcal{I}^i, \mathcal{M}^i, i) = q_i$, we show it holds for length $i + 1$

We first examine the content of the memories at $i + 1$ we have for the EHE: $\mathcal{M}^{i+1} = \mathcal{M}^i \dagger_2 \mathrm{memc}(e_{i+1}, \mathrm{ts}_{i+1})$ we have:

Lemma A.1. $\mathrm{eval}(\mathrm{idt}(e), \mathcal{M}^{i+1}_{\mathcal{A}}) \Leftrightarrow \mathrm{eval}(\mathrm{ts}_{i+1}(e), \mathcal{M}^{i+1})$.

In the case of the automaton, the memory is $\mathcal{M}^{i+1}_{\mathcal{A}} = \mathrm{memc}(e_{i+1}, \mathrm{idt})$. 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:

$$\begin{aligned}
\forall a \in \mathrm{dom}(\mathcal{M}^{i+1}_{\mathcal{A}}) : \quad & \langle i+1, a\rangle \in \mathrm{dom}(\mathcal{M}^{i+1}) \\
& \wedge \mathcal{M}^{i+1}_{\mathcal{A}}(a) = \mathcal{M}^{i+1}(\langle i+1, a\rangle) \\
\forall \langle i+1, a'\rangle \in \mathrm{dom}(\mathcal{M}^{i+1}) : \quad & a' \in \mathrm{dom}(\mathcal{M}^{i+1}_{\mathcal{A}}) \\
& \wedge \mathcal{M}^{i+1}(\langle i+1, a'\rangle) = \mathcal{M}^{i+1}_{\mathcal{A}}(a')
\end{aligned}$$

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:

$$\begin{aligned}
q_{i+1} &= \Delta^*(q_0, e_0 \cdot \ldots \cdot e_{i+1}) \\
&= \Delta(\Delta^*(q_0, e_0 \cdot \ldots \cdot e_i), e_{i+1}) \text{ (Def. 4.2)} \\
&= \Delta(q_i, e_{i+1}) \text{ (Induction Hypothesis)} \\
&\Leftrightarrow \exists \mathtt{expr} : \delta(q_i, \mathit{expr}) = q_{i+1} \wedge \mathrm{eval}(\mathit{expr}, \mathcal{M}^{i+1}_{\mathcal{A}}) = \top \text{ (1)}
\end{aligned}$$

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

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

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

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

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

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:

$$\begin{aligned}
\exists \mathtt{expr} : \delta(q_i, \mathit{expr}) = q_{i+1} \wedge \mathrm{eval}(\mathit{expr}, \mathcal{M}^{i+1}_{\mathcal{A}}) = \top && (1) \\
\Leftrightarrow \mathrm{eval}(\mathrm{ts}_{i+1}(\mathtt{expr}), \mathcal{M}^{i+1}) = \top && (4) \\
\Leftrightarrow \mathrm{to}(\mathcal{I}^i, i, q_{i+1}, \mathrm{ts}_{i+1}) = \top && (5)
\end{aligned}$$

(4) is obtained using Lemma A.1 and $\mathrm{idt}(\mathit{expr}) = \mathit{expr}$.
(5) follows from the disjunction.
Using the same approach, we can show that $\forall q' \in \mathrm{next}(\mathcal{I}^i, i) : q' \neq q_{i+1} \implies \mathrm{to}(\mathcal{I}^i, i, q', \mathrm{ts}_{i+1}) \neq \top$, 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, $\mathrm{to}(\mathcal{I}^i, i, q_{i+1}, \mathrm{ts}_{i+1}) = \top$ iff $\mathrm{sel}(\mathcal{I}^{i+1}, \mathcal{M}^{i+1}, i+1) = q_i$. □

B SIZE OF EHE

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

$$|\mathcal{I}^{\delta_t}| = \delta_t^2 |Q| LP.$$

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 LP 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 LP. We notice that indeed the size increases with the size of the labels and number of transition pairs.

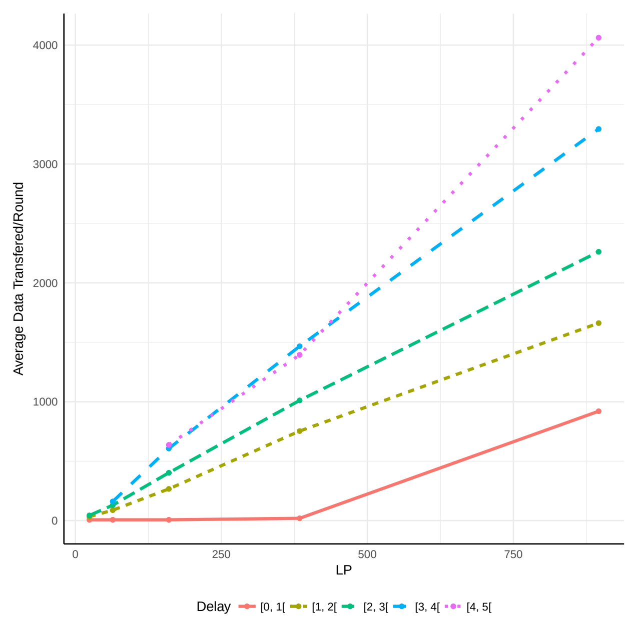

Figure 4: Size of EHE