Monitoring Decentralized Specifications
Antoine El-Hokayem, Yliès Falcone

To cite this version:

HAL Id: hal-01653725
https://hal.archives-ouvertes.fr/hal-01653725
Submitted on 1 Dec 2017

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

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

1 INTRODUCTION
Runtime Verification (RV) [2, 18, 21] 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 in instance in the context of decentralized automotive [9] and medical [22] 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 scaling existing approaches to decentralized systems, that is, systems with multiple components with no central observation point.

Challenges. Several algorithms have been designed [5, 6, 8, 16] 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 [5, 8, 25] 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 [17, 20, 24]) for centralized specifications exist [4, 10, 17], 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 [28]. 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 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

\(^1\)We use the example from [6]: \(GF(a) \land \neg(GF(a))\) is monitorable, but its subformulae are both non-monitorable.
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. An extended version of this paper (with full proofs) is available at [14].

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 [5, 8, 25]. 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 \( T \) (true) or \( F \) (false) at which point the algorithm terminates. Another approach [29] 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, \( G(\neg T) \) could be rewritten \( T \land G(T) \) and will keep growing in function of the number of timestamps. To tackle the unpredictability of rewriting LTL formulae, another approach [16] uses automata for monitoring regular languages, and therefore (i) can express richer specifications, and (ii) has predictable runtime behavior. This approach focuses on a centralized specification.

Another class of research focuses on handling a different problem that arises in distributed systems. In [6], 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 [3] 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 [3, 5, 8], and other works [11, 26, 27] 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) [28], 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., \( \text{dom}(f) \) and values are mapped to each entry of the domain. The query operation checks if a key \( k \in \text{dom}(f) \) and returns \( f(k) \). If \( k \notin \text{dom}(f) \), then it is undef. The merge operation of two dict \( f \) with another dict \( g \), is modeled as function composition. Two partial functions \( f \) and \( g \) are composed using operator \( \triangleright \) where \( \triangleright : (\text{dom}(f) \times \text{dom}(g)) \rightarrow (\text{dom}(f) \cup \text{dom}(g)) \) is a binary function.

\[
f \triangleright g(x) : \text{dom}(f) \cup \text{dom}(g) \rightarrow \text{dom}(f) \cup \text{dom}(g)
\]

\[
\begin{align*}
\triangleright & (f(x), g(x)) & \text{if } x \in \text{dom}(f) \cap \text{dom}(g) \\
& g(x) & \text{if } x \in \text{dom}(g) \setminus \text{dom}(f) \\
& f(x) & \text{if } x \in \text{dom}(f) \setminus \text{dom}(g) \\
& \text{undef} & \text{otherwise}
\end{align*}
\]

On sets of functions, \( \triangleright \) associates pairwise: \( \{g\}^{op} \{f_1, \ldots, f_n\} = \{(f_1 \triangleright \cdots \triangleright f_n)\} \). The following two operators are used in the rest of the paper: \( \triangleright \circ \) and \( \triangleright \triangleright \). We define both of these operators to be commutative, idempotent, and associative to ensure SEC.

\[
\triangleright \circ (x, x') = \begin{cases} 
  x' & \text{if } x < x' \\
  x & \text{otherwise}
\end{cases}
\]

\[
\triangleright \triangleright (x, x') = x \lor x'
\]

Operator \( \triangleright \circ \) 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 \( \triangleright \triangleright \) is simply the logical
or operator. Respectively, we denote the pairwise set operators as $\cup^2$ and $\vee^3$.

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 $enc : Expr_{AP} \rightarrow Expr_{Atoms}$ 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 $ts$, 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 $\{(a, \top), (b, \bot)\}$ 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 dictionary, and is modeled as a partial function $M : Atoms \rightarrow \mathbb{B}_3$ 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: $memc : 2^{AP \times \mathbb{B}_2} \times (Expr_{AP} \rightarrow Expr_{Atoms}) \rightarrow Mem$.

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

\[
\text{memc}(e, idt) = [\{a \mapsto \top, b \mapsto \bot\}], \\
\text{memc}(e, ts_1) = [\{(1, a) \mapsto \top, (1, b) \mapsto \bot\}].
\]

If we impose that $Atoms$ be a totally ordered set, then two memories $M_1$ and $M_2$ can be merged by applying operator $\oplus_2$. The total ordering is needed for the operator $\oplus_2$. 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 dictionary and $\oplus_2$ is idempotent, associative, and commutative, it follows from Prop. 3.1 that it is a CvRDT.

Corollary 3.6. A memory with operation $\oplus_2$ 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 $M$ using $rw(expr, M)$ defined as follows:

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

Using information from a memory $M$, the expression is rewritten by replacing atoms with a final verdict (a truth value in $\mathbb{B}_2$) 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 $? \notin \mathbb{B}_2$ then it will not be replaced. The resulting expression is $rw(e, M) = (\top \lor \bot) \land c$.

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 $e' = rw(e, M) = (b \lor \neg b)$. Atoms can be eliminated with $\text{simplify}(e')$. We finally get $\top$.

We combine both rewriting and simplification in the $\text{eval}$ 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.

\[
\begin{array}{l}
\text{eval} : Expr \times Mem \rightarrow \mathbb{B}_3:
\end{array}
\]

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

\[
\text{with } e' = \text{simplify}(rw(expr, M))
\]

Function $\text{eval}$ returns the $\top$ (resp. $\bot$) verdict if the simplification after rewriting is (boolean) equivalent to $\top$ (resp. $\bot$), otherwise it returns the verdict $?$.\footnote{This is also known as The Minimum Equivalent Expression problem [7].}

Example 3.11 (Evaluating expressions). Consider $M = [a \mapsto \top, b \mapsto \bot]$ and $e = (a \lor b) \land c$. Âž We have $\text{simplify}(rw(e, M)) = \text{simplify}((\top \lor \bot) \land c) = c$, and $\text{eval}(e, M) = ?$ 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 $\text{tr} : \mathbb{N} \times C \rightarrow 2^{AP \times \mathbb{B}_2}$.

Function $\text{tr}$ 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.\footnote{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).}

\[
l(u(p)) = e \text{ s.t. } \exists i \in N, \exists v \in B_2 : (ap, v) \in tr(t, c).
\]

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 LTL3. 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 \([4]\). We modify labels to be boolean expressions over atomic propositions (in \(\text{Expr}_{\text{AP}}\)).

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

We choose to label the transitions with boolean expressions as opposed to events, to keep a homogeneous representation.\footnote{Indeed, an event can be converted to an expression by the conjunction of all observations, negating the terms that are associated with the verdicts.}

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, \(A_i = (Q, q_0, \delta_i, \text{ver}_i)\). The semantics of the specification are given by a function \(\Delta_i\).

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

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

\[
\text{go}(q, q', M) \iff e \neq \emptyset \land (\exists q' : \text{go}(q, q', M) \land \text{eval}(e, M) = \top)
\]

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 \(\text{Atoms} = \text{AP} = \{a, b\}\) and the specification \(F(a \lor b)\). The automaton consists of two states: \(q_0\) and \(q_1\) associated respectively with the verdicts \(a\) 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 \(q_0 \cdot q_1 \cdot \ldots \cdot q_t \cdot q_{t+1} \ldots\). 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.

**Definition 4.4 (Execution History Encoding - EHE).** An Execution History Encoding (EHE) of the execution of an automaton \(A_i\) is a partial function \(I_i : N \times Q \rightarrow \text{Expr}\).

For a given execution, we encode the conditions to be in a state at a given timestamp as an expression in \(\text{Expr}\). \(I_i(t, q) = e\) indicates that the automaton is in a 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., \(3q' \in Q_i : \text{eval}(I_i(t, q)) = \top \implies \forall q' \in Q_i : q' \neq q \implies \text{eval}(I_i(t, q')) \neq \top\)

To compute \(I_i\) for a timestamp range, we will next define some (partial) functions: \(\text{sel}_i, \text{verAt}_i, \text{next}_i, \text{to}_i,\) and \(\text{mov}_i\). The purpose
of these functions is to extract information from \( I_t \) at a given
timestamp, which we can use to recursively build \( I_{t+1} \) for future
timestamps. Given a memory \( M \) which stores atoms, the function sel,
determines if a state is reached at a timestamp \( t \). If the memory
does not contain enough information to evaluate the expressions,
then the state is undefined. The state \( q \) at timestamp \( t \) with a memory
\( M \) is determined by:

\[
\text{sel}(I_t, M, t) = \begin{cases} 
q & \text{if } \exists q \in Q_1 : \text{eval}(I_t(t, q), M) = \top \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

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

\[
\text{verAt}(I_t, M, t) = \begin{cases} 
\text{ver}(q) & \text{if } \exists q \in Q_1 : q = \text{sel}(I_t, M, t) \\
? & \text{otherwise}
\end{cases}
\]

The automaton is in the first state at \( t = 0 \). We start building
up \( I_t \) with the initial state and associating it with expression \( \top \):
\( I_0 = [0 \mapsto q_0 \mapsto \top] \). Then, we check the next possible states in
the automaton; for timestamp \( t \), we look at the states in \( I_t(t) \) and
check for the possible states at \( t + 1 \) using \( \delta_i \).

\[
\text{next}(I_t, t) = \{ q' \mid \exists q \in I_t(t, q), \exists e : \delta_i(q, e) = q' \}
\]

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 \( \text{Expr}_\mathcal{M} \) 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 \( t \).

\[
\text{to}(I_t, t, q', f) = \bigvee (I_t(t, q) \land \text{enc}(e')) \land \delta(q, e) = q' \land (q, e')
\]

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.

\[
\text{mov}(I_t, t_s, t_e) = \bigvee_{I_t} \text{mov}(I_t', t_s + 1, t_e)
\]

otherwise

with: \( I_t' = I_t \uplus \bigvee_{q'\in\text{next}(I_t, t_s)} (t_s + 1 \mapsto q' \mapsto \text{to}(I_t, t_s, q', ts+1)) \).

Finally, \( I_t' \) is obtained by considering the next states and merging
all their expressions to \( I_t \). We omit the \( \mathcal{M} \) 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 \( I_0 = [0 \mapsto q_0 \mapsto \top] \).
From \( q_0 \), it is possible to go to \( q_1 \) or \( q_2 \), 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:
\[
\text{to}(I_0, 0, q_1, ts_1) = I_0^0(0, q_0) \land ((1, a) \lor (1, b))
\]

(\( 1, b \)) = (1, a) \lor (1, b). The encoding up to timestamp \( t = 2 \) 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 \( M = \text{mem}(e, ts_1) = \{(1, a) \mapsto \top, (1, b) \mapsto \bot\} \). It is possible
to infer the state of the automaton after computing only
\( I^1 = \text{mov}(I_0, 0, 1) \) by using sel\( (I^1, M, 1) \), we evaluate:

\[
\begin{align*}
\text{eval}(I^1(1, q_0), M) &= \neg(1, a) \land \neg(1, b) = \bot \\
\text{eval}(I^1(1, q_1), M) &= (1, a) \lor (1, b) = \top
\end{align*}
\]

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

**Proposition 4.6 (Soundness).** Given a decentralized trace \( t \) of
length \( n \), we reconstruct the global trace \( \tilde{t} = \rho(tr) = e_0 \cdots e_n \),
with:

\[
I^n = \text{mov}([0 \mapsto q_0 \mapsto \top], 0, n), \text{ and } M^n = \bigcup_{t \in [1,n]} \text{mem}(e_t, ts_t)
\]

EHE is sound wrt the specification automaton; both will indicate
the same state reached with a given trace. Thus, the verdict is the
same as it would be in the automaton. The proof is by induction
on the reconstructed global trace. First, we establish that both the
EHE and the automaton memories evaluate two similar expressions
modulo encoding with the same result. Then, starting from the
same state at length \( n \), we build the expression (for each encoding)
to reach the state at \( n + 1 \). Then we show that the expression (for
each encoding) is the only expression that evaluates to \( \top \).

**4.3 Reconciling Execution History**
We also note that EHE provides interesting properties for decentral-
ized monitoring. Merging two EHEs of the same automaton with \( \tau \)
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 \( I(t, q) = \epsilon \) and \( I'(t, q) = \epsilon' \), 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 specifi-
cation \( F(a \land b) \) (Fig. 2), and two components: \( c_0 \) and \( c_1 \) monitored
by \( m_0 \) and \( m_1 \) respectively. The monitors can observe the proposi-
tions \( a \) and \( b \) respectively and use one EHE each: \( I_0 \) and \( I_1 \) respec-
tively. 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 \( \tau \). We notice initially
that since \( b = \bot \), \( m_1 \) 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.
To decentralize the specification, we consider a set of monitor labels that can be observed by component \(c\). It consists of 2 monitors with the specification corresponding to the specification in Ex. 4.3. The automaton is restricted to either observations local to \(c\) or global. Monitor labels \(m\) and \(n\) are associated to components \(c_0\) and \(c_1\), respectively. Monitor \(m\) and \(n\) are idempotent, commutative and associative, it follows from Prop. 3.1 that a specification automaton \(M_{\text{spec}}\) and its semantics, and in Sec. 5.2, we tackle the memory represents the verdicts of the monitors. To evaluate an expression \(e\) in the memory is constructed using \(\text{mem}(e)\). This ensures that the automaton is labeled with observations or depends on other monitors.

\[ \Delta^*_k(q, \text{tr}(i), i) = \begin{cases} \Delta_k^*(\Delta_k^*(q, \text{tr}(i), i + 1, i), i < t) \\ \Delta_k^*(q, \text{tr}(i), i), \text{otherwise} \end{cases} \]

\[ \Delta_k^*(q, \text{tr}(i), i) = \begin{cases} q' & \text{if } \text{tr}(i, c_k) \neq \emptyset \\ q & \text{otherwise} \end{cases} \]

\[ M = \text{mem}(\text{tr}(i, c_k), \text{idt}), \text{idt} \rightarrow_{\text{tr}} \begin{cases} [\text{mem}(q_j) \rightarrow \text{ver}(q_j)] & \text{for } m_j \in \text{mem}(e) \end{cases} \]

For a monitor \(m_k\), we determine the new state of the automaton starting at a state \(q \in Q_k\), and running the trace \(\text{tr}(i)\) from timestamp \(t\) to \(\text{tr}(i)\) by applying \(\Delta_k^*(q, \text{tr}(i), i)\). To do so, we evaluate one transition at a time using \(\Delta_k^*(q, \text{tr}(i), i)\) and \(\Delta_k^*(q, \text{tr}(i), i)\) (see Def. 4.2). To evaluate \(\Delta_k^*(q, \text{tr}(i), i)\) 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 \(\text{mem}(\text{tr}(i, c_k), \text{idt})\) 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 \(t\) is evaluated recursively on the automaton \(A_i\) from the initial state \(q_{i0}\) in \(A_i\). Then, the verdict of the monitor is associated with \(m_j\) in the memory.

\[ \Delta_k^*(q, \text{tr}(i), i) = \begin{cases} \Delta_k^*(\Delta_k^*(q, \text{tr}(i), i + 1, i), i < t) \\ \Delta_k^*(q, \text{tr}(i), i), \text{otherwise} \end{cases} \]

\[ M = \text{mem}(\text{tr}(i, c_k), \text{idt}), \text{idt} \rightarrow_{\text{tr}} \begin{cases} [\text{mem}(q_j) \rightarrow \text{ver}(q_j)] & \text{for } m_j \in \text{mem}(e) \end{cases} \]

For a monitor \(m_k\), we determine the new state of the automaton starting at a state \(q \in Q_k\), and running the trace \(\text{tr}(i)\) from timestamp \(t\) to \(\text{tr}(i)\) by applying \(\Delta_k^*(q, \text{tr}(i), i)\). To do so, we evaluate one transition at a time using \(\Delta_k^*(q, \text{tr}(i), i)\) and \(\Delta_k^*(q, \text{tr}(i), i)\) (see Def. 4.2). To evaluate \(\Delta_k^*(q, \text{tr}(i), i)\) 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 \(\text{mem}(\text{tr}(i, c_k), \text{idt})\) 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 \(t\) is evaluated recursively on the automaton \(A_i\) from the initial state \(q_{i0}\) in \(A_i\). Then, the verdict of the monitor is associated with \(m_j\) in the memory.

\[ \Delta_k^*(q, \text{tr}(i), i) = \begin{cases} \Delta_k^*(\Delta_k^*(q, \text{tr}(i), i + 1, i), i < t) \\ \Delta_k^*(q, \text{tr}(i), i), \text{otherwise} \end{cases} \]

\[ M = \text{mem}(\text{tr}(i, c_k), \text{idt}), \text{idt} \rightarrow_{\text{tr}} \begin{cases} [\text{mem}(q_j) \rightarrow \text{ver}(q_j)] & \text{for } m_j \in \text{mem}(e) \end{cases} \]
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).

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

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 $\mathcal{A}_k$ 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 $\text{monitorable}(\mathcal{A}_k)$, if $\forall q \in Q. \exists q_f \in Q. \exists \text{expr} \in \text{paths}(q, q_f)$, such that (1) $e_f$ is satisfiable; (2) $\text{verAt}(\text{expr}(q_f)) \in \exists_2$; (3) $\forall m_j \in \text{dep}(e_f) : \text{monitorable}(A_j)$.

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 $\text{verAt}(q) = \bot$ (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 $\bot$, which 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 that at least one monitor will terminate. Therefore, $\lambda_k^*$ may not terminate.

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 $s_p$ to denote the size necessary to encode $e$. For example, $s_{AP}$ is the size needed to encode $AP$.

\(^6\)This is ensured if the automaton is generated from LTL [4].
we know the bound on the number of simplifications a monitor \( C \) −

| \( C \) |

then constant, \( d \) of a message takes at most \( \delta_t \) rounds. 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) \[30\] 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 \( \delta_t \) is then constant, \( \delta_t = d \). 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 \( MS_k = |tr(t, c_k)| \times (s_{\text{AP}} \times s_{\text{LP}}) \).

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 their connections, and attaches them to the components. We represent the connections between the various monitors using a directed graph (Mons, E) where \( E = 2^{\text{Mons}} \times \text{Mons} \) defines the edges describing the sender-receiver relationship between monitors. For example, the network \( \langle (m_0, m_1), (m_1, m_0) \rangle \) describes a network consisting of two monitors \( m_0 \) and \( m_1 \) where \( m_1 \) sends information to \( m_0 \). In this 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 \[8\] 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 \( \text{Atoms} = \mathbb{N} \times \text{AP} \). 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 \[4\].

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 \( \delta_t \) 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) \[30\] approach.

Memory. The memory required to store \( M \) 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 \( \delta_t \). The size of the memory is then:

\[
\sum_{t=i}^{i+\delta_t} |tr(c, t)| \times (s_{\text{AP}} \times s_{\text{LP}}).
\]

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 parameterize Migration by the number of active monitors at a timestamp \( m \). The presented choose in \[8\], 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 \( m \times (\delta_t|Q| = m(|C| − 1))^2Q[LP] \). In the worst case, the upper bound on the size of EHE is \( (|C| − 1)^2Q[LP] \). 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 \[5, 8\] 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, \( \text{Atoms} = (\mathbb{N} \times \text{AP}) \cup (\text{Mons} \times \mathbb{N}) \). Monitoring is done by replacing the subformula by the id of the monitor associated with it. Therefore, monitors are organized in a tree, the leaves 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 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 \[8\]), 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
THEMIS provides a base set of metrics (such as messages exchanged provides an API for monitoring and necessary data structures to load, and 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 adapted existing algorithms (Orch, Migr, and Chor - Sec. 6) and validate the behavior of the EHE data structure. 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 [12] then converting the LTL formulae to automata using 1tl2mon [4]. 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 T or ⊤. 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 $\delta_I$. $\delta_I$ 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 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

$^{9}$To generate formulae with basic operators, string false=0, true=0, xor=0, Eq=0, Eqv=0, implies=0, ap=0, l=2, f=0 is passed to randltl.

$^{8}$The THEMIS framework is further described and demonstrated in the companion tool-demonstration paper [13] and on its Website [14].
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:

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

At a round $t$, we consider all simplifications performed on all components $s_c^t$ 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 $s_c^t$, 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 Migr and Chor. In addition, we notice that Migr has an average $\delta_1$ that increases with $|C|$. We note that Migr 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 of the size of EHE with $|C|$ for Migr. The size of EHE determines the computation required, we see that for higher $\delta_1$, #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, Migr 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 Migr 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.

**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.

| Alg. | $|C|$ | $\delta_1$ | #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 |
| Migr | 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    | 0.65       | 2.43  | 30.42| 4.11 | 4.11  | 1.00 |
|      | 4    | 0.81       | 3.04  | 38.51| 4.80 | 4.80  | 1.00 |

9 CONCLUSIONS AND FUTURE WORK

We present a general approach to monitoring decentralized specifications. A specification is a set of automata associated with monitors that are attached to various components. We provide a general decentralized monitoring algorithm defining the major steps needed to monitor such specifications. 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 map three existing algorithms: Orchestration, Migration, and Choreography to our approach using our data structures. We develop and use THEMIS to implement algorithms and analyze their behavior by designing new metrics.

We 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 [15]: (i) decentralized runtime enforcement of centralized specifications and (ii) (decentralized) runtime enforcement of decentralized specifications.

REFERENCES


Borzo Bonadorguir, Pierez Fruignaund, Sergio Rajbauma, and Coerinten Traviers. 2016. Challenges in Fault-Tolerant Distributed Runtime Verification. See [23], 363–370. DOI: http://dx.doi.org/10.1007/978-3-319-47169-3_27


