**Noname manuscript No.**
(will be inserted by the editor)

# Choreography-Based Runtime Verification of Message Sequences in Distributed Message-Based Systems

**Mahboubeh Samadi · Fatemeh Ghassemi · Ramtin Khosravi**

the date of receipt and acceptance should be inserted later

**Abstract** In message-based systems such as the Internet of Things and Cyber-Physical Systems, some particular message orderings may lead to the unwanted happening, e.g., protocol violation or sensitive data disclosure. Due to the existence of third-party components, such message sequences as unwanted orderings can be neither statically inspected nor verified by revising their codes at design time. We propose a choreography-based runtime verification algorithm to detect unwanted sequences. Our algorithm is fully decentralized in the sense that each component is equipped with a monitor that has access to some parts of the message sequences. Due to the absence of a global clock and the network delay, the total ordering of messages cannot be recognized by the monitors. Monitors use vector clocks and declare the result of the sequence formation conservatively based upon the existence of concurrent messages. We prove the soundness and completeness of our algorithm. We evaluate the performance of our algorithm by developing a test case generator which produces message-based applications with different levels of complexity and a set of message sequences as unwanted orderings. We have also developed a simulator tool which simulates the generated applications and our algorithm and measures the performance metrics. Our experiment results show that our algorithm is scalable: with the increase of the complexity of applications or the length of message sequences, the number of monitoring messages grows linearly.

**Keywords** Asynchronous Message Passing · Choreography-Based Runtime Verification · Decentralized Monitoring Algorithm · Message Ordering

M. Samadi
University of Tehran, Iran
E-mail: mbh.samadi@ut.ac.ir

F. Ghassemi
University of Tehran, Iran
E-mail: fghassemi@ut.ac.ir

R. Khosravi
University of Tehran, Iran
E-mail: r.khosravi@ut.ac.ir

## 1 Introduction

With the rapid growth of the Internet of Things (IoT) and Cyber-Physical Systems (CPS), more physical entities are controlled by the collaboration of the computational elements. These systems are usually distributed in nature, comprising of various components which are possibly developed by different vendors. Hence, message-based communication can be a good choice for integrating these distributed components in contrast to tightly coupled synchronous communication methods such as remote procedure calls. This results in more maintainable and more scalable solutions.

The rise in using message-based systems brings into focus the need for formal methods to verify the correctness of these systems with respect to their required properties, e.g., data confidentiality, safety, and security. Especially, when the components in the system are closed-source, the system engineer does not have access to the source code or the design models to use well-known methods such as model checking. Hence, there is a need for verification methods which considers the components as black boxes.

One source of information that can be used to detect the violation of the properties in such systems is the messages exchanged among the components. For instance, in a smart home as an IoT application, one can infer by observing the message sequence $turnOff(cooler)$ (the cooler is turned off), $turnOn(hallwaylight)$ (the hallway light is turned on), $close(parkingdoor)$ (the parking door is closed) that the occupants have left home. By the occurrence of this sequence, the confidentiality of the occupant's existence at home is violated. As another example, assume a building with a set of entrances that is equipped with a set of smart security cameras, namely $A$, $B$, $C$ and $D$ to control the paths that people traverse [1]. The only legal path to the restricted area monitored by the camera $D$, is through the areas monitored by $A$ and then $C$. If an intruder is observed consecutively by the cameras $A$, $B$, and then $D$, it can be inferred that the intruder accesses the restricted area of $D$ illegally. This illegal access violates the security rules of the building and can be detected by the message sequence $observe(A,v)$ (the camera $A$ has observed the visitor $v$), $observe(B,v)$ and $observe(D,v)$. There are many examples of such sequence-based patterns in the *Complex Event Processing* domain [2–4]. Furthermore, the order of communicated messages in actor-based programs [5,6], as a sample of message-based programs, may lead to the *message protocol violation* bug [7,8]. This bug can be observed when the components exchange messages that are not consistent with the intended protocol of the application. *Linked predicate*s [9], properties (predicate) defined based on the sequence of events, are also an example of message sequences if events are restricted to messages.

As mentioned before, due to the existence of closed-source and proprietary components, the formation of unwanted message sequences can be neither inspected statically, nor verified by techniques like model checking. Rather, runtime verification, as a light-weight formal method based on monitoring executions of a system at runtime, can be used to check whether the behavior of the system meets certain properties [10]. One possible way to accomplish that is by using a *central monitor* that collects messages from components and then checks the property. This approach has severe communication and computation overhead due to many messages passing with the central monitor [11,12]. Another approach is that each component is equipped with a local monitor. These *decentralized monitors*

communicate with each other to evaluate the given property. Three decentralized settings *orchestration*, *migration*, and *choreography* are introduced for organizing the decentralized monitors [13]. In orchestration, a single monitor carries all the monitoring processes by receiving messages from local monitors (similar to the traditional central monitor). In migration, the monitor transports across the components of a system and evaluates the property as it goes along. In choreography, local monitors are organized into a network and collaborate with each other by using a specific protocol. The orchestration and migration settings deal with the *centralized specification* while choreography-based setting deals with the *decentralized specification* [14]. With the centralized specification, local monitors have the same global property. With the decentralized specification, each local monitor has its own local property which may depends to the local properties of the other monitors.

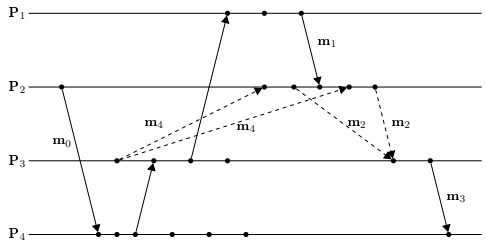



Fig. 1: The communicated messages among the four processes $P_1$, $P_2$, $P_3$, and $P_4$. The circles on the horizontal lines denote the execution moments of events. The message sending from the process $P_i$ to the process $P_j$ is depicted by an arrow

In this paper, we focus on the decentralized runtime verification of properties based on the messages sequences. These sequences specify particular orderings among sending and receiving messages of distributed components. Upon the occurrence of a message, it may either lead to the sequence formation or cancel the effect of the partially formed sequence and lead to the sequence corruption. Consider a system in which the sequence of messages $\mathfrak{m}_0\mathfrak{m}_1\mathfrak{m}_3$ should not occur. In Fig. 1, which illustrates an execution scenario of the system, the process $P_1$ has sent the message $\mathfrak{m}_1$ after the moment that $P_2$ has sent $\mathfrak{m}_0$. The process $P_3$ has sent $\mathfrak{m}_3$ after the sending moment of $\mathfrak{m}_1$. Assume that an occurrence of $\mathfrak{m}_2$ eliminates the effect of $\mathfrak{m}_1$. The process $P_2$ sends message $\mathfrak{m}_2$ as a consequence of receiving the message $\mathfrak{m}_4$. If the message $\mathfrak{m}_2$ has been sent after the sending of $\mathfrak{m}_1$ and before the sending of $\mathfrak{m}_3$, then it eliminates the effect of $\mathfrak{m}_1$ and so the sequence formation is stopped. Otherwise, the sequence $\mathfrak{m}_0\mathfrak{m}_1\mathfrak{m}_3$ has been formed. In general, we may have a set of message sequences that obey a specific pattern. We introduce a formal notation for specifying such message sequences based on nondeterministic finite automata (Sect. 3). The class of properties called *regular safety properties* (Sect. 2), can be recognized using this notation. Regular safety properties are used in many practical applications and can be recognized and processed based on finite automata. The decentralized runtime detection of messages sequences in a message-based system is challenging due to the absence of a global clock and the network delay. The delay of the network affects on the arrival time of messages

and messages are not received with the same order as they are sent. With the absence of a global clock, the order of messages can not be distinguished as components own their local clocks which are not synchronized [15]. So, the occurrence of messages among distributed components are *partially* ordered and the *total* order among them is not detectable. Our decentralized runtime verification is based on decentralized specifications. The decentralized specification keeps the communication among the components secret as a monitor only accesses to the part of the specification and so it is not aware of all communicated messages between components. Hence, we propose a decentralized message-based specification and a choreography-based algorithm for the collaboration among monitors to detect the sequence formations with regard to their local specifications (Sect. 4). We will use vector clocks [16], a vector of local clocks of components, in our messages to detect the partial ordering among messages. As monitors cannot detect the total order among messages, they may behave conservatively based upon the existence of concurrent messages and announce that the sequence may be formed or not formed so far. We aim to minimize the conservative results in our choreography-based algorithm when the message sequences have not been formed. We prove that our algorithm is *sound* and *complete* (Sect. 5). It is sound in the sense that if the total order of messages can be somehow constructed, then our algorithm declares the verification result for the total order as one of the possible results. It is also complete in the sense that among the results declared by the local monitors, at least one corresponds to the result for the total order of messages [17]. To the best of our knowledge, there is no decentralized runtime verification of decentralized specification for sequence-based properties in message-based systems. We evaluate the performance of our algorithm by developing a test case generator which produces message-based applications with different levels of complexity and a set of message sequences as unwanted orderings. We have also developed a simulator tool which simulates the generated applications and our algorithm and measures the performance metrics. Our experiment results show that our algorithm is scalable: with the increase of the complexity of applications or the length of message sequences, the number of monitoring messages grows linearly (Sect. 6).

## 2 Background

As we focus on properties that can be recognized based on the finite automaton, we first explain regular safety properties and the preliminary notation for specifying message-based properties. We borrow the terminology for regular safety properties from [18]. Then, we define the message-based systems and explain how the vector clocks are maintained in such systems.

### 2.1 Regular Safety Properties

Let $AP$ be a set of atomic propositions. A *safety property* is the set of infinite words over $2^{AP}$ such that every trace that violates the safety property has a bad prefix that causes the property to be false. Bad prefixes are finite, and the set of bad prefixes constitutes a language of finite words over the alphabet $2^{AP}$. Safety properties over $AP$ is called *regular* if its set of bad prefixes constitutes a regular

language over $2^{AP}$, and hence can be recognized by a finite automaton. We extend the notation of nondeterministic finite automaton in Section 3 to specify message-based sequences as the bad-prefixes of a regular safety property.

**Definition 1 (Nondeterministic Finite Automaton)** A nondeterministic finite automaton $\mathcal{M} = (Q, \Sigma, \delta, Q_0, F)$ is a 5-tuple where, $Q$ is the finite set of states, $\Sigma$ is the finite set of input symbols called the alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, $Q_0 \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of accepting states.

Let $\gamma = \gamma_1\gamma_2 \ldots \gamma_n$ be a string over the alphabet $\Sigma$. The automaton $\mathcal{M}$ accepts the string $\gamma$ if a sequence of states $q_0q_1 \ldots q_n$ exists in $Q$ such that $q_{i+1} \in \delta(q_i, \gamma_{i+1})$ for $i = 0 \ldots n-1$, and $q_n \in F$. The language recognized by $\mathcal{M}$, denoted by $L(\mathcal{M})$, is the set of strings accepted by $\mathcal{M}$.

### 2.2 Message-Based Systems

We define a *Message-Based System* $D = \{P_1, P_2, \ldots, P_n\}$ as a set of $n$ processes that communicate via asynchronous message-passing and guarantees in-order delivery, i.e., two messages that are sent directly from one process to another will be delivered and processed in the same order that they are sent. We assume that each process has a unique identifier and a message queue. A process sends messages to a target process by using its identifier. Each process takes messages from its queue one by one in FIFO order and invokes a handler regarding the name of the message.

Let *ID* be the set of possible identifiers, ranged over by $x$, $y$, and $z$. For simplicity, we assume $ID = \mathbb{N}$ throughout the paper. Let *MName* be the set of message names, *Var* be the set of variable names and *Msg* be the set of messages communicated among processes ranged over by $\mathtt{m}$. Each message $\mathtt{m} \in Msg$ has three parts: the sender identifier, the message name, and the receiver identifier, hence $Msg = ID \times MName \times ID$. Each process $P_x$ with the identifier $x$ is defined by a set of message handlers ($Hdlr_x$) and state variables ($Var_x$), denoted by $\langle x, Var_x, Hdlr_x \rangle$. For each message name $m \in MName$, a message handler $Hdlr_x(m)$ specifies how the received message must be responded to. For simplicity, we have ignored the parameters in message handlers. So, $Hdlr_x$ is considered as a function which maps a message name to a sequences of statements.

The computation of each process $P_x = \langle x, Var_x, Hdlr_x \rangle$ can be abstracted in terms of *event*s which are categorized into *internal*, *send*, and *handle* events:

- An *internal* event changes the state variables of $P_x$,
- The event $send(P_x, m, P_y)$ occurs when $P_x$ sends the message $m$ to $P_y$ which is denoted by $\texttt{send}(P_y, m)$ in the program text of $P_x$. This message is buffered in the queue of $P_y$.
- The event $handle(P_y, m, P_x)$ occurs when $P_x$ takes a message $m$ sent by $P_y$ from its queue.

### 2.3 Event Ordering in Message-Based Systems

The total ordering of events is determined by a global physical clock. As processes in a message-based system do not share a global clock and do not necessarily

execute at the same speed, the total order of events is not recognizable. However, events in a message-based system can be partially ordered by using happened-before relation [19]. The partial order does not specify the exact order among events, rather it only defines an ordering among events that are dependent on each other. The happened-before relation for a message-based system is defined as bellow.

**Definition 2 (Happened-Before Relation)**
Let $e_a$ denote the event generated as a consequence of executing a statement, called $a$, and a message $\mathtt{m}_i \in Msg$ be a triple of $(P_x, m_i, P_y)$. A happened-before relation $\rightsquigarrow$ defines a causal order among events. Within a single message handler, the ordering of events is defined as their execution order which can be determined unambiguously. For other cases, the happened-before relation is defined according to the following rules:

- if $a \in Hdlr_x(m_1)$, then $handle(P_y, m_1, P_x) \rightsquigarrow e_a$.
- if $\mathtt{send}(P_y, m_1) \in Hdlr_x(m_2)$, then $send(\mathtt{m}_1) \rightsquigarrow handle(P_y, m_1, P_x)$.
- if $\mathtt{send}(P_y, m_1)$, $\mathtt{send}(P_y, m_2) \in Hdlr_x(m)$, then $send(\mathtt{m}_1) \rightsquigarrow send(\mathtt{m}_2)$ implies $handle(P_y, m_1, P_x) \rightsquigarrow handle(P_y, m_2, P_x)$.
- For events $e_a, e_b, e_c$, if $e_a \rightsquigarrow e_b$ and $e_b \rightsquigarrow e_c$ then $e_a \rightsquigarrow e_c$.

The first condition defines the happened-before relation between a message handler and the statements of a process. The second condition defines that the *send* event in a process happens before the corresponding *handle* event in another process. The third condition explains that such systems guarantee in-order delivery. The transitivity of this relation is defined as the last condition. We employ the vector clock [16] to implement the happened-before relation. To this aim, each process $P_x = \langle x, Var_x, Hdlr_x \rangle$ has a reserved variable $vc_x \in Var_x$ denoting its vector clock. Each $vc_x$ is an array of integers where its length is equal to the number of processes in the system, where $vc_x[y]$ is the logical clock [19] of the process $P_y$ that $P_x$ is aware of. We assume that the vector clock of a process is updated atomically upon executing the events. Let $e$ be an event that occurred as the result of a statement executed by $P_x$. Then, $vc(e)$ denotes the value of $vc_x$ immediately after the occurrence of $e$. The value of all elements in $vc_x$ is initialized to zero. Upon executing a statement, $vc_x[x]$ is incremented by one. When $P_x$ sends a message to $P_y$, the vector clock of $P_x$ is piggybacked with the message. Upon taking a message from the message queue, $P_x$ extracts information about the logical clock of other processes. It increments its own logical clock (i.e. $vc_x[x]$) by one and updates other elements in its vector (i.e. $vc_x[y]$, $x \neq y$) by taking the maximum value of its own vector clock and the received one. There is a happened-before relation between two events $e_a$ and $e_b$ if and only if $vc(e_a) < vc(e_b)$, where $vc(e_a) < vc(e_b)$ is an abbreviation for $(\forall k \cdot vc(e_a)[k] \leq vc(e_b)[k]) \wedge (\exists j \cdot vc(e_a)[j] < vc(e_b)[j])$ [20]. If two events $e_a$ and $e_b$ are not related by $\rightsquigarrow$ relation, then they can be executed in any order:

**Definition 3 (Concurrent Events)**
Two events $e_a$ and $e_b$ are called *concurrent*, denoted by $e_a \parallel e_b$, if there is no happened-before relation between them, i.e., $e_a \not\rightsquigarrow e_b \wedge e_b \not\rightsquigarrow e_a$. Trivially two events $e_a$ and $e_b$ are concurrent if and only if $vc(e_a) \not< vc(e_b)$ and $vc(e_b) \not< vc(e_a)$.

## 3 Message-Based Property Specification

We consider regular safety properties that their bad-prefixes consist of the sequences of *send*/*handle* events. As a sequence of *send*/*handle* events may lead to the formation of a bad-prefix, the occurrence of one *send*/*handle* event may eliminate the effect of the part of the formed sequence and so cancel the formation of the bad-prefix. We introduce the *sequence automaton* to formally specify the set of bad message sequences.

Sequence automata extends nondeterministic finite automata by partitioning the transitions into two sets of *forward* and *backward* transitions. The self-loops transitions or the transitions on a simple directed path from an initial state to an accepting state are called forward and the other transitions are called backward. Forward transitions contribute to the formation of bad-prefixes while backward transitions cancel the formation of bad-prefixes. The alphabets of sequence automata are defined over *send* and *handle* events. To simplify the explanation of our algorithm and the representation of our property, we only focus on the sequence of *send* events. So, from now on, the alphabets of sequence automata are $\mathfrak{m} \in Msg$ where the label $\mathfrak{m}$ stands for $send(\mathfrak{m})$. In the general case, the message must be annotated with the event type *send* or *handle* to distinguish the two types of events. We use $q \rightarrow q'$ to denote that there is a forward transition between the state $q$ and the state $q'$ for some message $\mathfrak{m}$. Similarly, we use $q \dashrightarrow q'$ to denote that there is a backward transition from $q$ to $q'$ for some message $\mathfrak{m}$. Let $\rightarrow^*$ be the transitive closure of the $\rightarrow$ relation.

**Definition 4 (Sequence Automaton)**
Given a nondeterministic finite automaton $(Q, \Sigma, \delta, Q_0, F)$, the 6-tuple $(Q, \Sigma, \delta_f, \delta_b, Q_0, F)$ is a sequence automaton $(SA)$, where $\delta = \delta_f \cup \delta_b$, $\delta_f \cap \delta_b = \emptyset$, and all transitions specified by $\delta_f$ (resp. $\delta_b$) are forward (resp. backward) transitions, and:

- For all simple paths from any initial state $q_0 \in Q_0$ to any final state $q_n \in F$ passing through $q_1 \ldots q_{n-1}$, it holds that $\forall\, i < n, q_i \not\dashrightarrow q_{i+1}$.
- $q_i \dashrightarrow q_j \Rightarrow q_i \neq q_j \wedge q_j \rightarrow^* q_i$.

The first condition ensures that only forward transitions contribute to the simple paths from initial states to the final states. The second condition ensures that for each backward transition $(q_i, \mathfrak{m}, q_j)$, there exists a path which is made up of at least one forward transition, connecting $q_j$ to $q_i$. Furthermore, it restricts self-loops to forward transitions. The backward transition $(q_i, \mathfrak{m}, q_j)$ eliminates the effect of the occurrence of the messages which are labeled over the sequence of forward transitions on a path from $q_j$ to $q_i$. A sequence automaton $\mathcal{A} = (Q, \Sigma, \delta_f, \delta_b, Q_0, F)$ accepts the sequence $\mathfrak{m}_1\mathfrak{m}_2 \ldots \mathfrak{m}_n$ iff the underlying nondeterministic finite automaton $(Q, \Sigma, \delta_f \cup \delta_b, Q_0, F)$ accepts $\mathfrak{m}_1\mathfrak{m}_2 \ldots \mathfrak{m}_n$.

The sequence automaton $\mathcal{A}_1$, in Fig. 2, represents the sequences of *send* events as the bad-prefixes of a property. The forward transitions are shown with solid lines and the backwards with dashed lines. For instance, the automaton $\mathcal{A}_1$ describes that if first the message $\mathfrak{m}_2$ is sent and then the message $\mathfrak{m}_4$ is sent while the message $\mathfrak{m}_5$ is not sent after $\mathfrak{m}_2$ and before $\mathfrak{m}_4$, then the sequence $\mathfrak{m}_2\mathfrak{m}_4$ as a bad-prefix is formed. If the sequence $\mathfrak{m}_2\mathfrak{m}_5\mathfrak{m}_4$ is observed, the occurrence of $\mathfrak{m}_5$ after $\mathfrak{m}_2$ has eliminated the effect of the occurrence of $\mathfrak{m}_2$ and so, the occurrence of $\mathfrak{m}_4$ will not make a bad-prefix (as the reaching state $q_0$ is not a final state). However,

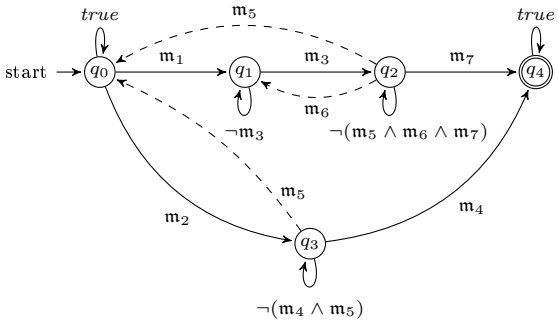



Fig. 2: The sequence automaton $\mathcal{A}_1$

a bad-prefix is formed by the sequence $\mathfrak{m}_2\mathfrak{m}_5\mathfrak{m}_2\mathfrak{m}_4$. The self-loop over the state $q_3$ expresses that between the occurrences of $\mathfrak{m}_2$ and $\mathfrak{m}_4$, any message except $\mathfrak{m}_4$ and $\mathfrak{m}_5$ can be sent. We use propositional logic formula $\phi$ over a self-loop transition $(q, \phi, q)$ to denote a set of transitions $(q, \mathfrak{m}, q)$ where $\phi ::= \mathit{true} \mid \mathfrak{m} \mid \neg\phi \mid \phi_1 \wedge \phi_2$. We define $\mathfrak{m} \models \phi$ as:

$$\begin{array}{l} \mathfrak{m} \models \mathit{true} \\ \mathfrak{m} \models \mathfrak{m} \\ \mathfrak{m} \models \neg\phi \quad\quad \text{, if } \mathfrak{m} \not\models \phi \\ \mathfrak{m} \models \phi_1 \wedge \phi_2 \text{ , if } \mathfrak{m} \models \phi_1 \wedge \mathfrak{m} \models \phi_2. \end{array}$$

Upon the occurrence of a message $\mathfrak{m}$, its corresponding transition $(q, \mathfrak{m}, q')$ leads to the formation of a bad-prefix if at least a message over one of its preceding transition $t$ has occurred and no message over the backward transitions has violated the effect of $t$. As our runtime verification algorithm reasons about the *preceding* and *violating* transitions, we define the two functions *preTrns* and *vioTrns*. Since the self-loop transitions do not change the states of the sequence automaton and so have no effect in our algorithm, we define these functions for non-self-loop transitions. The pre-transitions of a given transition $(q, \mathfrak{m}, q')$ is the set of transitions whose labeled messages can occur before $\mathfrak{m}$ in a sequence, and so they have the same destination as the source of $(q, \mathfrak{m}, q')$, i.e., $q$:

**Definition 5 (pre-transition)** For the given sequence automaton $\mathcal{A}$, the pre-transitions of the transition $(q, \mathfrak{m}, q') \in \delta_f$, where $q \neq q'$, is the set of forward transitions that end in state $q$. Also, the pre-transitions of the transition $(q, \mathfrak{m}, q') \in \delta_b$ is the set of forward transitions that end in state $q$ and are visited on a path from $q'$ to $q$:

$$\mathit{preTrns}(\mathcal{A}, (q, \mathfrak{m}, q')) = \begin{cases} \{(q'', \mathfrak{m}', q) \mid (q'', \mathfrak{m}', q) \in \delta_f \wedge q \neq q''\}, \\ \qquad\qquad \text{if } \ (q, \mathfrak{m}, q') \in \delta_f \\ \{(q'', \mathfrak{m}', q) \mid (q'', \mathfrak{m}', q) \in \delta_f \wedge q \neq q'' \wedge q' \rightarrow^* q''\}, \\ \qquad\qquad \text{if } \ (q, \mathfrak{m}, q') \in \delta_b \end{cases}$$

A backward transition $(q, \mathfrak{m}, q')$ can violate the effect of all forward transitions on a path from $q'$ to $q$. Intuitively, this means that if the labeled message $\mathfrak{m}$ over

this backward transition occurs after the occurrence of labeled messages over the sequence of forward transitions on a path from $q'$ to $q$, then $\mathfrak{m}$ eliminates the effect of the occurrence of the labeled messages over these forward transitions.

**Definition 6 (vio-transition)** For the given sequence automaton $\mathcal{A}$, the vio-transitions of the transition $(q, \mathfrak{m}, q') \in \delta_f$, where $q \neq q'$, is the set of backward transitions that can violate the effect of $(q, \mathfrak{m}, q')$ in a path made up of only forward transitions from the destination to the source of the backward one:

$$vioTrns(\mathcal{A}, (q, \mathfrak{m}, q')) = \{(q_n, \mathfrak{m}', q_0) \mid (q_n, \mathfrak{m}', q_0) \in \delta_b \wedge q_0 \rightarrow^* q \wedge q' \rightarrow^* q_n\}$$

For example, in Fig. 2, $preTrns(\mathcal{A}_1, (q_0, \mathfrak{m}_2, q_3)) = \emptyset$ and $preTrns(\mathcal{A}_1, (q_3, \mathfrak{m}_4, q_4))$ and $preTrns(\mathcal{A}_1, (q_3, \mathfrak{m}_5, q_0))$ are equal to $(q_0, \mathfrak{m}_2, q_3)$. Furthermore, the transition $(q_3, \mathfrak{m}_5, q_0)$ violates the effect of $(q_0, \mathfrak{m}_2, q_3)$ and so $vioTrns(\mathcal{A}_1, (q_0, \mathfrak{m}_2, q_3)) = \{(q_3, \mathfrak{m}_5, q_0)\}$.

## 4 Choreography-Based Monitoring of Message-Based Systems

In our approach, the local monitors of the processes are organized as a network. As the property is decentralized, a monitor sends the verdicts of its local property to other monitors. Verdicts can be either pushed into or pulled from a monitor. We use pulling strategy for message-based systems to declare the verification result more accurately. We explain the reason with an example.

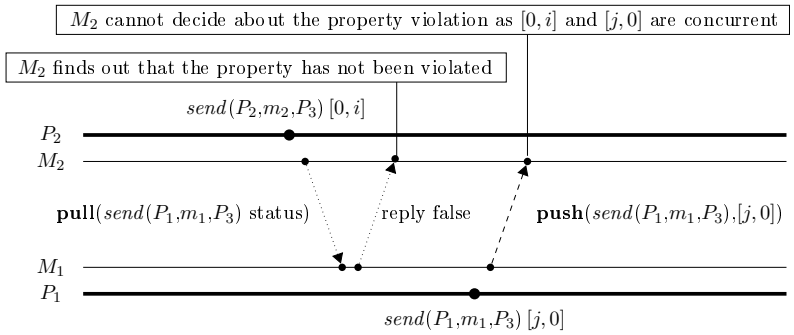



Fig. 3: The two communication protocols between the monitors where the pulling strategy is denoted by dotted lines and the pushing strategy is denoted by a dashed line. The vector clock $[0, i]$, where $i > 0$, denotes that $P_2$ executes the event $send(P_2, m_2, P_3)$ as the i$^{th}$ event while it has no information about the events of $P_1$

As we have no global clock, processes and monitors append their vector clocks to the events or communicated messages. Consider the property that the event $send(P_2, m_2, P_3)$ must never occur after the event $send(P_1, m_1, P_3)$. Assume that the process $P_1$ sends the message $m_1$ after the message $m_2$ has been sent, but the vector clocks of these messages are concurrent as depicted in Fig. 3. With a pushing strategy, the monitor of $P_1$, called $M_1$, must inform the monitor of $P_2$,

called $M_2$, the moment that the message $m_1$ has been sent, i.e., $[j,0]$. When the process $P_2$ sends the message $m_2$, $M_2$ cannot conclude about the violation of the property unless it has previously received the moment that the message $m_1$ was sent. After pushing the moment of $m_1$ by $M_1$, the monitor $M_2$ cannot conclude the order among the two events accurately and decide on the property as the vector clocks of the messages are concurrent, i.e., $[0,i] \parallel [j,0]$. However, with the pulling strategy, the monitor $M_2$ inquiries about the sending status of $m_1$ from the monitor $M_1$ after sending the message $m_2$. If $P_1$ has not sent the message $m_1$ yet, then $M_1$ responds with a false verdict. Upon receipt of this response, $M_2$ can conclude accurately that the property is not violated.

We use a two truth-value domain $\mathbb{B}_2 = \{\top, \bot\}$ for the verdict declaration, where $\top$ denotes no sequence of bad prefixes has been formed so far, and $\bot$ denotes at least one sequence of bad-prefixes has been formed. If it is not recognizable certainly whether a sequence is formed, we use the set $\{\top, \bot\}$ to denote a sequence is probably formed or probably not formed so far. In the following, first we decompose the message-based property specification among the monitors to have a decentralized specification. Finally, we explain our choreography-based runtime verification algorithm with the pulling strategy.

### 4.1 Decentralized Message-Based Property Specification

In this section, we aim to break down a message-based property specification into a decentralized specification among processes of a system. We assume bad-prefixes of a property can be specified by a sequence automaton, as we explained in Section 3. This automaton is broken down into a set of transition tables. Each table is maintained by a monitor and contains the set of transitions labeled by the messages that the corresponding process is their sender. For each transition, the set of its pre-transitions is also stored in the table. Since the effect of a pre-transition may be violated by the occurrence of its vio-transitions, it is necessary to store the set of vio-transitions for each pre-transition in the table too. A transition is uniquely identified in terms of the identifiers of its source/destination states. Self-loops are ignored in the transition tables, as they do not change the state of monitors.

**Definition 7 (Transition Tables)**
A sequence automaton $\mathcal{A} = (Q, \Sigma, \delta_f, \delta_b, Q_0, F)$ over $\Sigma = Msg$, is decomposed into the set of transition tables $\mathcal{T}_1, \ldots, \mathcal{T}_k$, where $k$ is the number of processes and $\mathcal{T}_x$ belongs to the process $x$. Each row of the table $\mathcal{T}_x$ is a quadruple of the form $(t, \mathit{fin}, \mathit{pre}, \mathit{vio})$ where $t = (q, \mathfrak{m}, q') \in \delta_b \cup \delta_f$ is a transition with the message label $\mathfrak{m} = (x, m, y)$ such that $x$ is the sender of the message, *fin* is a boolean value indicating whether the destination state of $t$ is a final state, $\mathit{pre} \in \mathit{preTrns}(\mathcal{A}, t)$, and $\mathit{vio} = \mathit{vioTrns}(\mathcal{A}, \mathit{pre})$. Note that *vio* is a set of transitions that its size is possibly greater than one. The transition tables $\mathcal{T}_1, \ldots, \mathcal{T}_k$ of a sequence automaton $\mathcal{A}$ is defined as below:

(1) $\forall\, (q,(x,m,y),q') \in \delta_f \wedge \forall\, \mathit{pre} \in \ \mathit{preTrns}(\mathcal{A},(q,(x,m,y),q'))$
$\Leftrightarrow ((q,(x,m,y),q'), q' \in F, \mathit{pre}, \mathit{vioTrns}(\mathcal{A},\mathit{pre})) \in \mathcal{T}_x$

(2) $\forall\, (q,(x,m,y),q') \in \delta_b \wedge \forall\, \mathit{pre} \in \mathit{preTrns}(\mathcal{A},(q,(x,m,y),q'))$
$\Leftrightarrow ((q,(x,m,y),q'), \bot, \mathit{pre}, \emptyset) \in \mathcal{T}_x$

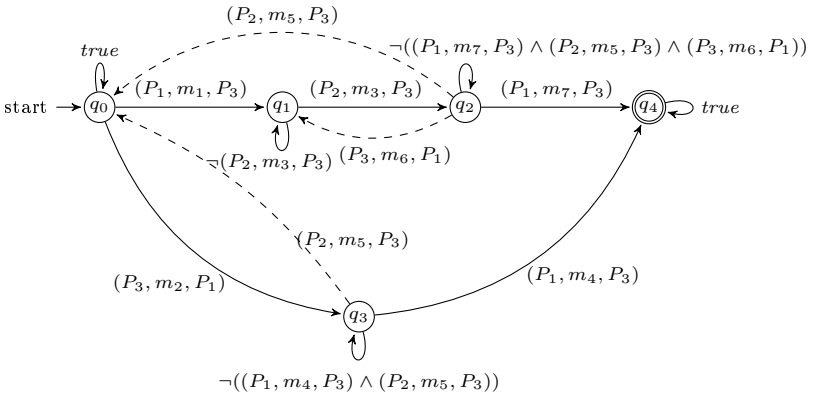



Fig. 4: The sequence automaton $\mathcal{A}_1$ with the explicit labeled messages

The first condition explains that for each forward transition in a sequence automaton $\mathcal{A}$ whose labeled message is $\mathfrak{m} = (x, m, y)$, then for each of its pre-transitions *pre* and the vio-transitions of *pre*, a row is included in $\mathcal{T}_x$. We remark that backward transitions never directly reach to a final state and no transition violates their effect. Thus, the second condition explains that for each backward transition and for each of its pre-transitions *pre*, a row is included in $\mathcal{T}_x$ with an empty vio-transition and the false value. To keep the communication among monitors secret and a monitor is not aware of the communicated messages between two processes, we omit the labeling message from *pre* and *vio* transitions and keep a reference to its sender instead. So, we store the information of pre-transition/vio-transition $(q, (y, m, z), q')$ as $(q, @\, y, q')$ where $@\, y$ is a reference to the process $y$. The unique values of $q$ and $q'$ are used by $y$ to identify the transition.

For instance, Fig. 4 shows a sequence automaton for a system of three processes $P_1$, $P_2$ and $P_3$. This automaton is decomposed into three tables Table 1, 2 and 3 which belong to $P_1$, $P_2$ and $P_3$, respectively. Table 1 contains information of the transitions whose sender of the labeled messages is $P_1$. The transition $(q_0, (P_1, m_1, P_3), q_1)$ does not reach to the final state and it has no pre-transition and so no corresponding vio-transition. So, the first row $((q_0, (P_1, m_1, P_3), q_1), \bot, \emptyset, \emptyset)$ is included in $\mathcal{T}_{P_1}$. The transition $(q_2, (P_1, m_7, P_3), q_4)$ reaches to the final state and has only one pre-transition as $(q_1, (P_2, m_3, P_3), q_2)$ and two corresponding vio-transitions of $(q_2, (P_2, m_5, P_3), q_0)$ and $(q_2, (P_3, m_6, P_1), q_1)$. The corresponding row of the transition $(q_2, (P_1, m_7, P_3), q_4)$ in $\mathcal{T}_{P_1}$ is :

$$((q_2, (P_1, m_7, P_3), q_4), \top, (q_1, @P_2, q_2), \{(q_2, @P_2, q_0), (q_2, @P_3, q_1)\}).$$

## 4.2 Monitoring Algorithm Design

In this section, we aim to introduce the choreography-based runtime verification algorithm for message-based systems, where the bad-prefixes of the intended property is specified on the sequences of *send* events. The detailed description of the algorithm will be illustrated after explaining the main idea of the algorithm.

Table 1: The transition table $\mathcal{T}_{P_1}$

| transition | final | pre-transition | vio-transition |
|---|---|---|---|
| $(q_0, (P_1, m_1, P_3), q_1)$ | $\bot$ | $\emptyset$ | $\emptyset$ |
| $(q_3, (P_1.m_4.P_3), q_4)$ | $\top$ | $(q_0, @\, P_3, q_3)$ | $\{(q_3, @\, P_2, q_0)\}$ |
| $(q_2, (P_1, m_7, P_3), q_4)$ | $\top$ | $(q_1, @\, P_2, q_2)$ | $\{(q_2, @\, P_2, q_0), (q_2, @\, P_3, q_1)\}$ |

Table 2: The transition table $\mathcal{T}_{P_2}$

| transition | final | pre-transition | vio-transition |
|---|---|---|---|
| $(q_1, (P_2, m_3, P_3), q_2)$ | $\bot$ | $(q_0, @\, P_1, q_1)$ | $\{(q_2, @\, P_2, q_0)\}$ |
| $(q_2, (P_2, m_5, P_3), q_0)$ | $\bot$ | $(q_1, @\, P_2, q_2)$ | $\emptyset$ |
| $(q_3, (P_2, m_5, P_3), q_0)$ | $\bot$ | $(q_0, @\, P_3, q_3)$ | $\emptyset$ |

Table 3: The transition table $\mathcal{T}_{P_3}$

| transition | final | pre-transition | vio-transition |
|---|---|---|---|
| $(q_0, (P_3, m_2, P_1), q_3)$ | $\bot$ | $\emptyset$ | $\emptyset$ |
| $(q_2, (P_3, m_6, P_1), q_1)$ | $\bot$ | $(q_1, @\, P_2, q_2)$ | $\emptyset$ |

*The Algorithm Sketch*

For a message-based system $D = \{P_1, \ldots, P_n\}$, we equip each process $P_x$ with a *monitor process* $M_x$ which owns a separate message queue. The process and its monitor have two shared variables $vc_x$ and $list_x$. The variable $vc_x$ is the vector clock of the process, and $list_x$ is the list of messages which have been sent by the process. We assume that the mutual exclusion of shared variables is ensured by using some well-known mechanisms like semaphore [27,33]. Monitors communicate with each other using monitoring messages. The process $P_x$ updates $vc_x$ according to Definition 2, but its monitor only updates $vc_x$ when it handles a monitoring message by taking the maximum value of $vc_x$ and the vector clock appended in the monitoring message. In a situation that processes rarely communicate with each other, updating $vc_x$ by monitors causes that processes gain some knowledge about each other.

The monitor $M_x$ maintains a transition table $\mathcal{T}_x$ as described by Definition 7 to verify the given message-based property in a choreography manner at runtime. A transition of $\mathcal{T}_x$ is called *enabled* if at least one of its preceding transitions has been enabled before and after that, no violating transition (of those preceding transition) disabled its effect. The monitor $M_x$ maintains a history that contains those transitions of $\mathcal{T}_x$ that was previously enabled. The time that a transition is enabled equals the vector clock appended to the message (over that transition) when it was sent.

When the process $P_x$ sends a message, it also puts the message into $list_x$. The monitor $M_x$ takes messages from $list_x$ and inspects if the message enables any transition by consulting with other monitors. After taking $\mathfrak{m}$, $M_x$ finds those rows of $\mathcal{T}_x$ whose labeled message on its transition equals $\mathfrak{m}$. For each row, $M_x$ inquires about the enabled status of the pre-transition and vio-transitions in the row by sending appropriate monitoring messages paired by the current value of its $vc_x$ to the monitors which are corresponded to the sender of the messages over these transitions. Some information is appended to the monitoring messages such as the

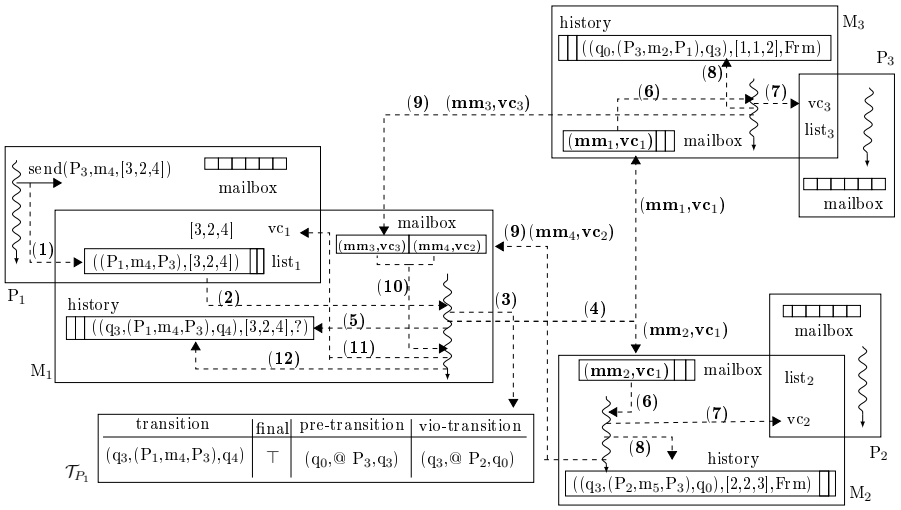



Fig. 5: The algorithm steps taken upon sending the message $m_4$ by the process $P_1$. The process $P_i$ and its monitor $M_i$ have an execution thread and a mailbox and the shared variables $vc_i$ and $list_i$

vector clock of the sending event of $\mathfrak{m}$, the inspected transition of $\mathcal{T}_x$ which is labeled by $\mathfrak{m}$, called $t$, and the inquired transitions. Then, $M_x$ adds the temporary record $(t, vc, ?)$ to its history. The triple $(t, vc, result)$ expresses that the enabled status of the transition $t$ that its labeled message was sent at the moment $vc$, is either under inspection or defined. The former case is indicated by the result value of "?" while the latter is indicated by the result values of $Frm$ or $Frm_p$ which are explained later. Adding the record $(t, vc, ?)$ is helpful when another monitor inquires $M_x$ about the enabled status of the transition $t$. In such cases, the monitor $M_x$ must postpone its response to the inquiry until the result of the transition $t$ be defined.

Fig. 5 illustrates the steps of our algorithm when applied on the example in Fig. 4 where $P_1$ sends a message $m_4$ to $P_3$. It is assumed that the vector clock of $P_1$ equals $[3, 2, 4]$ upon sending this message. As in Fig. 5, $P_1$ puts the message $(P_1, m_4, P_3)$ paired by the vector clock $[3, 2, 4]$ in the shared list $list_1$ and then the monitor $M_1$ handles it **(1)**-**(2)**. The monitor $M_1$ checks the transition table $\mathcal{T}_{P_1}$ and finds a row with a transition $(q_3, (P_1, m_4, P_3), q_4)$ **(3)**. Then, $M_1$ prepares two monitoring messages $\mathbf{mm}_1$ and $\mathbf{mm}_2$ paired by the current value of $vc_1$ to inquire about the enabled status of the pre-transition $(q_0, @P_3, q_3)$ and the vio-transition $(q_3, @P_2, q_0)$ from $M_3$ and $M_2$, respectively **(4)**. Then, $M_1$ adds the triple $((q_3, (P_1, m_4, P_3), q_4), [3, 2, 4], ?)$ to its history **(5)**.

Upon receiving the monitoring message, $M_y$ first updates $vc_y$ according to the vector clock paired by the received message. If $M_y$ has an unknown result "?" about the inquired transition in its history or an unhandled message in its $list_y$ which is corresponded to the labeled message on the inquired transition, for a mo-

ment before the sending moment of $\mathfrak{m}$ appended in the monitoring message, then it must postpone responding to the monitoring message. Otherwise, if $M_y$ finds a record with the defined result value about the inquired transition, it infers that the transition has been previously enabled. If such a row is found, $M_y$ attaches the corresponding information found in its history to its response monitoring message. Otherwise, it attaches an empty set to the monitoring message. Then, $M_y$ communicates with $M_x$ by sending the response monitoring message paired by the current value of its $vc_y$.

For example, in Fig. 5, the monitor $M_3$ receives the monitoring message $\mathbf{mm}_1$ paired by the vector clock $vc_1$ and then updates the shared vector clock $vc_3$ **(6)**-**(7)**. Then, it checks its history to see if there exists any information about the inquired transition $(q_0, @P_3, q_3)$ **(8)**. As it finds an information in its history, the monitor $M_3$ prepares a monitoring message $\mathbf{mm}_3$ which contains the record found in its history, and sends $\mathbf{mm}_3$ paired by the current value of $vc_3$ to $M_1$ **(9)**.

When $M_x$ receives all responses from other monitors, it updates the value of $vc_x$ according to the values of the received vector clocks in the response messages. Then, it checks whether the inspected transition can be enabled. The inspected transition $t$ can be enabled when its pre-transition was enabled, and the time that it was enabled is not after the occurrence of $\mathfrak{m}$ and also is not before its enabled vio-transitions. In this case, $M_x$ updates the result value of the corresponding record of this transition in its history. Otherwise, it removes the record $(t, vc, ?)$ from its history and adds nothing to the history. If there exists at least one pre-transition for the transition $t$ that it was not disabled by its vio-transitions and its enabled time is before the occurrence of $\mathfrak{m}$, it is concluded that a bad-prefix is going to be formed. So, the result value of the corresponding record of $t$ in the history is updated to "*Frm*". This result denotes that a sequence as a bad-prefix of the property is going to be formed. In the case that the enabled time of the pre-transition is concurrent by the enabled time of the transition $t$, then the monitor decides conservatively and updates the result value of the corresponding record of $t$ to "$Frm_p$". This result denotes that due to the concurrent occurrence of the events, the bad-prefix probably may be formed. It is noteworthy that if the transition of a row has no pre-transition, the monitor $M_x$ does not consult with any monitor and adds this transition with the result of "*Frm*" to the history. Finally, if the transition reaches a final state, then the monitor declares the verdict of $\perp$ or $\{\perp, \top\}$ depending on the result appended to the transition. The verdict $\perp$ denotes that a bad-prefix has been formed and the appended result to the inspected transition is *Frm* while $\{\top, \perp\}$ denotes that a bad-prefix either may be formed and the appended result of this transition is $Frm_p$ or it may be not formed so far.

In Fig. 5, $M_1$ handles the received monitoring messages and updates the shared vector clock $vc_1$ according to the values of the received vector clocks $vc_2$ and $vc_3$ **(10)**-**(11)**. The monitor $M_1$ decides about the enabled status of $(q_3, (P_1, m_4, P_3), q_4)$ when it receives the both monitoring messages $\mathbf{mm}_3$ and $\mathbf{mm}_4$. The message $\mathbf{mm}_3$ contains the record $((q_0, (P_3, m_2, P_1), q_3), [1, 1, 2], Frm)$ and $\mathbf{mm}_4$ contains the record $((q_3, (P_2, m_5, P_3), q_0), [2, 2, 3], Frm)$. As the enabled time of the vio-transition (i.e., $[2, 2, 3]$) is after the enabled time of the pre-transition (i.e., $[1, 1, 2]$), $M_1$ finds out that the effect of the pre-transition has been eliminated. As a result, $M_1$ removes $((q_3, (P_1, m_4, P_3), q_4), [3, 2, 4], ?)$ from its history as the bad-prefix has not been formed **(12)**.

*Detailed Description of the Algorithm*

We explain the detailed description of the algorithm sketched above. In Algorithm 1, each monitor process first initializes its local variables by invoking the procedure *Init* and then enters an infinite loop in which it handles the messages stored in the shared variable $list_x$ by calling the procedure *RcvRegularMsg*, and handles the messages of the monitoring queue *mqueue* by calling the procedure *RcvMonitoringMsg*. The procedure *Init* initializes the local variables *history*, *rcvHist*, *tmphist*, *viohist*, *mmlist*, and *reqTrans* described below.

**Algorithm 1** Monitor Process $M_x$

```
   Input: transition table T_x
 1: Init();
 2: while true do
 3:     while ¬isEmpty(mqueue) do
 4:         RcvMonitoringMsg(head(mqueue));
 5:         mqueue := tail(mqueue);
 6:     if ¬isEmpty(list_x) then
 7:         RcvRegularMsg(head(list_x));
 8:         list_x := tail(list_x);
 9: ------------------------------------------------------------
10: procedure Init
11:     history := ∅; rcvHist := ∅; tmphist := ∅; viohist := ∅; mmlist := ∅; reqTrans := ∅;
12: end procedure
```

- *history* is a vector representing information of $\mathcal{T}_x$ transitions which have been previously enabled and have led to the formation of a bad-prefix.
- *rcvHist* is a mapping that for each pair of a transition and the associated vector clock maintains received information from other monitors.
- *tmphist* and *viohist* are temporary sets keeping transitions and vio-transition of $\mathcal{T}_x$ which have been previously enabled. These sets are processed after getting all the responses from other monitors using *rcvHist* to update *history*. These temporary sets become empty after updating *history*.
- *mmlist* is the list of monitoring messages $M_x$ is going to handle in the future.
- *reqTrans* is a mapping that for each pair of a transition and the associated vector clock maintains the set of transitions that a monitor must inquire about their enabled status.

Regular Message Handler (Algorithm 2)

The goal of the procedure *RcvRegularMsg*, parameterized by a message *msg* and a vector clock *vc*, is to check whether the sent message *msg* can enable any transition of $\mathcal{T}_x$'s rows. The value of vector clock *vc* specifies the time that the message *msg* was sent. To this aim, it first finds the rows like $(t, fin, pre, vio)$ such that the transition $t$ is labeled by *msg*. For each row, in line 4, $M_x$ stores its pre-transition and vio-transition in the mapping *reqTrans*. For each pair $(t, vc)$, *reqTrans* maintains the set of transitions that $M_x$ must inquire about their enabled status to decide about the enabled status of $t$. We remark that the enabled status of the transition $t$ may be evaluated for different values of *vc*.

**Algorithm 2** Regular Message Handler in $M_x$

```
1:  procedure RcvRegularMsg(regularMsg msg, vectorClock vc)
2:      trnsProcess := ∅; trnsAll := ∅;
3:      for each (t, fin, pre, vio) ∈ T_x do
4:          if t.m = msg then reqTrans := reqTrans ∪ {(t, vc) ↦ {pre ∪ vio}};
5:      for each (t, vc) ∈ reqTrans do
6:          if reqTrans(t, vc) ≠ ∅ then
7:              trnsAll := reqTrans(t, vc);
8:              for each (q_i, @y, q_i') ∈ trnsAll do
9:                  mm := new monitorMsg();
10:                 mm.owner := M_x; mm.target := M_y; mm.vc := vc; mm.respondRecs := ∅;
11:                 trnsProcess := trnsProcess ∪ {(q_i, @y, q_i')};
12:                 for each (q_j, @y, q_j') ∈ trnsAll do
13:                     trnsProcess := trnsProcess∪{(q_j, @y, q_j')}; remove((q_j, @y, q_j'), trnsAll);
14:                 mm.targetTrans := trnsProcess; mm.originTrans := t;
15:                 send(M_y, mm, vc_x);
16:                 trnsProcess := ∅;
17:             history := history ∪ (t, vc, ?);
18:         else history := history ∪ (t, vc, Frm);
19: end procedure
```

If the set of inquired transitions for each pair $(t, vc)$ is not empty, then $M_x$ must prepare a monitoring message for each inquired transition. Each monitoring message $mm$ is of *monitorMsg* data structure with the following fields:

- *owner* is the monitor that creates $mm$,
- *target* is the monitor that $mm$ must be sent to,
- $vc$ is the vector clock upon sending *msg*,
- *originTrans* is the transition that we are inspecting its enabled status,
- *targetTrans* is the list of inquired transitions with the same *target*, and
- *respondRecs* is the set of records corresponding to the enabled status of the inquired transitions maintained by the history of *target*.

For each inquired transition, in lines $8 - 11$, a monitoring message $mm$ is prepared and the four fields *owner*, *target*, $vc$, and *respondRecs* are set. In lines $12 - 13$, all inquired transitions with the same target for whom the monitoring messages is prepared, are added to the local variable *trnsProcess*. The fields of *targetTrans* and *originTrans* are set, in line 14, regarding this local variable and the transition $t$. Finally, in line 15, $M_x$ sends the pair of the monitoring message and the current vector clock of the process $P_x$ to the target monitor. In line 17, it adds the temporary record $(t, vc, ?)$ to its history.

When the set of inquired transitions for $t$ is empty, it indicates that *msg* is the start of a sequence. So, in line 18, this transition together with its corresponding information is added as the triple $(t, vc, Frm)$ to the history.

Monitoring Message Handler (Algorithm 3)

In the procedure *RcvMonitorMsg*, parameterized by a pair of the received monitoring message $mm$ and the vector clock $vc_r$, $M_x$ first updates the shared variable $vc_x$, in line 2, with regard to the vector clock $vc_r$. Then, it examines whether the monitor $M_x$ is the owner of the monitoring message $mm$.

**Algorithm 3** Monitoring Message Handler in $M_x$

```
1:  procedure RcvMonitorMsg(monitorMsg mm, vectorClock vc_r)
2:      vc_x := updVc(vc_x, vc_r);
3:      t := mm.originTrans; inquiredTrans := mm.targetTrans;
4:      if mm.owner = M_x then
5:          rcvHist := rcvHist ∪ {(t, mm.vc) ↦ mm.history};
6:          reqTrans := reqTrans \ {(t, mm.vc) ↦ inquiredTrans};
7:          if reqTrans(t, mm.vc) = ∅ then
8:              for each (t, fin, pre, vio) ∈ T_x do
9:                  for each t' ∈ vio do
10:                     recs := getRecs(t', rcvHist(t, mm.vc));
11:                     viohist := viohist ∪ recs;
12:                     rcvHist := rcvHist \ recs;
13:             recs := rcvHist(t, mm.vc);
14:             for each r ∈ recs do
15:                 EvalVc(t, mm.vc, r);
16:             EvalHist(t, mm.vc);
17:     else if mm.target = M_x then
18:         for each t'' ∈ inquiredTrans do
19:             if UnknownRsult(t'', mm.vc, history) ∨ UnhandleMsg(t'', mm.vc, list_x) then
20:                 mm_new := new monitorMsg(); mm_new := mm;
21:                 mm_new.targetTrans := t''; mmlist := mmlist ∪ mm_new;
22:                 inquiredTrans := inquiredTrans \ t'';
23:             else
24:                 recs := recs ∪ getRecs(t'', mm.vc, history) ∪ makeRecs(t'', {⊤, ⊥});
25:         mm.respondRecs := recs; mm.targetTrans := inquiredTrans;
26:         send(mm.owner, mm, vc_x);
27: end procedure
```

If $M_x$ is the owner, then it first updates its relevant mappings and then examines whether it can decide about the enabled status of the inspecting transition $t = mm.originTrans$. The data structure *rcvHist* is a mapping which keeps the received information *mm.respondRecs* for the inquired transitions *mm.targetTrans* from inquired monitors as a consequence of inspecting the transition $t$ at the moment *mm.vc*. We remark that the received information may contain different records for an inquired transition as the inquired transition may be previously enabled at different times. For each pair $(t, vc)$, if $M_x$ receives all responses about inquired transitions, then it evaluates the enabled status of $t$. To this aim, it first separates the information about the pre-transition from the vio-transition maintained by *rcvHist*. To find the information about vio-transition, in lines $8-12$, it first finds those rows like $(t, fin, pre, vio)$ of the table $\mathcal{T}_x$ such that there is any information about *vio* transitions in *rcvHist*. This information is copied into *viohist*, in line 11. We remark that if there is no information about an inquired transition in *rcvHist*, then it can be concluded that the inquired transition was not previously enabled. Since the records of vio-transition are removed from the mapping *rcvHist*, the remaining records of this mapping correspond to the pre-transition of $t$. In lines $13-16$, $M_x$ evaluates the enabled status of $t$ regarding the pre-transition of $t$. To this aim, in line 15, it evaluates the enabled statues of $t$ regarding one of its pre-transition. As a result of executing the procedure *EvalVc*, which is explained later, some temporary triple about the transition $t$ is added to the temporary vector *tmphist*. After investigating the information of all pre-transition of $t$ in *rcvHist*, in line 16, $M_x$ decides about the enabled status of

$t$ and whether the corresponding information of $t$ must be added to the history. The procedure *EvalHist*, which is explained later, aggregates the information of *tmphist* and updates the information of its *history*.

If $M_x$ is not the owner, in line 17, then it must respond to the owner of $mm$ that has inquired $M_x$ about the transitions $mm.targetTrans$, assigned to the local variable *inquiredTrans*. As *inquiredTrans* may have more than one transition, in lines $18-26$, $M_x$ investigates the enabled status of each inquired transition $t''$. If $M_x$ has an unknown result "?" about the inquired transition $t''$ in its history or an unhandled message $t''.\mathfrak{m}$ in its $list_x$ for a moment before $mm.vc$, then it must postpone responding to $mm$. The first and second conditions are examined by calling the functions $UnknownRsult(t'', mm.vc, history)$ and $UnhandledMsg(t'', mm.vc, list_x)$, respectively. The first condition indicates that $M_x$ is in the middle of asking from other monitors to decide on the enabled status of $t''$. The second condition implies that $P_x$ has sent the message labeled on $t''$ and put this message into the shared $list_x$, but its monitor $M_x$ has not processed it yet. So, the monitoring message $mm$ is added to its pending monitoring messages *mmlist* to be responded later. Otherwise, those records in its history that belongs to the moments before or concurrent to $mm.vc$ are retrieved from the history by calling the overloaded function $getRecs(t'', mm.vc, history)$. If there are either the records with an unknown result "?" about the inquired transition $t''$ in the history or an unhandled message $t''.\mathfrak{m}$ in $list_x$ for a moment concurrent with $mm.vc$, then $M_x$ understand that it must respond conservatively. Hence, for each record $(t'', vc, ?)$ in its history and each unhandled message of $t''.\mathfrak{m}$ in $list_x$ with the appended vector clock $vc$, it prepares a record $(t'', vc, \{\top, \bot\})$ by calling the function $makeRecs(t'', \{\top.\bot\})$.

```
1: procedure EvalVc(transition t, vectorClock v, record r)
2:     vc := getVc(r); rsult := getRsult(r); pre := getTrans(r);
3:     status := NotViolate(t, pre, vc);
4:     if status = true then
5:         if (v ∥ vc) then tmphist := tmphist ∪ (t, v, Frm_p);
6:         else if (vc ⇝ v) then tmphist := tmphist ∪ (t, v, rsult);
7:     else if status = true_p then
8:         if (v ↛ vc) then tmphist := tmphist ∪ (t, v, Frm_p);
9: end procedure
```

The goal of the procedure *EvalVc* is to evaluate the enabled status of a transition regarding one of its pre-transition. This procedure is parameterized by a transition $t$ and a moment $v$ that the message labeled on $t$ was sent and a record $r$ that corresponds to the history information of a pre-transition of $t$. The pre-transition, the corresponding vector clocks, and the result are retrieved from the record $r$ in line 2. In line 3, it checks whether the pre-transition *pre* has been disabled by one of its vio-transition. The procedure *NotViolate*, which is explained later, returns "*true*" if the pre-transition *pre* has not been disabled, "*false*" if the pre-transition *pre* has been disabled, and "$true_p$" if it is not clear whether *pre* has been disabled (i.e., the moment that the message labeled on *pre* was sent is concurrent with the moment that the message labeled on vio-transition of *pre* was sent). If *pre* has not been disabled, in lines $4-8$, $M_x$ checks the enabled status of $t$ with regards to *pre*. If $v$ is concurrent by $vc$, then it is not recognizable whether

*pre* has been enabled before or after $t$. In this case, the monitor behaves in a conservative manner and adds the record $(t, v, Frm_p)$ to the temporary history *tmphist*. Otherwise, if $vc$ is less than $v$, it can be concluded that the message of *pre* has been sent before the message of $t$, as in lines $7-8$. However, the final result for the transition $t$ is defined by the result of *pre*; If the result of *pre* is "*Frm*", then it can be concluded that the sequence has been formed so far. So, the final result of $t$ must be "*Frm*" as *pre* has been enabled before $t$. However, if the result of *pre* is $Frm_p$, then it is not recognizable whether the sequence up to the transition $t$ has been formed. Despite the enabling of *pre* before $t$, the result $Frm_p$ must be considered for the transition $t$. In the case that *pre* has been disabled by its vio-transition (i.e., the function *NotViolate* returns "*false*") or the message of $t$ has been sent before the message of *pre* (i.e., $v \rightsquigarrow vc$), nothing is added to *tmphist* as the sequence formation is corrupted. If it is not clear whether *pre* has been disabled (i.e., the function *NotViolate* returns "$true_p$"), in line 9, $M_x$ checks the enabled status of $t$ with regards to *pre*. If $v$ is not less than $vc$, the monitor behaves in a conservative manner and adds information of the transition $t$ with the result value of $Frm_p$ to the temporary history *tmphist* as $M_x$ does not know certainly whether *pre* has been disabled previously.

```
1:  function NotViolate(transition t, transition pre, vectorClock vc^pre)
2:      status := true;
3:      for each (t, fin, pre, vio) ∈ T_x do
4:          for each t' ∈ vio do
5:              recs := getRecs(t', viohist);
6:              for each r ∈ recs do
7:                  vc^vio := getVc(r);
8:                  if vc^pre ⇝ vc^vio then status := false; break;
9:                  else if  vc^vio ∥ vc^pre then status := true_p;
10:         if status = false then break;
11:     return status;
12: end function
```

The goal of the procedure *NotViolate* is to check whether the given transition *pre* has been disabled by one of its vio-transition. This procedure is parameterized by the transitions $t$, *pre*, and a moment $vc^{pre}$ that *pre* has been enabled. In lines $3-5$, for each row $(t, fin, pre, vio)$, $M_x$ finds those transitions of *vio* that there exists some information about them in *viohist*. In lines $6-9$, for each vio-transition, it is checked whether the transition *pre* has been disabled by this vio-transition. If *pre* has been enabled before *vio*, then it can be concluded that *pre* has been disabled by *vio* and so *status* is set to *false*. If *pre* is concurrent with *vio*, then the order between *pre* and *vio* is not determined, and *status* is set to $true_p$.

Given a transition $t$ and a moment $vc$ that the message on $t$ was sent, the procedure *EvalHist* updates the information of the transition $t$ in the history with regard to the information gathered in *tmplist*. If relevant information of $t$ exists in the temporary history *tmphist* (when at least one of its pre-transition has been enabled), then they are retrieved and are set to the local variables *recs*. If there is a record with the result value "*Frm*" for $t$, then it is concluded that a sequence is going to be formed. Thus, $M_x$ updates the temporary record $(t, vc, ?)$ to $(t, vc, Frm)$ in *history*, in line 6. Otherwise, the result values of relevant records for

```
1: procedure EvalHist(transition t, vectorClock vc)
2:     if contain(t, tmphist) then
3:         recs := getRecs(t, tmphist);
4:         if hasRsult(t, vc, Frm, tmphist) then updHist((t, vc, Frm), history);
5:         else updHist((t, vc, Frm_p), history);
6:         if ReachtoFinalState(t) = true then Declare(getVrdct(t));
7:         tmphist := tmphist \ recs;
8:     else history := history \ (t, vc, ?);
9:     HndlPndMonitorMsg(t);
10: end procedure
```

$t$ are definitely "$Frm_p$" and it is not recognizable whether the sequence has been formed until enabling of $t$. Hence, in line 7, $M_x$ updates the temporary record $(t, vc, ?)$ to $(t, vc, Frm_p)$. If the transition $t$ reaches to the final state, the verdict is declared, in line 8, by calling the function *Declare*. If there is no records for $t$ in *tmphist*, then it is concluded that none of its pre-transition have been enabled or they may be disabled by their vio-transitions. In this case, the transition $t$ is not enabled and its temporary record with the result value "?" is removed from *history*, in line 10. Finally, the monitor $M_x$ handles those pending monitoring messages of *mmlist* waiting for the status of the transition $t$ by calling the procedure *HndlPndMonitorMsg*.

```
1: procedure HndlPndMonitorMsg(transition t)
2:     for each mm ∈ mmlist do
3:         if mm.targetTrans = t then
4:             if ¬UnknownRsult(t, mm.vc, history) ∧ ¬ UnhandleMsg(t, mm.vc, list_x) then
5:                 mm.respondRecs := getRecs(t, mm.vc, history) ∪ makeRecs(t, {⊤, ⊥});
6:                 send(mm.owner, mm, vc_x);
7:         mmlist := mmlist \ mm;
8: end procedure
```

The procedure *HndlPndMonitorMsg*, parameterized by a transition $t$, handles the monitoring messages of *mmlist* that their target transition is $t$. If there is no temporary record in *history* or an unhandled message in $list_x$ for a moment before $mm.vc$, then $M_x$ handles the pending monitoring messages of *mmlist*. In this case, in lines $5-6$, $M_x$ retrieves the relevant records of $t$ that is belongs to the moment before or concurrent to $mm.vc$ from its history. If there are some temporary records in *history* or an unhandled message in $list_x$ for a moment concurrent with $mm.vc$, $M_x$ adds the conservative responses by adding the corresponding records with the result value of $\{\top, \bot\}$. Finally, it appends the retrieved information to the monitoring message and sends the pair of $mm$ and its $vc_x$ to the owner of $mm$.

## 5 Soundness and Completeness of the Algorithm

To provide the *soundness* and *completeness* results for our algorithm, we use the following lemma indicating that the vector clock manipulation by the monitors in our algorithm preserves the partial orders of the total order.

**Lemma 1** *The event $e_i$ has occurred before the event $e_j$ in the total order, denoted by $e_i \rightarrow e_j$, if and only if $\neg(vc(e_j) < vc(e_i))$.*

*Proof* To illustrate the correctness of this lemma, we first prove $e_i \rightarrow e_j \Rightarrow \neg(vc(e_j) < vc(e_i))$ by contradiction. Assume that the contradictory of the assertion is correct and $vc(e_j) < vc(e_i)$. We consider two cases:

- The events $e_i$ and $e_j$ have occurred in one process $P_x$: The process $P_x$ and its monitor $M_x$ update the shared vector clock $vc_x$ increasingly with the difference that $M_x$ does not update $vc_x[x]$. As we assume that $vc(e_j) < vc(e_i)$ is correct, so the event $e_j$ has occurred before the event $e_i$. Therefore, $e_j$ must appear before $e_i$ in the total order as the events of a process is totally ordered. However, this is contradiction by the assumption that $e_i \rightarrow e_j$.
- The events $e_i$ and $e_j$ have occurred in different processes $P_x$ and $P_y$, respectively. If $P_x$ does not communicate with $P_y$ between the execution of $e_i$ and $e_j$, then there will be no relation between $vc(e_i)$ and $vc(e_j)$, i.e., $e_i \parallel e_j$. As $vc(e_j) < vc(e_i)$, it is concluded that $M_y$ communicates with other monitors after the occurrence of $e_j$, and it propagates the shared vector clock $vc_y$. So, two events $e_i$ and $e_j$ are causally related by the communication between the monitors. This causality must be preserved in the total order, i.e., $e_j \rightarrow e_i$, and this is a contradiction.

Now, we prove the correctness of $\neg(vc(e_j) < vc(e_i)) \Rightarrow e_i \rightarrow e_j$ by contradiction. Assume $\neg(vc(e_j) < vc(e_i))$, but $e_j \rightarrow e_i$. We consider two cases:

- The events $e_i$ and $e_j$ have occurred in one process: Since the events of a process are totally ordered, the event $e_j$ must occur before the event $e_i$. We explained before that the shared vector clock of a process is updated increasingly. So, we have $vc(e_j) < vc(e_i)$ and this is a contradiction.
- The events $e_i$ and $e_j$ have occurred in different processes $P_x$ and $P_y$, respectively: If $e_i$ and $e_j$ are causally related, there will be some communications between the processes or their monitors. In the case that there is no communication between $P_x$ and $P_y$, the monitors $M_x$ and $M_y$ may communicate with each other and update the vector clocks of $P_x$ and $P_y$. In this case, the event $e_i$ and $e_j$ may be causally related as the monitors update the shared vector clocks increasingly. As $e_j$ occurs before $e_i$ in the total order and the causally relation must be preserved by the total order, it is concluded that the monitors $M_y$ and $M_x$ update their shared vector clock after the execution of $e_j$ and before the execution of $e_i$ and so $vc(e_j) < vc(e_i)$. However, this is in contradiction to the assumption. It is noteworthy that if $P_x/M_x$ does not communicate with $P_y/M_y$, there will be no relation between the vector clocks of $e_i$ and $e_j$. □

As we consider regular safety properties, our algorithm only declares a verdict when the property is violated. We remark that our algorithm is choreography-based and monitors detect the violation of different sequences. Hence, only a subset of monitors may declare the violation of the property.

**Theorem 1 (Soundness)** *Given the sequence automaton $\mathcal{A}$, if the total order $\gamma$ of events can be somehow constructed such that $\gamma \in L(\mathcal{A})$, then the false verdict is declared as one of the results of the corresponding monitor of the transition reaching the final states of $\mathcal{A}$.*

*Proof* It is trivial that if $\gamma \in L(\mathcal{A})$, then there exists at least a sub-sequence $\mathfrak{m}_0^i \mathfrak{m}_1^j \mathfrak{m}_2^w \dots \mathfrak{m}_l^h \dots \mathfrak{m}_n^k$, called $\widehat{\gamma}$, in the total order $\gamma$ such that $\widehat{\gamma} \in L(\mathcal{A})$ where for the message $\mathfrak{m}_l^h$, $h$ is the index of the message in $\gamma$ and $l$ is the index of the message in $\widehat{\gamma}$, i. e., $0 \leq l \leq n$. Furthermore for each pair of $\mathfrak{m}_{l'}^{i'} \mathfrak{m}_{l'+1}^{j'}$ of $\widehat{\gamma}$, there is no message $\mathfrak{m}_o$ in $\gamma$, where $i' < o < j'$, that eliminates the effect of $\mathfrak{m}_{l'}^{i'}$. In other words, the messages of $\widehat{\gamma}$ constitute a path of only forward transitions from the initial state of $\mathcal{A}$ to the final state, and for each pair of $\mathfrak{m}_{l'}^{i'} \mathfrak{m}_{l'+1}^{j'}$, $\mathfrak{m}_{l'}^{i'}$ occurs as the label of a pre-transition of the transition carrying $\mathfrak{m}_{l'+1}^{j'}$. We show that the corresponding monitor of $\mathfrak{m}_k$, has the *false* verdict as one of its result.

By Lemma 1, it can be concluded that either $\mathfrak{m}_0^i$ has happened before $\mathfrak{m}_1^j$ or $\mathfrak{m}_0^i$ is concurrent with $\mathfrak{m}_1^j$ i.e., $\mathfrak{m}_1^j \not\rightsquigarrow \mathfrak{m}_0^i$ in short. By running our algorithm, in the procedure *RcvRegularMsg*, the monitor of $\mathfrak{m}_1^j$, namely $M_1$, checks the enabled status of its pre-transitions and the vio-transitions of the pre-transitions. So, a transition labeled by $\mathfrak{m}_0^i$, called $t$, and its corresponding vio-transitions are inquired. If a message belonging to the vio-transitions of $t$, called $\mathfrak{m}_v$, has occurred in $\gamma$, it must have occurred before $\mathfrak{m}_0^i$, where $v < i$, in $\gamma$ due to our condition on the sub-sequence. According to Lemma 1, two cases can be distinguished: either the message $\mathfrak{m}_v$ has happened before $\mathfrak{m}_0^i$ or $\mathfrak{m}_v$ is concurrent with $\mathfrak{m}_0^i$. In the first case, in the function *NotViolate*, where the vector clock of $\mathfrak{m}_v$ is less than the vector clock of $\mathfrak{m}_0^i$, the effect of $\mathfrak{m}_i$ has not been eliminated by $\mathfrak{m}_v$. So, $M_1$ checks the vector clocks of $\mathfrak{m}_0^i$ and $\mathfrak{m}_1^j$, in the procedure *EvalVc*. If $\mathfrak{m}_0^i$ has happened before $\mathfrak{m}_1^j$, $M_1$ concludes that the sequence $\mathfrak{m}_0^i \mathfrak{m}_1^j$ has been formed so far and stores the "*Frm*" result in its history. If there is no relation between the vector clocks of $\mathfrak{m}_0^i$ and $\mathfrak{m}_1^j$, $M_1$ behaves conservatively and stores the result value "$Frm_p$" in its history. In the second case, where the vector clock of $\mathfrak{m}_0^i$ is concurrent with $\mathfrak{m}_v$, $M_1$ does not know whether $\mathfrak{m}_v$ has eliminated the effect of $\mathfrak{m}_0^i$ and so it adds the result value "$Frm_p$" to its history in the procedure *EvalVc* since $\mathfrak{m}_1^j \not\rightsquigarrow \mathfrak{m}_0^i$.

Up to here, the result with "*Frm*" or "$Frm_p$" value has been correctly inserted into the $M_1$'s history by the procedure *EvalHist*. With the same discussion, we select $\mathfrak{m}_2^w$ of $\widehat{\gamma}$ and assume that the message $\mathfrak{m}_{v'}$, which eliminates the effect of $\mathfrak{m}_1^j$, has occurred and due to our condition on the sub-sequence, it must have occurred before $\mathfrak{m}_1^j$ in $\gamma$. By Lemma 1, $\mathfrak{m}_1^j$ has happened before $\mathfrak{m}_2^w$ or $\mathfrak{m}_1^j$ is concurrent with $\mathfrak{m}_2^w$. By applying our algorithm, the monitor of $\mathfrak{m}_2^w$, namely $M_2$, inquiries about $\mathfrak{m}_1^j$ and $\mathfrak{m}_{v'}$. The monitor $M_2$ compares the received information about $\mathfrak{m}_1^j$ and $\mathfrak{m}_{v'}$ and decides whether $\mathfrak{m}_{v'}$ eliminates the effect of $\mathfrak{m}_1^j$. If $\mathfrak{m}_{v'}$ has not eliminated the effect of $\mathfrak{m}_1^j$, $M_2$ checks the vector clocks of $\mathfrak{m}_1^j$ and $\mathfrak{m}_2^w$. If $\mathfrak{m}_1^j$ has happened before $\mathfrak{m}_2^w$, then $M_2$ stores the result value "$Frm_p$" or "*Frm*" sent by $M_1$ to the history. If there is no relation between the vector clocks of $\mathfrak{m}_1^j$ and $\mathfrak{m}_2^w$, it behaves conservatively and stores the "$Frm_p$" result to its history. These scenarios will be continued to reach the final message $\mathfrak{m}_k$ and the "*Frm*" or "$Frm_p$" results values have been correctly propagated to the monitor of the message $\mathfrak{m}_k$ and it declares the false verdict.

**Theorem 2 (Completeness)** *Among the set of verdicts computed by the local monitors, at least one of the results corresponds to the total order $\gamma$.*

*Proof* We distinguish two cases based on whether $\gamma \in L(\mathcal{A})$ and prove the theorem by contradiction:

- $\gamma \in L(\mathcal{A})$: The assertion $\gamma \in L(\mathcal{A})$ holds if and only if there exists at least a sub-sequence $\mathfrak{m}_0^i\mathfrak{m}_1^p \dots \mathfrak{m}_l^h \dots \mathfrak{m}_{n-2}^w\mathfrak{m}_{n-1}^j\mathfrak{m}_n^k$, called $\widehat{\gamma}$, in the total order $\gamma$ such that $\widehat{\gamma} \in L(\mathcal{A})$ where for the message $\mathfrak{m}_l^h$, $h$ is the index of the message in $\gamma$ and $l$ is the index of the message in $\widehat{\gamma}$, i.e., $0 \leq l \leq n$. Furthermore, for each pair of $\mathfrak{m}_{l'}^{i'}\mathfrak{m}_{l'+1}^{j'}$ of $\widehat{\gamma}$, there is no message $\mathfrak{m}_o$ in $\gamma$, where $i' < o < j'$, that eliminates the effect of $\mathfrak{m}_{l'}^{i'}$. The element $\mathfrak{m}_{n-1}^j$ occurs as the label of a pre-transition of the transition carrying $\mathfrak{m}_n^k$ which leads to a final state. By running our algorithm, in the procedure *RcvRegularMsg*, the monitor of $\mathfrak{m}_n^k$, called $M_1$, checks the enabled status of its pre-transitions and the vio-transitions of the pre-transitions. So, a transition labeled by $\mathfrak{m}_{n-1}^j$ and its corresponding vio-transition labeled by $\mathfrak{m}_v$ are inquired. By contradiction, we assume that $M_1$ has not declared any result. Three cases can be considered: (1) The monitor $M_1$ receives information about the messages $\mathfrak{m}_{n-1}^j$ and $\mathfrak{m}_v$, in the procedure *RcvMonitorMsg*, and finds that the message $\mathfrak{m}_v$ has eliminated the effect of $\mathfrak{m}_{n-1}^j$, in the procedure *EvalVc*. In this case, the vector clock upon sending the message $\mathfrak{m}_v$ must be bigger than the vector clock of $\mathfrak{m}_{n-1}^j$. By Lemma 1, $\mathfrak{m}_v$ appears after $\mathfrak{m}_{n-1}^j$ in $\gamma$ and this contradicts the approach that we select the sub-sequence. (2) The effect of $\mathfrak{m}_{n-1}^j$ has not been eliminated by $\mathfrak{m}_v$ but the vector clock of $\mathfrak{m}_{n-1}^j$ is bigger than the vector clock of $\mathfrak{m}_n^k$. However, this is in contradiction with the fact that the message $\mathfrak{m}_{n-1}^j$ appears before $\mathfrak{m}_n^k$ in $\widehat{\gamma}$. (3) The monitor $M_1$ has received a response from the monitor of $\mathfrak{m}_{n-1}^j$, called $M_2$, indicating that the transition of $\mathfrak{m}_{n-1}^j$ has not been enabled. The third case happens when one of the three above mentioned cases again holds in the monitor $M_2$ after receiving the responses for the inquiries about the transition of $\mathfrak{m}_{n-2}^w$ and its vio-transitions. This scenario will be continued to reach the message $\mathfrak{m}_0^i$. The response of the monitor of $\mathfrak{m}_0^i$ (to the monitor of $\mathfrak{m}_1^p$) indicates that the transition of $\mathfrak{m}_0^i$ has not enabled only when the message of $\mathfrak{m}_0^i$ has not been previously sent and this is a contradiction.

- $\gamma \notin L(\mathcal{A})$: The assertion $\gamma \notin L(\mathcal{A})$ holds if and only if there does not exist any sub-sequence like $\widehat{\gamma} \equiv \mathfrak{m}_0^i\mathfrak{m}_1^p \dots \mathfrak{m}_l^h \dots \mathfrak{m}_{n-2}^w\mathfrak{m}_{n-1}^j\mathfrak{m}_n^k$, as we explained in the previous item, in the total order $\gamma$ such that $\widehat{\gamma} \in L(\mathcal{A})$. By running our algorithm, in the procedure *RcvRegularMsg*, the monitor of $\mathfrak{m}_n^k$, called $M_1$, checks the enabled status of its pre-transitions and the vio-transitions of the pre-transitions. So, a transition labeled by $\mathfrak{m}_{n-1}^j$ and its corresponding vio-transition labeled like $\mathfrak{m}_v$ are inquired. By contradiction, we assume that the monitor of $\mathfrak{m}_n^k$ declares the false verdict which denotes that the sub-sequence $\mathfrak{m}_0^i\mathfrak{m}_1^p \dots \mathfrak{m}_{n-2}^w\mathfrak{m}_{n-1}^j\mathfrak{m}_n^k$ certainly has been formed. The monitor $M_1$ certainly declares the false verdict only in one case: It receives information about the messages $\mathfrak{m}_{n-1}^j$ and $\mathfrak{m}_v$, in the procedure *RcvMonitorMsg*, and finds that the message $\mathfrak{m}_v$ has not eliminated the effect of $\mathfrak{m}_{n-1}^j$, i.e., $\mathfrak{m}_v \rightsquigarrow \mathfrak{m}_{n-1}^j$, in the function *NotViolate*. Then, in the procedure *EvalVc*, it finds that the enabled status of a transition labeled by $\mathfrak{m}_{n-1}^j$ is true and $\mathfrak{m}_{n-1}^j \rightsquigarrow \mathfrak{m}_n^k$. The enabled status of the transition labeled by $\mathfrak{m}_{n-1}^j$ becomes true when the monitor of $\mathfrak{m}_{n-1}^j$, called $M_2$, receives information about the enabled status of a transition labeled by $\mathfrak{m}_{n-2}^w$ and its vio-transitions like $\mathfrak{m}_{v'}$ and finds that the message $\mathfrak{m}_{v'}$

has not eliminated the effect of $\mathfrak{m}^w_{n-2}$. Then, $M_2$ finds that the enabled status of a transition labeled by $\mathfrak{m}^w_{n-2}$ is true and $\mathfrak{m}^w_{n-2} \rightsquigarrow \mathfrak{m}^j_{n-1}$. This scenario will be continued to reach the message $\mathfrak{m}^i_0$. The monitor of this message immediately returns the false verdict as it has no pre-transition. Hence, we will have $\mathfrak{m}^i_0 \rightsquigarrow \mathfrak{m}^p_1 \rightsquigarrow \ldots \rightsquigarrow \mathfrak{m}^w_{n-2} \rightsquigarrow \mathfrak{m}^j_{n-1} \rightsquigarrow \mathfrak{m}^k_n$ and consequently by the reverse of Lemma 1, the sub-sequence $\mathfrak{m}^i_0\mathfrak{m}^p_1 \ldots \mathfrak{m}^w_{n-2}\mathfrak{m}^j_{n-1}\mathfrak{m}^k_n$ exists in the total order $\gamma$ and this is a contradiction.

It is noteworthy that in the both cases ($\gamma \in L(\mathcal{A})$, $\gamma \notin L(\mathcal{A})$), the monitor $M_1$ may behave conservatively and declare that the sub-sequence $\mathfrak{m}^i_0 \ldots \mathfrak{m}^k_n$ either may be formed or may not be formed so far, i.e., $\{\top, \bot\}$, which includes the corresponding result of $\gamma$. □

## 6 Evaluation and Experimental Results

To evaluate our choreography-based monitoring algorithm, we formulate four research questions:

- **RQ1**: How scalable is our algorithm for monitoring complex message-based systems, i.e., systems involving many processes in which quite a number of long message sequences as the bad-prefixes of a safety property should not occur?
- **RQ2**: How much overhead is imposed by our monitoring algorithm in terms of communication and memory consumption?
- **RQ3**: How well does our algorithm perform? What is the average latency of the verdict declaration in our algorithm?
- **RQ4**: How conservative is our algorithm when no bad-prefix is formed?

To answer these questions, we should evaluate our monitoring algorithm on message-based systems. There are many programming styles and approaches for implementing such systems. As we assumed in Section 2, processes own message queues to buffer received messages when they are busy with handling another message. Furthermore, each process has a number of message handlers to be invoked upon processing a message from the queue. In the computation model of Actors [5], these assumptions are built-in with the notions of actors for processes and methods for message handlers. Each actor encapsulates its variables and owns a mailbox to buffer messages. There are many actor-based programming languages and frameworks, namely, Akka [21], Elixir [22], Erlang [23], JCoBox [24], etc. Here, we focus on actor-based applications as the sample of message-based systems for evaluating our monitoring algorithm.

To inspect the scalability and the performance of our monitoring algorithm, we need a set of benchmark applications. Finding real-world applications that fall into various categories defined by a number of complexity metrics is nontrivial [25]. Furthermore, finding bad sequences that should not occur in those applications needs the knowledge about the domain of applications. To address this problem, we develop a test case generator. This tool produces actor-based applications with different levels of complexity and a set of message sequences as the bad-prefixes of a property according to the generated application for our experiments. Applications

are generated in terms of a simple actor-based language we designed for our experiments with limited instructions supporting message communication statements. Our statements include assignment of integer or boolean constants to state variables of processes, conditional statements (where their condition are boolean state variables), and sending messages. We also develop a simulator which simulates the execution of the actor-based application and our monitoring algorithm and measures the performance metrics. The simulator tools assume the existence of the network delay and the messages are received after a random delay. Our test case generator tool was developed in Python while our simulator was implemented in Java. The generated actor-based applications were fed into the simulator in JSON format. These tools were executed on a single machine dual core (Intel i5-520M 2.4GHz) with 4 GB memory.

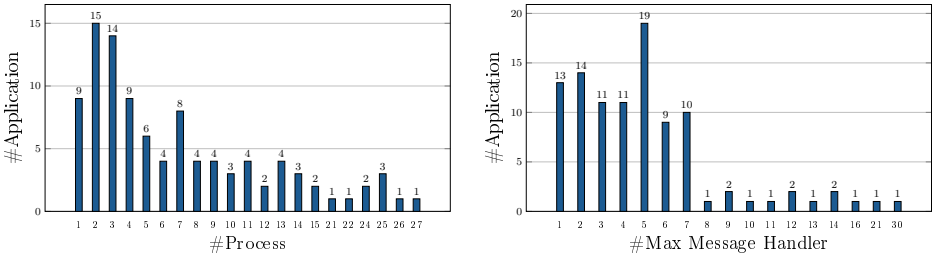
For answering RQ1, we measure the impact of the different levels of complexity of applications and the different number of message sequences with varying lengths on the scalability of our algorithm. For answering RQ2, RQ3, and RQ4, we use the synthesized applications with the same level of complexity and measure the conservative rate and the overhead costs in terms of the monitoring messages, the average memory consumption of the monitors, and the average latency of the verdict declaration.

### 6.1 Benchmark Applications and Properties

We aim to generate the benchmark applications with different categories defined by a number of complexity metrics. First, we performed an empirical study involving real world applications written in actor-based language Elixir [22] and analyzed approximately 100 applications chosen randomly from Github [26]. The selected applications were in various categories such as education, communication, etc. We analyzed these applications according to the two complexity metrics that may affect our monitoring algorithm: (1) the number of processes in the application, (2) the maximum number of message handlers per process. Depending on the property, the first metric may affect the number of monitors that involve in our algorithm while the second may have an impact on the number of messages that our algorithm inquires about. Fig. 6 shows the distribution of these complexity metrics among the 100 real world Elixir-based applications from Github.

Following the approach of [25], we define *complexity classes* for generating subject applications with varying values of these two metrics in our experiments. To this aim, we aggregate the results of the two distributions and produce the overall application complexity classes ranging from 10th to 90th percentile, shown in Table 4. For instance, the 10th overall complexity in Table 4 corresponds to the 10th percentile in the two distributions shown in Fig. 6. This means that an application belonging to a lower class is less complex with respect to the two metrics compared to an application from a higher complexity class.

Our test case generator tool produces an actor-based application based on the two parameters: the complexity class and *the maximum message communication chain* between the processes. The latter parameter defines the maximum number of processes in a chain of message handlers that send messages to each other. Based on the synthesized application, the tool constructs different message sequences, specified by sequence automata. The required parameters for generating these




(a) (b)

Fig. 6: Complexity metrics distribution from a random sample of 100 Elixir-based applications

Table 4: The overall complexity classes of applications

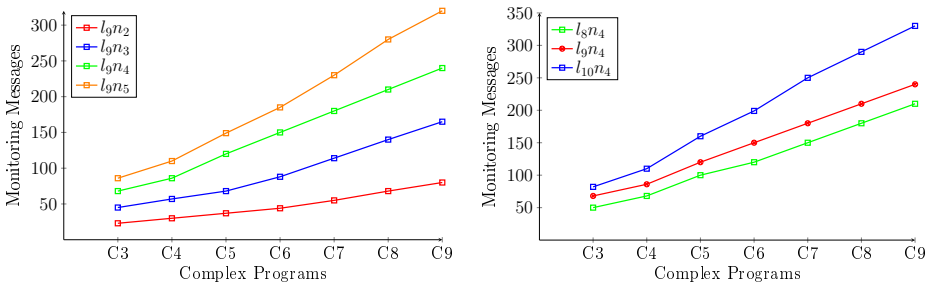
| Complexity Class | #Process | #Message Handler |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 2 | 2 |
| 3 | 3 | 3 |
| 4 | 4 | 4 |
| 5 | 5 | 5 |
| 6 | 7 | 5 |
| 7 | 9 | 6 |
| 8 | 12 | 7 |
| 9 | 14 | 9 |

sequences are the total number of message sequences and the minimum length of the sequences i.e., the length of the path over forward transitions. As our algorithm is based on the decentralized specification, the resulting sequence automaton is decomposed into some transition tables as explained in Section 4.1.

### 6.2 Experiment Results

To evaluate our monitoring algorithm, we conduct each experiment by adjusting the following experimental parameters: (1) applications from different complexity classes, and (2) sets of message sequences as the bad-prefixes of a safety property.

To answer **RQ1**, we generate different synthesized applications with different complexities. To synthesize application for the complicity class $C_i$, we adjust the parameter of the maximum message communication chain to $i$; this means that $i$ processes send messages to each other in a chain. The tool also generates message sequences based on the maximum message communication chain for each synthesized application. In cases that the intended length of the sequences is more than the maximum message communication chain of an application, the tool constructs the sequence based on the messages in the chain and some random messages with no causal relationship with others. Here, we adjust the length of the sequences to nine and the number of sequences varies from two to five sequences. The reason for choosing the length of nine is that the maximum message communication chain for the most complex class, i.e., $C_9$, equals nine. To evaluate the scalability of our algorithm, we measure the overhead cost of communicating monitoring messages.




(a) Results for different number of sequences of fixed lengths

(b) Results for a fixed number of sequences of various lengths

Fig. 7: The monitoring messages overhead for different complex applications and the set of message sequences $l_k n_j$ where $j$ denotes the number of message sequences and $k$ denotes their lengths

Our algorithm is scalable if the number of communicated monitoring messages grows linearly by increasing the complexity of applications. We measure the average number of communicated monitoring messages among all monitors by running each experiment ten times. The average number of monitoring messages for each synthesized application of complexity class $C_3$ to $C_9$ is shown in Fig. 7. When either the length or the number of sequences increases, the number of messages that monitors should inquiry about them will also increase. When the number and length of sequences are fixed, it is expected that the complexity of applications has no effect on the number of monitoring messages in Fig. 7. Now, we explain the reason for the smooth increase in the number of messages. The number of times that a monitor inquires others because of an specific message $\mathfrak{m}$ (occurring in bad sequences) depends on the number of times that the message $\mathfrak{m}$ has occurred at runtime. If $\mathfrak{m}$ occurs once, then the corresponding monitor inquires others for once and so the number of monitoring messages for different complex applications according to a message sequence will be the same. Otherwise, if the message occurs more than once, then its monitor inquiries others more than one and so the monitoring messages are increased. To make our experiments fair, we enforce the restriction that each message can appear in at most two communication chains. In this case, each constituent message of the sequence can occur at most two times. Concluding that as the length of message communication chain ($i$ for each class $C_i$) increase, the number of monitoring messages grows for complex applications.

To answer **RQ2** and **RQ3**, we use the synthesized application from the complexity class $C_6$ and consider three message sequences whose lengths vary from two to six. Then, we evaluate the number of monitoring messages, the average memory consumption of the monitors, and the average latency of the verdict declaration. As illustrated in Fig. 8a and Fig. 8b, the number of monitoring messages and the memory consumption of the monitors grows linearly as the length of sequences increases. Instead of measuring the average time for the verdict declaration, we measure the average latency that the monitors of the applications need to receives responses from other monitors and decide to whether add a record to their history. It is shown in Fig. 8c that as the length of the sequence increases, monitors need

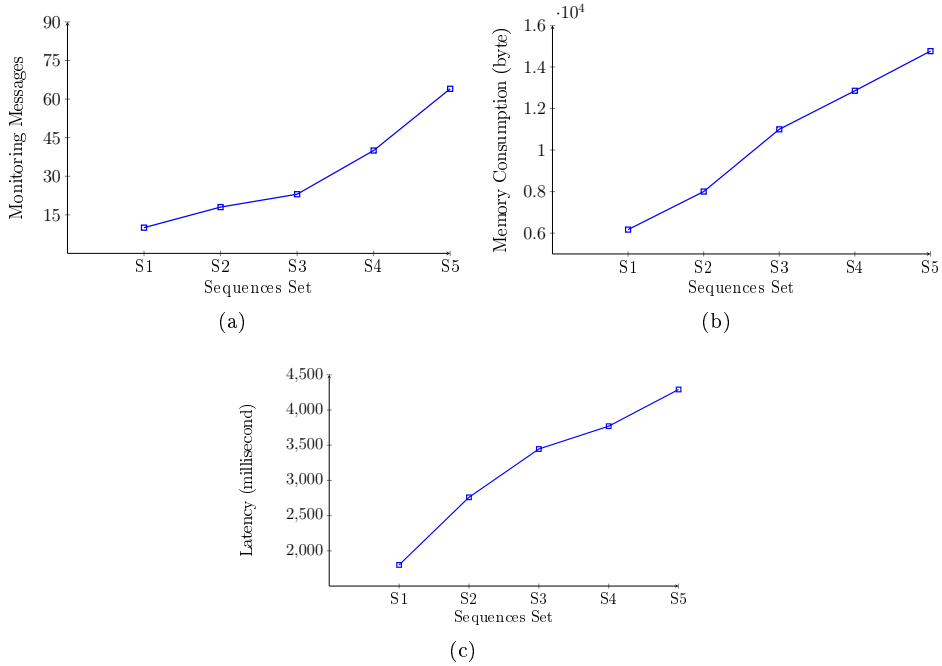



Fig. 8: (a) The average number of monitoring messages, (b) the average memory consumption, and (c) the average latency of the verdict declaration, for synthesized applications of the complexity class $C_6$ and the set of message sequences $S_i = l_{i+1}n_3$ where $S_i$ denotes three message sequences whose lengths are $i+1$

more collaboration, hence, more time is needed to gather all responses from other monitors.

To answer **RQ4**, we use the synthesized application from the complexity class $C_6$ and consider three message sequences whose lengths are seven. We run this experiment one hundred times and measure the times that the algorithm declares either no result, i.e., the bad-prefixes has not been formed, or $\{\top, \bot\}$ result, i.e., the sequence may be formed or not formed so far. In this experiment, our algorithm declares the non-formation of the sequences in 76% of executions. This percentile depends on the network delay value and the order that messages are executed.

## 7 Related Work

In this paper, we focus on the decentralized runtime verification of decentralized specifications based on the messages sequences in a message-based systems. There are various runtime verification techniques that use different specification languages and make different assumptions on the monitored system [39, 38, 43, 42, 41, 40, 44, 45]. In this section, we address those works that either their specifications or their assumptions are close to us.

Runtime verification of distributed component-based systems with a central monitor is discussed in [44, 45]. All distributed components send a copy of their

events to the central monitor to verify the property. However, using a central monitor in a distributed system is undesirable because it leads to more communication, more network overhead and more risks of failure. A decentralized monitoring algorithm is introduced in [37] for runtime verification of regular properties in distributed systems that its components share a global clock. The local monitors communicate with each other to emit the global verdict while their messages are timestamped by the *global clock*. The decentralized monitoring of LTL formulas has been studied in [29,30,34] using a global clock in the system. In [29], it is shown how orchestration, choreography, and migration can be applied to LTL monitors. A choreography-based runtime verification approach, based on pushing strategy, with the three-valued semantics [46] is proposed in [30] for synchronous distributed systems. All these algorithms are applicable for either synchronous distributed systems or asynchronous distributed systems with the assumption of a global clock.

Runtime verification of properties based on messages sequences also have been considered in [32,31]. In [32] each message has a unique timestamp and the total ordering among messages can be determined by these unique timestamps. In [31], systems implemented by Erlang [23] are verified by exploiting the tracing mechanism of the Erlang framework. This mechanism returns a unique trace instead of the set of partitioned traces. As we do not have a global clock, the total ordering among messages is not determined at runtime.

Among the works involved in the *decentralized* monitoring of message-based systems, the verification of past-linear temporal logic has been addressed in [38]. In this research, monitors gain knowledge about the state of the system by piggybacking on the existing communication among processes. If a process rarely communicates, then some violation of properties cannot be detected and so their approach is not sound. The approach of [38] is extended in [12] by using the distributed temporal logic and specifying some liveness properties. Similar to the previous work, monitoring information are exchanged via communicated messages and so the soundness of their work depends to the communication. In [35,36], it is assumed that messages either can be received out-of-order or they may be failed. In these works, each process owns a local clock to timestamp each event. However, they make a strong assumption that local clocks are accurate and the order of events that are timestamped by the local clocks complies with the total order timestamped by a global physical clock. They emphasize that in a system where local clocks are not accurate, local clocks should be synchronized. Decentralized runtime verification of dynamic properties which are created or evolved during runtime, is considered in [28]. The authors only use a simple timestamp instead of a vector clock and so their approach is not complete. In [17], a decentralized algorithm for runtime verification of LTL properties as a centralized specification is given. In this work, each local monitor has a local copy of a central specification that specifies by an automaton. The algorithm is based on the online predicate detection algorithm [47] with the difference of evaluating multi predicate at runtime. A local monitor may collaborate with others monitor to evaluate a predicate on the transitions of the given automaton. Then, the monitor moves on the automaton and tries to evaluate the next predicate. The evaluation of the predicates are independent of each other and the moments that different predicates are evaluated is not important. This work only considers the conjunction and disjunction predicates and do not deal with the linked predicates. The linked predicates specify

sequences of events that can be ordered by the happened-before relation [9]. The proposed solution in [9] detects a linked predicate at runtime by inserting additional information about events in the communication messages. This work has been extended in [48] by considering a negation operator in the predicate. However, they assume that if an event with a negation operator in a given predicate occurs, then the whole predicate is violated. None of these works has considered the situation that an event eliminates only the part of the formed sequence. Furthermore, they assume that there is a happened-before relation between all events of a given predicate. Here, we aim to detect the message sequences in a way that there is no pre-assumption on the occurrence orders of such messages at runtime.

## 8 Conclusion and Future work

In this paper, we addressed the choreography-based runtime detection of the message sequences in message-based systems in which distributed processes communicate via asynchronous message passing. We have assumed that there is no global clock and the network may postpone delivery of messages. Our proposed algorithm is fully decentralized in a sense that each process is equipped with a monitor and each monitor does not access the whole message sequences. By using the vector clock, monitors can not identify the total ordering of messages and hence, may announce the result of the sequence formation conservatively. To minimize the conservative result, monitors pull information from their counterparts and update the shared vector clock. The former helps to detect the non-formation of the message sequences certainly and the latter is suitable when the processes rarely communicate with each other. Our experimental results indicate that by using the pulling strategy, our algorithm declares no result, i.e., the bad-prefixes has not been formed, in more than 70% of executions depending on the network delay. We developed an actor-based benchmark suite to evaluate the effect of the complexity of apps or the length of message-sequences on the number of monitoring messages, memory consumption of monitors, and the latency detection. Our experiment results show that our algorithm is scalable: with the increase of the complexity of apps or the length of message sequences, the number of monitoring messages, memory consumption, and the latency grows linearly.

We aim to reduce the size of conservative results by using the idea of matrix clocks [49]. We can also extend the specification formalism in a way that the sequence of messages can eliminate the effects of one (some) message(s).

## References

1. Kolchinsky. I, Schuster. A, Efficient adaptive detection of complex event patterns, In Journal Proceedings of the VLDB Endowment, Vol.11, Issue.11, pp.46-59, ACM, 2018.
2. Qi. Y, Cao. L, Ray. M, Rundensteiner. E, Complex event analytics: online aggregation of stream sequence patterns, In Proceedings Conference on Management of Data, ACM, 2014.
3. Palanisamy. S, Dürr. F, Adnan. M, Rothermel. K, Preserving Privacy and Quality of Service in Complex Event Processing through Event Reordering, In Proceedings Conference on Distributed and Event-based Systems, ACM, 2018.
4. Wang. D, Rundensteiner. E, Wang. H, Ellison. R, Active complex event processing: applications in real-time health care, In Journal Proceedings of the VLDB Endowment, Vol.3, Issue.2, pp.45-48, ACM, 2010.

5. Agha. G, ACTORS - a model of concurrent computation in distributed systems, In MIT Press series in artificial intelligence, 1990.
6. Koster. J, Cutsem. T, Meuter. W, 43 years of actors: a taxonomy of actor models and their key properties, In Proceeding Workshop on Programming Based on Actors, Agents, and Decentralized Control, ACM, 2016.
7. Lopez. C, Marr. S, Gonzalez. E, Mössenböck. H, A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs, Programming with Actors, In Programming with Actors, Lecture Notes in Computer Science, pp.155-185, Springer, 2018.
8. Long. Y, Bagherzadeh. M, Lin. E, Upadhyaya. G, Rajan. H, On Ordering Problems in Message Passing Software, In Proceedings Conference on Modularity, ACM, 2016.
9. Miller. B, Choi. J, Breakpoints and halting in distributed programs, In Conference on Distributed Computing Systems, IEEE, 1988.
10. Leucker. M, Schallhart. C, A brief account of runtime verification, In Journal of Logic and Algebraic Programming, Vol.78, Issue.5, pp.293-303, ScienceDirect, 2009.
11. Fraigniaud. P, Rajsbaum. S, Travers. C, On the Number of Opinions Needed for Fault-Tolerant Run-Time Monitoring in Distributed Systems, In Conference on Runtime verification, pp. 92-107, Springer, 2014.
12. Scheel. T, Schmitz. M, Three-valued asynchronous distributed runtime verification, In Proceedings Conference on Formal Methods and Models for Code design, pp. 52-61. IEEE, 2014.
13. Francalanza. A, Gauci. A, Pace. G, Distributed system contract monitoring, In Journal of Logic and Algebraic Programming, Vol.82, Issues.5-7, pp.186–215, ScienceDirect, 2013.
14. El-Hokayem. A, Falcone. Y, On the Monitoring of Decentralized Specifications: Semantics, Properties, Analysis and Simulation, In Transactions on Software Engineering and Methodology, Vol.29, Issue.1, pp.1-57, ACM, 2019.
15. Sanches. C, Shneider. G, Ahrendt. W, et al, A Survey of Challenges for Runtime Verification from Advanced Application Domains (Beyond Software), In Journal of Formal Methods in System Design, Vol.54, Issue.3, pp.273-335, Springer, 2018.
16. Mattern. F, Virtual Time and Global States of Distributed Systems, Parallel and Distributed Algorithms, In North-Holland Press, 1988.
17. Mostafa. M, Bonakdarpour. B, Decentralized Runtime Verification of LTL Specifications in Distributed Systems, In Conference on Parallel and Distributed Processing Symposium, IEEE, 2015.
18. Baier. C, Katoen. J, Principles of Model Checking, In MIT Press series, 2008.
19. Lamport. L, Time, clocks, and the ordering of events in a distributed system, In Communications of the ACM, Vol.21, Issue.7, pp.558-565, ACM, 1978.
20. Fifge. C, Timestamps in message-passing systems that preserve partial ordering, In Proceedings Conference on Computer Science, 1988.
21. Akka: http://akka.io/
22. Elixir: https://elixir-lang.org/
23. Armstrong. J, Virding. R, Williams. M, Concurrent Programming in Erlang, In Prentice-Hall Press, 1993.
24. Schafer.J, Poetzsch. A, JCoBox: Generalizing active objects to concurrent components, In Conference on Object-Oriented Programming, Springer, 2010.
25. Mirzaei. N, Bagheri. H, Mahmood. R, Malek. S, Sig-Droid:Automated system input generation for Android applications, In Symposium on Software Reliability Engineering, IEEE, 2015.
26. Github: https://github.com
27. Kshemkalyani. A, Singhal. M, Distributed Computing: Principles, Algorithms, and Systems, In Cambridge University Press, 2011.
28. Francalanza. A, Gauci. A, Pace. G, Distributed system contract monitoring, In Journal of Logic and Algebraic Programming, Vol.82, Issues.5-7, pp.186-215, ScienceDirect, 2013.
29. Colombo. C, Falcone. Y, Organising LTL monitors over distributed systems with a global clock, In Conference on Runtime Verification, Springer, 2014.
30. Colombo. C, Falcone. Y, Organising LTL monitors over distributed systems with a global clock, In Journal of Formal Methods on System Design, Vol.42, Issues.1-2, pp.109-158, Springer, 2016.
31. Francalanza. A, Seychell. A, Synthesising correct concurrent runtime monitors, In Journal of Formal Methods in System Design volume, Vol.46, Issue.3, pp.226-261, Springer, 2015.
32. Shafiei. N, Tkachuk. O, Mehlitz. P, MESA: Message-Based System Analysis Using Runtime Verification, In ntrs.nasa.gov publication, 2017.

33. Downey. A, The little book of semaphore, In Green tea Press, 2009.
34. Bauer. A, Falcone. Y, Decentralized LTL monitoring, In Symposium on Formal Methods, Springer, 2012.
35. Basin. D, Klaedtke. F, Zalinesu. E, Failure-aware runtime verification of distributed systems, In Proceedings Conference on Foundation of Software Technology and Theoretical Computed science, Schloss Dagstuhl, 2015.
36. Basin. D, Klaedtke. F, Zalinesu. E, Runtime Verification of Temporal Properties over Out-of-Order Data Streams, In Journal on Computer Aided Verification, Vol. abs/1707.05555, pp.56-76, 2017.
37. Falcone. Y, Cornebize. T, Fernandez. J, Efficient and generalized decentralized monitoring of regular languages, In Proceedings Conference on Formal Techniques for Distributed Objects, Components, and Systems, Springer 2014.
38. Sen. K, Vardhan. A, Agha. G, Rosu. G, Efficient decentralized monitoring of safety in distributed systems, In Proceedings Conference on Software Engineering, IEEE, 2004.
39. Francalanza. A, Pérez. J, Sánchez. C, Runtime Verification for Decentralised and Distributed Systems, Lectures on Runtime Verification, pp 176-210, Springer, 2018.
40. Barringer. H, Goldberg. A„ Havelund. K, Sen. K, Rule-based runtime verification. In Proceedings Conference on Verification, Model Checking and Abstract Interpretation, Springer, 2004.
41. Basin. D, Klaedtke. F, Muller. S, Monitoring metric first-order temporal properties, In Journal of the ACM, Vol.62, Issue.2, ACM, 2015.
42. Bauer. A, Leucker. M, Schallhart. C, Runtime verification for LTL and TLTL, In Transaction of Software Engineering Methodology, Vol.20, Issue.4, ACM, 2011.
43. Meredith. P, Jin. D, Griffith. D, Chen. F, Rosu. G, An overview of the MOP runtime verification framework, In Journal of Software Tools Technology, Vol.14, Issue.3, pp.249–289, Springer, 2012.
44. Nazarpour. H, Falcone. Y, Jaber. M, Bensalem. S, Bozga. M, Monitoring Distributed Component-Based Systems, In CoRR, Vol. abs/1707.05555, 2017.
45. Falcone. Y, Jaber. M, Nguyen. T, Bozga. M, Bensalem. S, Runtime verification of component-based systems, In Conference on Software Engineering and Formal Methods, Springer, 2011.
46. Bauer. A, Leucker. M, Schallhart. C, Comparing LTL semantics for runtime verification, In Journal on Logic Computing, Vol.20, Issue.3, pp.651–674, 2010.
47. Chauhan. H, Garg. V, Natarajan. A, Mittal. N, A Distributed Abstraction Algorithm for Online Predicate Detection, In Symposium on Reliable Distributed Systems, IEEE, 2013.
48. Hurfin. M, Plouzeau. N, Raynal. M, Detecting atomic sequences of predicates in distributed computations. In Proceedings workshop on Parallel and distributed debugging, ACM, 1993.
49. Drummond. L, Barbosab. V, On reducing the complexity of matrix clocks, In Journal of Parallel Computing, Vol. 29, Issue 7, pp. 895-905, ACM, 2003.