Decentralized Runtime Enforcement of Message Sequences in Message-Based Systems

Mahboubeh Samadi
University of Tehran, Iran
mbh.samadi@ut.ac.ir

Fatemeh Ghassemi
University of Tehran, Iran
fghassemi@ut.ac.ir

Ramtin Khosravi
University of Tehran, Iran
r.khosravi@ut.ac.ir

Abstract

In the new generation of message-based systems such as network-based smart systems, distributed components collaborate via asynchronous message passing. In some cases, particular ordering among the messages may lead to violation of the desired properties such as data confidentiality. Due to the absence of a global clock and usage of off-the-shelf components, there is no control over the order of messages at design time. To make such systems safe, we propose a choreography-based runtime enforcement algorithm that given an automata-based specification of unwanted message sequences, prevents certain messages to be sent, and assures that the unwanted sequences are not formed. Our algorithm is fully decentralized in the sense that each component is equipped with a monitor, as opposed to having a centralized monitor. As there is no global clock in message-based systems, the order of messages cannot be determined exactly. In this way, the monitors behave conservatively in the sense that they prevent a message from being sent, even when the sequence may not be formed. We aim to minimize conservative prevention in our algorithm when the message sequence has not been formed. The efficiency and scalability of our algorithm are evaluated in terms of the communication overhead and the blocking duration through simulation.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies

Keywords and phrases Asynchronous Message Passing, Choreography-Based, Runtime Enforcement, Runtime Prevention, Message Ordering

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.21

1 Introduction

The new generation of message-based systems such as network-based smart applications are usually distributed and may consist of off-the-shelf components developed by different vendors. These systems are maintainable and scalable as components collaborate via asynchronous message passing.

Such systems must satisfy the required properties such as data confidentiality, safety, robustness, and security. However, a sequence of messages may lead to the property violation. As an example (inspired by [19]), assume a building that consists of different locations named A-E where the location E is restricted and a visitor must enter the restricted location through a legal path (Figure 1). The only legal path to the restricted location is through the consecutive locations A, C, and then E. Each location is equipped with a smart security camera and a smart door that the visitor must use a smart door to enter the location. The path between different locations is such that if the consecutive locations B and D are visited, then the visitor will return to the location A. If a visitor is entered the restricted location by passing through the consecutive locations A, B, and then E, it can be inferred that

the visitor accesses the restricted location illegally. This illegal access violates the security rules of the building and can be detected by the message sequence *Open*(A,v) (the smart door of the location A has opened by the visitor v), *Open*(B,v) and *Open*(E,v). There are other examples of sequence-based patterns in the *Complex Event Processing* domain [28, 30]. Furthermore, a *message protocol violation* bug [24] and *linked predicates* [26] are also related to the certain order of communicated messages. The former occurs in an actor-based program [1], as a sample of a message-based program, when the components exchange messages that are not consistent with the intended protocol of the application. The latter defines properties (predicates) on a sequence of events interpreting events to messages.

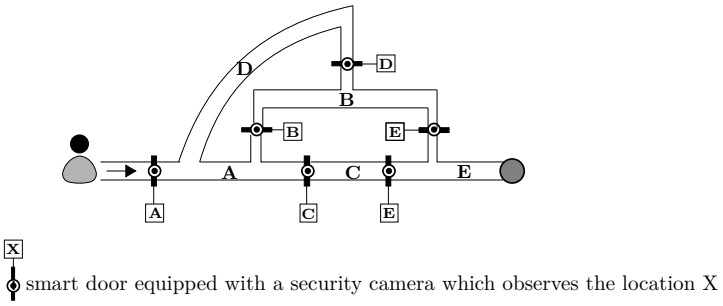

Figure 1 The locations of a building denoted by A, B, C, D, and E separated by smart doors equipped with security cameras.

As most systems in practice are an integration of various components which may be closed-source and proprietary, the message sequences cannot be inspected statically at design time to guarantee that unwanted sequences never happen at runtime. Runtime enforcement can be used as a verification technique that makes sure such systems satisfy the given properties and correct the execution of the system [23]. In this paper, we focus on the *decentralized* runtime enforcement of properties where each component is equipped with a local monitor. These *decentralized monitors* communicate with each other to prevent the violation of the given property. The given property is violated by the formation of messages sequences, where the sequences obey a specific pattern and specify the 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. In the previous example, the message sequence *Open*(A,v), *Open*(B,v), *Open*(D,v), *Open*(C,v), and *Open*(E,v) does not violate the security rule as the visitor returns to the previous location A by passing through the consecutive locations B and D. Finally, the visitor enters the restricted location E by passing through the location C.

Our decentralized runtime enforcement approach (Sect. 3) uses the *choreography* setting [9], where local monitors are organized into a network and collaborate with each other by using a specific protocol. This setting deals with *decentralized specifications* in which each local monitor has access to some parts of the message sequences. The decentralized runtime prevention of messages sequences formation in a message-based system is challenging due to the absence of a global clock and asynchronous message passing. 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 [34]. With the asynchronous message passing, a component is not synchronized with other components and so it has no information about the status of a sequence formation. In the proposed algorithm (Sect. 4), we will use vector clocks [25] in our messages to detect the *partial* ordering among messages, and then prevent the sequence formation. When monitors cannot detect the *total* order among messages, they

may prevent the sequence formation conservatively in the sense that they prevent a certain message from being sent even if that message does not lead to a sequence formation. We aim to minimize the conservative prevention in our algorithm when the message sequences have not been formed. To prevent a sequence formation, a component may be blocked before sending its message until its monitor makes sure about the effect of that message on the sequence formation. We also aim to prevent the sequence formation by minimizing the number of blocked components and manipulation of messages ordering. To the best of our knowledge, there is no decentralized runtime enforcement of sequence-based properties in message-based systems. We evaluate the performance of our algorithm and 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 and the blocking duration of processes grow linearly (Sect. 5).

2 Background

2.1 Message-Based Systems

We define a *Message-Based System* $D = \{P_1, \ldots, P_n\}$ as a set of n processes that communicate via asynchronous message passing and guarantees in-order delivery, i.e., two messages 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 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 and Msg be the set of messages communicated among processes ranged over by $\mathfrak{m}$. Each message $\mathfrak{m} \in Msg$ 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 and state variables where a message handler specifies how the received message must be responded to. The computation of the process P_x can be abstracted in terms of *event*s which are categorized into *internal*, *send*, and *take* events, where an *internal* event changes the state variables of P_x, the event $send(P_x, m, P_y)$ occurs when P_x sends m to P_y where $m \in MName$, and the event $take(P_y, m, P_x)$ occurs when P_x takes m ,which is sent by P_y, from its queue.

Events in the message-based system can be partially ordered according to the happened-before relation [21] which is implemented by the vector clock. Let a message $\mathfrak{m}_i \in Msg$ be a triple of (P_x, m_i, P_y). A happened-before relation $\rightsquigarrow$ defines a causal order among events: (1) within a single message handler, the ordering of events is defined as their execution order which can be determined unambiguously, (2) $send(P_x, m, P_y) \rightsquigarrow take(P_x, m, P_y)$, and (3) for events e_a, e_b, e_c, if $e_a \rightsquigarrow e_b$ and $e_b \rightsquigarrow e_c$ then $e_a \rightsquigarrow e_c$.

Two events e_a and e_b are concurrent and denoted by $e_a \parallel e_b$ if there is no happened-before relation between them.

2.2 Message-Based Property Specification

We aim to prevent certain unwanted sequences of send/take events from being formed. For an unwanted sequence, the occurrence of some events contributes to the formation of the sequence, while some other events may cancel the effect of the previous ones. To formalize our message sequences, we use the *sequence automaton* defined in [32] as an extension to

nondeterministic finite automata. In this model, transitions are partitioned into two sets of *forward* and *backward* transitions. Forward transitions, denoted by $\rightarrow$, lead to the sequences formation while backward transitions, denoted by $\dashrightarrow$, cancel the formation of sequences. Let $\rightarrow^*$ be the transitive closure of the $\rightarrow$ relation.

▶ **Definition 1** (Sequence Automaton [32]). *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 the transitions specified by δ_f (resp. δ_b) are forward (resp. backward) transitions, i.e.,*

- *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 $\forall\, i < n, q_i \not\dashrightarrow q_{i+1}$.*
- *$q_i \dashrightarrow q_j \Rightarrow q_i \neq q_j \wedge q_j \rightarrow^* q_i$.*

To simplify the explanation, we restrict Σ to *send* events and show $send(\mathfrak{m})$ by $\mathfrak{m}$ in our graphical representation of sequence automata.

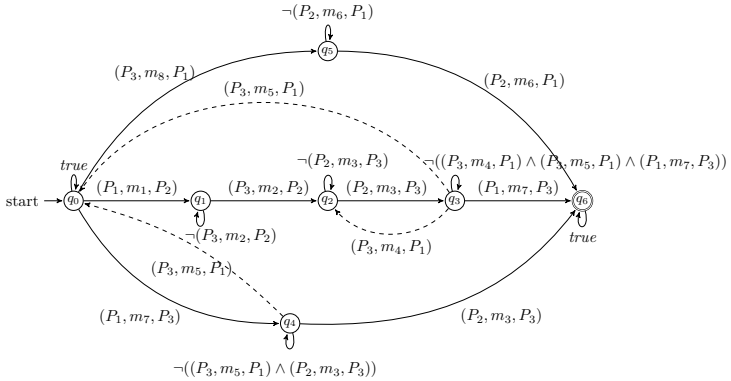

Figure 2 The sequence automaton $\mathcal{A}_1$ where the solid edges denote the forward transitions and the dashed edges denote the backward transitions.

The sequence automaton $\mathcal{A}_1$, in Figure 2, represents the sequences of *send* events. For instance, this automaton describes that if first the message (P_1, m_7, P_3) is sent and then the message (P_2, m_3, P_3) is sent while the message (P_3, m_5, P_1) is not sent after (P_1, m_7, P_3) and before (P_2, m_3, P_3), then the sequence $(P_1, m_7, P_3)(P_2, m_3, P_3)$ is formed. If the sequence $(P_1, m_7, P_3)(P_3, m_5, P_1)(P_2, m_3, P_3)$ is observed, the occurrence of (P_3, m_5, P_1) has eliminated the effect of the occurrence of (P_1, m_7, P_3) and so, the occurrence of (P_2, m_3, P_3) will not form a sequence (as the reaching state q_0 is not a final state). However, a sequence is formed by the occurrence of $(P_1, m_7, P_3)(P_3, m_5, P_1)(P_1, m_7, P_3)(P_2, m_3, P_3)$. The self-loop over the state q_4 expresses that between the occurrences of (P_1, m_7, P_3) and (P_2, m_3, P_3), any message except (P_3, m_5, P_1) and (P_2, m_3, P_3) can be sent.

When a message $\mathfrak{m}$ occurs, a transition like $(q, \mathfrak{m}, q')$ may lead to the formation of a sequence from the initial state up to q'. To form such a sequence, it is necessary that at least a message over one of the preceding transition of $(q, \mathfrak{m}, q')$, like t, has occurred and no message over the backward transitions has eliminated the effect of t. The pre-transitions of $(q, \mathfrak{m}, q')$ is the set of preceding transitions whose labeled messages can occur before $\mathfrak{m}$ in a sequence. The preceding transitions have the same destination as the source state of $(q, \mathfrak{m}, q')$, i.e., q.

▶ **Definition 2** (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:

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

A backward transition $(q, \mathfrak{m}, q')$ can eliminate the effect of all forward transitions on a path from q' to q, when $\mathfrak{m}$ occurs after the occurrence of labeled messages over the sequence of forward transitions on a path from q' to q.

▶ **Definition 3** (vio-transition [32]). *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 Figure 2, $preTrns(\mathcal{A}_1, (q_0, \mathfrak{m}_1, q_1)) = \emptyset$, $preTrns(\mathcal{A}_1, (q_4, \mathfrak{m}_3, q_6))$ and $preTrns(\mathcal{A}_1, (q_4, \mathfrak{m}_5, q_0))$ are equal to $(q_0, \mathfrak{m}_7, q_4)$. Furthermore, the transition $(q_4, \mathfrak{m}_5, q_0)$ violates the effect of $(q_0, \mathfrak{m}_7, q_4)$ and so $vioTrns(\mathcal{A}_1, (q_0, \mathfrak{m}_7, q_4)) = \{(q_4, \mathfrak{m}_5, q_0)\}$.

3 Choreography-Based Runtime Enforcement Approach

We aim to prevent the formation of unwanted message sequences that are specified by a sequence automaton in a message-based system at runtime. A message sequence $\mathfrak{m}_1 \dots \mathfrak{m}_x \dots \mathfrak{m}_n$ is formed if we move from the initial state by the message $\mathfrak{m}_1$ and reach a final state by the message $\mathfrak{m}_n$. To avoid the sequence formation, we equip each process P_x with a *monitor* M_x. The local monitors of the processes are organized as a network and communicate with each other to prevent the sequence formation.

To prevent the formation of $\mathfrak{m}_1 \dots \mathfrak{m}_{x-1}\mathfrak{m}_x \dots \mathfrak{m}_n$, the process P_n as the sender of $\mathfrak{m}_n$ must make sure that $\mathfrak{m}_1 \dots \mathfrak{m}_{n-1}$ has not been formed before sending $\mathfrak{m}_n$. One possible solution is that its monitor, i.e., M_n, communicates with other monitors and asks if all of the messages $\mathfrak{m}_1$ to $\mathfrak{m}_{n-1}$ have been sent. However, this solution imposes a high overhead on the system as there may be many sequences that lead to $\mathfrak{m}_n$, and so M_n must ask other monitors about the sending status of many messages. So, we propose a choreography-based prevention approach where monitors have access to some parts of the sequence, detect the sequence formation incrementally, and finally the monitor M_n informs its process to either send the message $\mathfrak{m}_n$ safely or send an error message. To this end, upon sending a message $\mathfrak{m}_x$, the sender P_x informs its monitor M_x, which in turn asks M_{x-1} about the formation of $\mathfrak{m}_1 \dots \mathfrak{m}_{x-1}$. The sequence $\mathfrak{m}_1 \dots \mathfrak{m}_{x-1}\mathfrak{m}_x$ will be formed if $\mathfrak{m}_1 \dots \mathfrak{m}_{x-1}$ is formed and $\mathfrak{m}_{x-1} \rightsquigarrow \mathfrak{m}_x$. This way, the communication overhead between the monitors is distributed over time, instead of happening all at the final states.

In the following, first, we explain how monitors have access to some parts of the specification. Then, we demonstrate how monitors must communicate with each other to prevent the formation of unwanted message sequences.

3.1 Choreography-Based Property Specification

In this section, we use the choreography-based specification in [32] where each monitor has its own local property. As we explain in Section 2.2, the message sequences can be specified

by a sequence automaton. To specify the choreography-based specification, the sequence automaton should be broken down into a set of transition tables. Each monitor M_x maintains a transition table that contains the transitions labeled by the messages that their sender is P_x. 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.

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

transition	final	pre-transition	vio-transition
$(q_0, (P_1, m_1, P_2), q_1)$	$\bot$	$\emptyset$	$\emptyset$
$(q_0, (P_1.m_7.P_3), q_4)$	$\bot$	$\emptyset$	$\emptyset$
$(q_3, (P_1, m_7, P_3), q_6)$	$\top$	$(q_2, @\, P_2, q_3)$	$\{(q_3, @\, P_3, q_2),$ $(q_3, @\, P_3, q_0)\}$

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

transition	final	pre-transition	vio-transition
$(q_2, (P_2, m_3, P_3), q_3)$	$\bot$	$(q_1, @\, P_3, q_2)$	$\{(q_3, @\, P_3, q_0)\}$
$(q_4, (P_2, m_3, P_3), q_6)$	$\top$	$(q_0, @\, P_1, q_4)$	$\{(q_4, @\, P_3, q_0)\}$
$(q_5, (P_2, m_6, P_1), q_6)$	$\top$	$(q_0, @\, P_3, q_5)$	$\emptyset$

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

transition	final	pre-transition	vio-transition
$(q_1, (P_3, m_2, P_2), q_2)$	$\bot$	$(q_0, @\, P_1, q_1)$	$\{(q_3, @P_3, q_0)\}$
$(q_3, (P_3, m_4, P_1), q_2)$	$\bot$	$(q_2, @\, P_2, q_3)$	$\emptyset$
$(q_0, (P_3, m_8, P_1), q_5)$	$\bot$	$\emptyset$	$\emptyset$
$(q_4, (P_3, m_5, P_1), q_0)$	$\bot$	$(q_0, @\, P_1, q_4)$	$\emptyset$
$(q_3, (P_3, m_5, P_1), q_0)$	$\bot$	$(q_2, @\, P_2, q_3)$	$\emptyset$

For instance, the automaton in Figure 2 is decomposed into three tables shown in Table 1, 2, and 3. Table 1 is maintained by the monitor of P_1 and contains information of the transitions which the sender of the labeled messages is P_1. The transition $(q_0, (P_1, m_1, P_2), q_1)$ does not lead to a final state and it has no pre-transition and so no corresponding vio-transition. So, the first row $((q_0, (P_1, m_1, P_2), q_1), \bot, \emptyset, \emptyset)$ is included in $\mathcal{T}_{P_1}$. The transition $(q_3, (P_1, m_7, P_3), q_6)$ leads to the final state and has only one pre-transition as $(q_2, (P_2, m_3, P_3), q_3)$ and two corresponding vio-transitions of $(q_3, (P_3, m_5, P_1), q_0)$ and $(q_3, (P_3, m_4, P_1), q_2)$. The corresponding row of the transition $(q_3, (P_1, m_7, P_3), q_6)$ in $\mathcal{T}_{P_1}$ is :

$$((q_3, (P_1, m_7, P_3), q_6), \top, (q_2, @P_2, q_3), \{(q_3, @P_3, q_2), (q_3, @P_3, q_0)\}).$$

3.2 Choreography-Based Communication Mechanism

In this section, we demonstrate how monitors communicate with each other to prevent sequences formation. Upon sending a message $\mathfrak{m}$ by the process P_x, the monitor M_x

communicates with other monitors to determine the sequence formation up to $\mathfrak{m}$. In the case that $\mathfrak{m}$ is the last message in at least one sequence, P_x must be blocked before sending $\mathfrak{m}$ until M_x gets information about the partial sequence formation from others and makes sure that sending $\mathfrak{m}$ does not complete the sequence formation up to $\mathfrak{m}$. Otherwise, P_x sends the message $\mathfrak{m}$, and then its monitor tries to detect the sequence formation up to $\mathfrak{m}$.

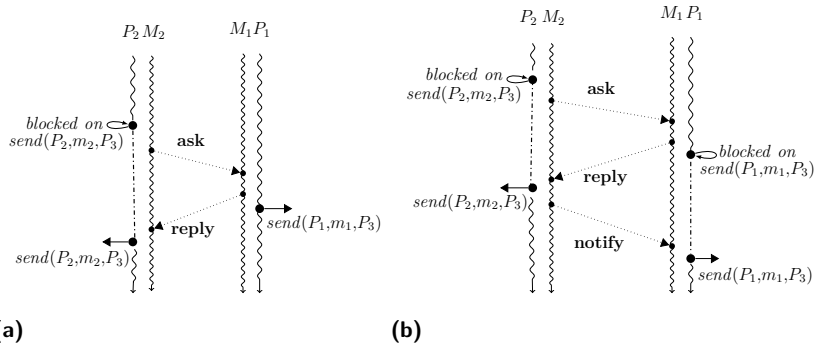

Figure 3 The monitors collaborate to avoid the sequence formation $(P_1, m_1, P_3)(P_2, m_2, P_3)$. The sequence has been formed in **(a)** as the process P_1 sends the message m_1 immediately before M_1 receives the *notify* message. However, in **(b)**, the sequence has not been formed as the process P_1 sends the message m_1 after M_1 receives the monitoring message *notify*. The dashed part of a thread denotes that the process is blocked until its monitor gets some information from other monitors.

The monitors communicate with each other using monitoring messages. There are three types of monitoring messages called *ask*, *reply*, and *notify*. A monitor sends the monitoring message *ask* to inquire if a message has been sent and receives the response by the monitoring message *reply*.

▶ **Example 4.** In Figure 3a, the monitors communicate with each other to avoid the sequence formation $(P_1, m_1, P_3)(P_2, m_2, P_3)$ at runtime. The process P_2 is blocked on the message (P_2, m_2, P_3) as it is the last message in the sequence. Then, M_2 sends the monitoring message *ask* to M_1 to check if m_1 has been sent. The monitor M_1 responds to M_2 by sending the monitoring message *reply*.

In the case that the process P_x is blocked on $\mathfrak{m}$ until the monitor M_x makes sure about the completion of the sequence formation, M_x may receive the response that the inquired message has not been sent. Due to the delay of the network, this response may be received late and meanwhile the inquired message may be sent before receiving this response. So, the process P_x sends $\mathfrak{m}$ and the sequence is formed. To avoid the sequence formation, the inquired message must not be sent by the process of the inquired monitor until the message $\mathfrak{m}$ is sent by P_x, and the monitor M_x notifies the inquired monitor.

▶ **Example 5.** In Figure 3a, M_1 responds to M_2 that the message (P_1, m_1, P_3) has not been sent. However, P_1 sends m_1 immediately after sending this response. In this case, when M_2 receives the response, it finds that m_1 has not been sent and so P_2 can send m_2 safely. But, P_2 sends the message m_2 after sending m_1 and so the sequence has been formed. In Figure 3b, the sequence is not formed as the inquired message (P_1, m_1, P_3) cannot be sent until M_1 receives the monitoring message *notify* from M_2.

3.2.1 Choreography-Based Communication Strategy

As the message sequence $\mathfrak{m}_1 \dots \mathfrak{m}_{x-1}\mathfrak{m}_x \dots \mathfrak{m}_n$ is decentralized between monitors, the monitor M_{x-1} must inform M_x the result of the sequence formation $\mathfrak{m}_1 \dots \mathfrak{m}_{x-1}$. Results can be either pushed into or pulled from a monitor. We use a pulling strategy for collaboration among monitors. With this strategy, monitors find out the order of messages more accurately. We explain the reason through an example.

As we assumed that there is no global clock, processes and monitors append their vector clocks to the events and 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 P_1 sends m_1 after m_2 has been sent, but the vector clocks of these messages are concurrent as depicted in Figure 4. With a pushing strategy, the monitor M_1 must inform the monitor M_2 the moment that m_1 has been sent, i.e., $[j, 0]$. When P_2 sends m_2, M_2 cannot conclude about the violation of the property as it has not received the moment that m_1 was sent. After pushing the moment of m_1 by M_1, M_2 cannot conclude the order among the two events accurately and decide on the property, which is not held, as the vector clocks of the messages are concurrent, i.e., $[0, i] \parallel [j, 0]$. However, with the pulling strategy, M_2 inquires about the sending status of m_1 from M_1 after sending m_2. If P_1 has not sent m_1 yet, then M_1 responds with a false result. Upon receipt of this response, M_2 can conclude accurately that the property is not violated.

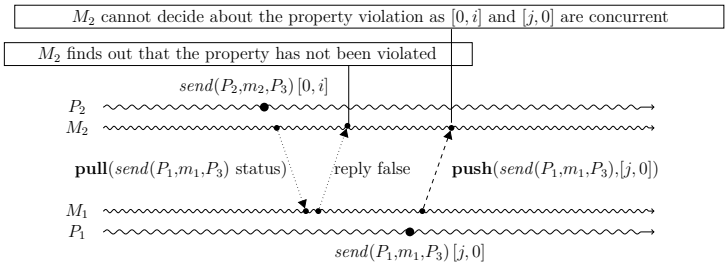

Figure 4 The two communication strategies between the monitors where the pulling strategy is denoted by dotted lines and the pushing strategy is denoted by a dashed line. The vector clock $[0, i]$, where $i > 0$, denotes that P_2 executes the event $send(P_2, m_2, P_3)$ as the i^{th} event while it has no information about the events of P_1.

4 Choreography-Based Runtime Enforcement Algorithm

In this section, we aim to introduce the choreography-based runtime enforcement algorithm, where the unwanted message sequences are specified on the sequences of *send* events.

4.1 The Process Environment

The process P_x maintains the variable *lastmessages* which denotes the list of messages labeled on transitions reach to final states. The process is blocked before sending $\mathfrak{m} \in$ *lastmessages* until its monitor makes sure that sending $\mathfrak{m}$ does not complete a sequence formation. The process and its monitor also have three shared variables $sendingmsg_x$, $blockmsg_x$, and $waitingmsg_x$, where:

- $sendingmsg_x$ is a list of triples which consists of a message, which the process is going to send, a vector clock of the process upon sending the message, and the type of a monitoring message which the monitor must send to other monitors.

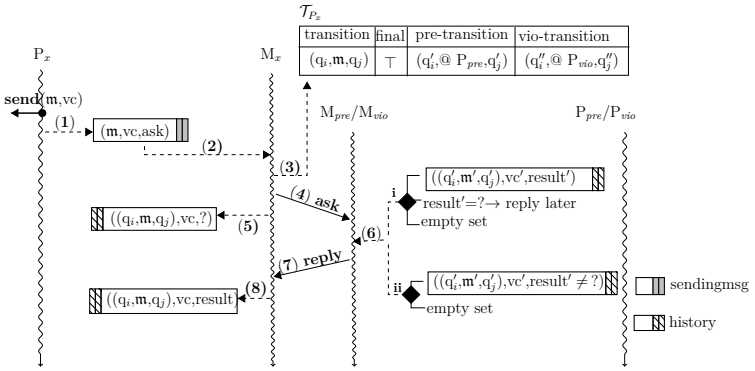

Figure 5 The algorithm steps taken upon sending the message $\mathfrak{m}$ by the process P_x where $\mathfrak{m}$ is not the last message in any sequence.

- $blockmsg_x$ is a pair of a message which P_x has been blocked on it , and the status of a message to be sent in which it can be either *ok* or *error*.
- $waitingmsg_x$ is the list of messages that must not be sent by P_x until its monitor receives a notify message as explained in Section 3.2.

We assume that the mutual exclusion of shared variables is ensured by using some well-known mechanisms like semaphore and monitors [20, 17].

4.2 The Monitor Environment

The monitor M_x maintains a transition table $\mathcal{T}_x$ as described in Section 3.1 to prevent a sequence formation. We call a transition t of the table $\mathcal{T}_x$ is *taken* if its labeled message $\mathfrak{m}$ has been sent, and a partial sequence up to the transition t has been formed. A partial sequence up to the transition t is formed if at least one of its preceding transitions has been taken before, and after that, no violating transition (of those taken preceding transition) has been taken. In the case that t has no preceding and violating transition, the transition t is taken when its labeled message has been sent. The time that the transition t is taken equals the time that $\mathfrak{m}$ was sent, and is denoted by a vector clock appended to $\mathfrak{m}$. The monitor M_x also maintains a variable $history_x$ which is the list of triples that consists of the transition t that is taken before, a vector clock of a process upon sending the message $\mathfrak{m}$, and a result of a partial sequence formation up to t.

4.3 The Algorithm Sketch

When the process P_x wants to send a message $\mathfrak{m}$, there will be two cases depending on whether $\mathfrak{m}$ is the last message in at least a sequence. In the following, we explain the behavior of the process and its monitor in the two cases.

Case 1: $\mathfrak{m}$ is not the last message in any sequence

If $\mathfrak{m}$ is not the last message in any sequence, i.e., $\mathfrak{m} \notin lastmessages$, the process P_x sends the message and appends the triple $(\mathfrak{m}, vc, ask)$ to the end of $sendingmsg_x$. The monitor M_x takes a message from $sendingmsg_x$. If the type of message is *ask*, it inspects if any transition of $\mathcal{T}_x$ can be taken. Then, 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 taken status of the pre-transition and vio-transitions in the row by sending appropriate monitoring messages to the monitors

corresponding to the sender of the messages over these transitions. Additional information is appended to the monitoring messages including the vector clock of the sending event of $\mathfrak{m}$, the inspected transition of $\mathcal{T}_x$ labeled by $\mathfrak{m}$, called t, the blocked status of the process P_x on $\mathfrak{m}$, the type of the monitoring message, and the inquired transitions. Then, M_x adds the temporary record $(t, vc, ?)$ to its history. The triple (t, vc, result) expresses that the taken 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 taken 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. Figure 5 shows the steps of the algorithm in this case and $(\mathbf{1}) - (\mathbf{5})$ denotes the steps explained so far.

▶ **Example 6.** In Figure 2, when P_2 sends the message m_3, it appends the triple $(\mathfrak{m}_3, vc, \mathit{ask})$ to the end of $\mathit{sendingmsg}_2$. Upon taking this triple from $\mathit{sendingmsg}_2$, the monitor M_2 checks the transition table $\mathcal{T}_{P_2}$ (Table 2) and finds two transitions labeled by (P_2, m_3, P_3). For instance, as the transition $(q_4, (P_2, m_3, P_3), q_6)$ has one pre-transition and one vio-transition, M_2 prepares two monitoring messages to inquire about the taken status of the pre-transition $(q_0, @P_1, q_4)$ from M_1 and the vio-transition $(q_4, @P_3, q_0)$ from M_3. Then, M_2 adds the triple $((q_4, (P_2, m_3, P_3), q_6), vc, ?)$ to its history.

Upon receiving the monitoring message ask by M_y, there are two cases according to the blocked status of P_y which is the sender of a message $\mathfrak{m}'$ labeled on the inquired transition:

(1) The process P_y is not blocked on $\mathfrak{m}'$: In this case, if M_y has either an unknown result "?" in $\mathit{history}_y$ corresponded to the inquired transition or an unhandled message in $\mathit{sendingmsg}_y$ corresponded to $\mathfrak{m}'$ in which $\mathit{send}(\mathfrak{m}') \rightsquigarrow \mathit{send}(\mathfrak{m})$, then it must postpone responding to the monitoring message. Otherwise, if M_y finds a record with a defined result value about the inquired transition, it infers that the transition has been previously taken. If so, M_y attaches the corresponding information found in its history to its response monitoring message. If there is no record with a defined result value about the inquired transition, it attaches an empty set to the monitoring message (**6i**). Then, M_y communicates with M_x by sending the monitoring message *reply*.

▶ **Example 7.** In Figure 2, suppose that the monitor M_2 inspects the taken status of the transition $(q_4, (P_2, m_3, P_3), q_6)$, and inquires about the taken status of $(q_0, @P_1, q_4)$ from M_1. If M_1 finds any record $((q_0, \mathfrak{m}', q_4), vc', ?)$ in $\mathit{history}_1$, where $\mathit{send}(\mathfrak{m}') \rightsquigarrow \mathit{send}(P_2, m_3, P_3)$, then it postpones responding to this monitoring message until the result value "?" be defined. Otherwise, it attaches the found records to the monitoring message and send to M_2.

(2) The process P_y is blocked on $\mathfrak{m}'$: The monitor M_y checks $\mathit{history}_y$ to investigate whether $\mathfrak{m}'$ has been previously sent and there is any record with a defined result value corresponding to the inquired transition. If such a record with a defined result value is found, M_y attaches the found information to its response monitoring message. Otherwise, it attaches an empty set to the monitoring message (**6ii**). Then, M_y communicates with M_x by sending the monitoring message *reply*. In this case, there may be a record corresponding to the inquired transition with the unknown result "?" in $\mathit{history}_y$. However, a defined value of this result does not affect on the sequence formation as $\mathfrak{m}'$ has not been sent yet, and the inspected message $\mathfrak{m}$ has been sent by P_x. So, M_y can send the records with a defined result value to M_x irrespective of the records with an unknown value about the inquired transition.

▶ **Example 8.** In Figure 2, if P_1 is blocked on the message (P_1, m_7, P_3), then there will be a record with an unknown result value for the transition $(q_0, (P_1, m_7, P_3), q_4)$ in $history_1$. At this time, if M_1 receives a monitoring message with the inspected transition $(q_4, (P_3, m_5, P_1), q_0)$, then M_1 does not wait for a defined result value of $(q_0, (P_1, m_7, P_3), q_4)$. As the inspected message (P_3, m_5, P_1) has been sent and the inquired message (P_1, m_7, P_3) has not been sent up to now, the sequence $(P_1, m_7, P_3)(P_3, m_5, P_1)$ cannot been formed.

When M_x receives all responses from other monitors, it checks whether the inspected transition can be taken. In this case, M_x updates the result value of the corresponding record in $history_x$ (**7**) – (**8**). Otherwise, it removes the record $(t, vc, ?)$ from $history_x$.

If there exists at least one taken pre-transition for the transition t that was not taken before its vio-transitions and its taken 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 there is no happened-before relation between the taken time of the pre-transition and the taken 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 $\mathcal{T}_x$ has no pre-transition, the monitor M_x does not consult with any monitor, and adds this transition with the result of "*Frm*" to $history_x$.

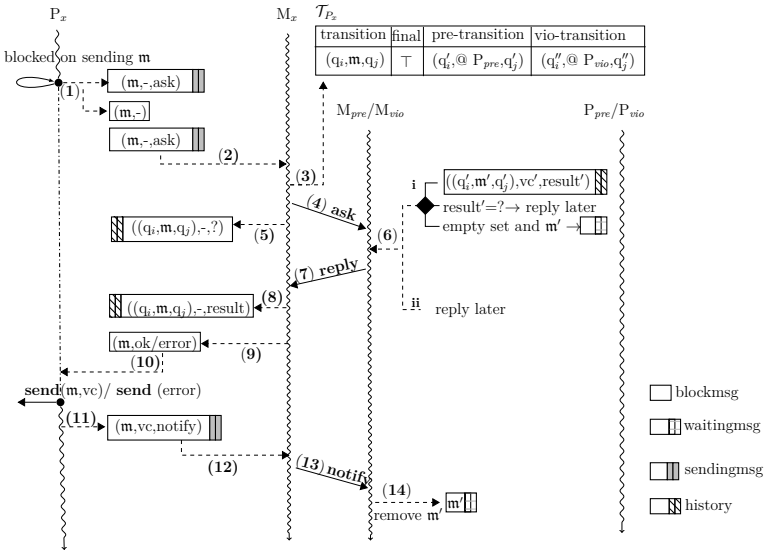

Figure 6 The algorithm steps taken upon sending the message $\mathfrak{m}$ by the process P_x where $\mathfrak{m}$ is the last message in at least one sequence.

Case 2: $\mathfrak{m}$ is the last message in at least one sequence

If $\mathfrak{m}$ is the last message in at least one sequence, i.e., $\mathfrak{m} \in lastmessages$, P_x does not send the message until it makes sure that sending $\mathfrak{m}$ does not lead to a sequence formation. Then, P_x appends the triple $(\mathfrak{m}, -, ask)$ to the end of $sendingmsg_x$, where "$-$" denotes that the process is blocked before sending $\mathfrak{m}$ and so $\mathfrak{m}$ has no assigned vector clock. The process P_x also sets the $blockmgs_x$ to $(\mathfrak{m}, -)$, and then it is blocked. Figure 6 shows the steps of the algorithm in this case. The process P_x will be blocked until its monitor updates $blockmsg_x$

to either $(\mathfrak{m}, ok)$ or $(\mathfrak{m}, error)$. The pair $(\mathfrak{m}, ok)$ denotes that no sequence will be formed by sending $\mathfrak{m}$. In this case, P_x can continue its execution and send the message safely. The pair $(\mathfrak{m}, error)$ denotes that a sequence up to the last message $\mathfrak{m}$ has been formed and so sending $\mathfrak{m}$ leads to a complete sequence formation. The monitor M_x behaves similarly to the previous case upon taking a message from $sendingmsg_x$ $(\mathbf{2}) - (\mathbf{4})$. Then, M_x adds the temporary record $(t, -, ?)$ to $history_x$ $(\mathbf{5})$.

Upon receiving the monitoring message by M_y, there are two cases based on the blocked status of P_y which is the sender of the message labeled on the inquired transition, i.e., $\mathfrak{m}'$:

(1) The process P_y is not blocked on $\mathfrak{m}'$: The monitor M_y behaves similarly to the first item of the previous case, except that it must also add $\mathfrak{m}'$ to $waitingmsg_y$ if it finds no record with a defined result value about the inquired transition (**6i**).

(2) The process P_y is blocked on $\mathfrak{m}'$: As the process P_y is blocked on $\mathfrak{m}'$, there will be a record $((q, \mathfrak{m}', q'), -, ?)$ in $history_y$. The monitor M_y must postpone responding to this monitoring message as it does not know whether a sequence up to $\mathfrak{m}'$ is formed (**6ii**).

The monitor M_x behaves similarly to the previous case upon receiving all responses from other monitors. If a sequence up to $\mathfrak{m}$ is formed, then M_x updates $blockmsg_x$ to $(\mathfrak{m}, error)$ to inform P_x that sending $\mathfrak{m}$ leads to a complete sequence formation. Otherwise, it updates $blockmsg_x$ to $(\mathfrak{m}, ok)$ to inform P_x that $\mathfrak{m}$ can be sent safely $(\mathbf{7}) - (\mathbf{9})$.

The process P_x either sends $\mathfrak{m}$ or sends an error message regarding the status of the message in $blockmsg_x$, and appends the triple $(\mathfrak{m}, vc, notify)$ to the end of $sendingmsg_x$ $(\mathbf{10}) - (\mathbf{11})$. The monitor M_x takes the triple with the message type *notify* from $sendingmsg_x$ and sends the corresponding monitoring message $(\mathbf{12}) - (\mathbf{13})$. If P_x sends $\mathfrak{m}$ by the vector clock vc, M_x also updates the vector clock of the transition labeled by $\mathfrak{m}$ in $history_x$ from "$-$" to vc. Finally, the monitor M_y which receives the notify message from M_x, removes the message labeled on the inquired transition from $waitingmsg_y$ $(\mathbf{14})$.

4.4 Discussion

We have assumed that processes and their monitors behave honestly and do not suffer from any failures or byzantine behavior [14]. If a monitor fails or a process fails before updating the shared variable, the algorithm will not be sound due to the loss or the incomplete information of the monitor. In cases that processes tamper with events or behave maliciously, they may not inform their monitors upon the occurrence of *send*/*take* events. Hence, monitors conclude wrongly and the algorithm will not be sound again. The proposed algorithm can be implemented in the execution framework of message-based systems. For instance, the send function of the open source Akka library [2] or the control layer of Theater [8] which regulates the message scheduling and dispatching can be modified to incorporate our enforcement algorithm. On the other hand, the given specification for the unwanted message sequences may lead our algorithm to reach a communication deadlock [31] among the monitors. For instance, suppose that we aim to prevent the formation of the message sequences $\mathfrak{m}_1\mathfrak{m}_2$ and $\mathfrak{m}_2\mathfrak{m}_1$. According to our algorithm, the process P_2 is blocked before sending the message $\mathfrak{m}_2$ and then M_2 asks M_1 if $\mathfrak{m}_1$ has been sent. The process P_1 also may be blocked before sending $\mathfrak{m}_1$ and then M_1 inquires M_2 if $\mathfrak{m}_2$ has been sent. So, both processes P_1 and P_2 will be blocked as their monitors cannot determine the sequence formation up to their blocked messages. However, our algorithm works correctly for sequences without such dependencies. The proof sketch of our algorithm is given in Appendix A. We are working on an extended version of our algorithm to detect and resolve communication deadlocks as a future work.

5 Evaluation and Experimental Results

In this section, we present the results of a set of experiments to evaluate our runtime enforcement algorithm. We investigate the effect of different parameters on the efficiency of the algorithm including the number of processes, the maximum number of message handlers of processes, the maximum message communication chain between processes, and the length of the message sequences. The maximum message communication chain denotes the maximum number of processes in a chain of message handlers that send messages to each other. We develop a test case generator [1] which produces message-based applications with different parameters and a set of message sequences according to the generated application. Applications are generated in terms of a simple actor-based language [1]. We also develop a simulator [2] which simulates the execution of each application and our prevention algorithm, and then measures the communication overhead of our algorithm. The simulator tools assume a random network delay and our simulator delivers messages after this delay. We perform all the experiment on a single machine with a dual core processor (Intel i5-520M 2.4GHz) with 4 GB memory.

To evaluate the scalability and the monitoring communication overhead of our algorithm, we generate four applications with 3, 6, 9, and 12 processes, where each process has maximum five message handlers, and the maximum message communication chain in each application is 4, 5, 6, and 7, respectively.

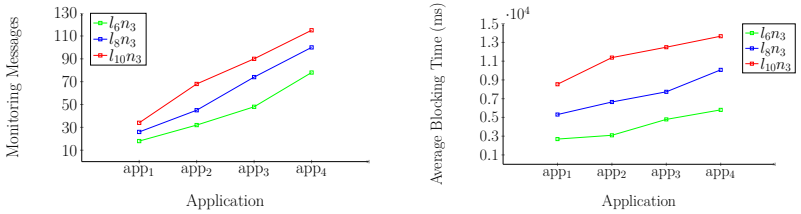

Figure 7 (a) The average number of monitoring messages in different applications regarding three sequences (denoted by n_3) with different lengths of six (denoted by l_6), eight, and ten, (b) The average time that processes are blocked regarding the same message sequences of (a).

Scalability: To show that our algorithm is scalable in terms of the average number of monitoring messages and the average blocking time of processes, we run each experiment ten times. The average number of monitoring messages for each application is shown in Figure 7a where the given properties are three message sequences with the length of six, eight, and ten. The number of times that a monitor inquires others because of a message $\mathfrak{m}$ (occurring in sequences) depends on the number of times that the message $\mathfrak{m}$ has occurred at runtime. 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. Our results show that as the length of message communication chain increases, the number of monitoring messages grows linearly for complex applications.

We also evaluate the average time that the processes of each application are blocked. In the proposed algorithm, only the senders of the last messages in the sequences are blocked. Figure 7b shows that the average blocking time of processes grows linearly for complex application to prevent the formation of three message sequences with the length of six, eight, and ten.

[1] Available at `https://gitlab.com/vmoh.ir/rebeca-generator`, Accessed: 2020-11-04
[2] Available at `https://gitlab.com/mSamadi/enforcement`, Accessed: 2020-11-04

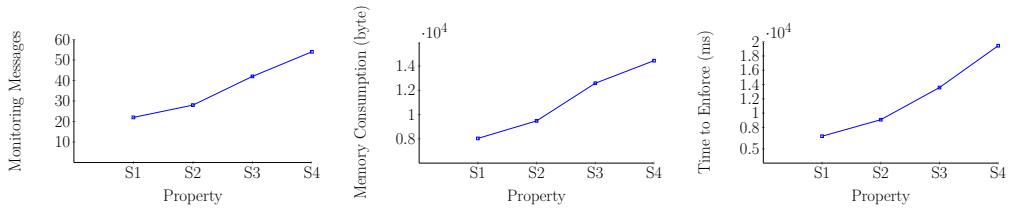

Figure 8 (a) the average number of monitoring messages, (b) the average of memory consumption, (c) the average time for preventing the message sequence formation for different properties where S_i denotes three sequences with the length of $i + 4$.

Monitoring Communication Overhead:

We evaluate the average number of monitoring messages, the average of memory consumption of the monitors, and the average time to enforce a property for the application with nine processes. As illustrated in Figure 8a and Figure 8b, the average number of monitoring messages and the average memory consumption of the monitors grows linearly as the length of sequences increases. To measure the average time to enforce the property, we measure the average time that the monitors are waited for receiving the responses from other monitors, plus the total time that a process is blocked until its monitor informs it to send a message. It is shown in Figure 8c that as the length of the sequence increases, the monitors involve in more collaborations and hence, more time to gather all responses from other monitors.

6 Related Work

Several *centralized* monitoring algorithms [34, 4, 9] and *decentralized* ones [32, 27, 3] have been proposed to *detect* the property violation in distributed systems at runtime . Among the centralized runtime enforcement approaches which aim to *avoid* the property violation, we can mention [33] which introduces security automata to specify security properties. Using this model, the execution of the program is stopped if a sequence of events does not satisfy the desired property. Using the edit automaton [22], the execution of the program can be corrected by suppressing or inserting a new event. This automaton assumes that monitors can predetermine the results of events without executing them. In [10, 23], an enforcement model is presented for the cases that the results of events are not predetermined. In the presented model, for every event generated by the program, the underlying executing system returns a result to the target program. The predictive runtime enforcement [29] deals with systems that are not entirely black-box, and there is some knowledge about their behavior. The knowledge allows to output some events immediately, and the system is not blocked until more events are observed. The timed properties are enforced, in [12] at runtime. Furthermore, in [6], an enforcement approach for the reactive systems is presented where the output should be corrected only if necessary, as little as possible, and without delay. In addition to these work, [11, 7] deals with the runtime enforcement of component-based systems, where systems are modeled within the BIP framework [5]. In this approach, monitors are synchronized with their components. However, the proposed algorithm is decentralized and monitors collaborate to prevent the unwanted sequences formation.

The existing approaches in the domain of runtime enforcement are categorized in [13], and decentralized runtime enforcement is considered as an open challenge in distributed systems. We can address [15] as a decentralized enforcement approach in which a framework, called service automata, is specified in Hoare's CSP language [18]. This framework considers networks of service automata that are not fully connected. Each service automaton

synchronizes with the system on the critical events. This automaton controls the execution of a program and communicates with other service automata to decide whether a property is satisfied. However, in our choreography-based approach, the monitors are fully connected and a monitor can communicate with others monitors directly and so fewer monitoring messages are transmitted in the network. In addition, there is no synchronization among processes and their monitors, and the monitors take advantage of the specific communication mechanism (Sect. 3.2) to prevent the scenario of sequence formation given in Figure 3. The work of [16] is considered as a decentralized enforcement approach in the domain of business processes where a document must follow a specific workflow. It uses the notion of migration strategy [9] where the document is transmitted among different parties. The document carries fragments of its history, and is protected from tampering using hashing and encryption. Here, the workflow as a specification is shared among different parties as opposed to our method.

7 Conclusion and Future work

We addressed the choreography-based runtime prevention of message sequences formation in systems where 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 the sense that each process is equipped with a monitor which has partial access to some parts of the property specification. Monitors cannot identify the total ordering among messages using the vector clock and hence, may prevent a sequence formation conservatively. We developed a simulator to evaluate the effect of different application and the length of the message sequences on various factors, including the number of monitoring messages, memory consumption of the monitors, and the time to prevent the sequence formation. Our experimental results show that with the increase of the complexity of application or the length of message sequences, the number of monitoring messages, memory consumption, and the time to prevent the sequence formation grows linearly. We are going to resolve the possible communication deadlock based on the given message sequences in the future and integrate our algorithm with the AKKA library.

References

1 Gul. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence, 1990.

2 Akka. `https://akka.io`. Accessed: 2020-09-09.

3 Bavid Basin, Felix Klaedtke, and Eugen Zălinescu. Runtime verification of temporal properties over out-of-order data streams. In *Proc. CAV*. Springer, 2017.

4 Andreas Bauer and Yliès Falcone. Decentralized LTL monitoring. In *Proc. FM*. Springer, 2012.

5 Simon Bliudze and Joseph Sifakis. The algebra of connectors—structuring interaction in bip. *Journal of TC. IEEE*, 57(10):1–16, 2008.

6 Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: runtime enforcement for reactive systems. In *Proc. TACAS*. Springer, 2015.

7 Hadil Charafeddine, Khalil El-Harake, Yliès Falcone, and Mohamad Jaber. Runtime enforcement for component-based systems. In *Proc. SAC*. ACM, 2015.

8 Franco Cicirelli, Libero Nigro, and Paolo Sciammarella. Model continuity in cyber-physical systems:a control-centered methodology based on agents. *Journal of Simul Model Pract Theory. Elsevier*, 83:93–107, 2018.

9 Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with a global clock. *Journal of FMSD. Springer*, 42(1):109–158, 2016.

10 Egor Dolzhenko, Jay Ligatti, and Srikar Reddy. Modeling runtime enforcement with mandatory results automata. *JIS. Springer*, 14(2):47–60, 2015.

11 Yliès Falcone and Mohamad Jaber. Fully automated runtime enforcement of component-based systemswith formal and sound recovery. *Journal of STTT. Springer*, 19(3):341–365, 2017.

12 Yliès Falcone, Thierry Jéron, Hervé Marchand, and Srinivas Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Journal of SCP. Elsevier*, 123(1):2–41, 2016.

13 Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. Runtime failure prevention and reaction. In *Lectures on Runtime Verification*. Springer, 2018.

14 Yliès Falcone, Robert Shostak, and Marshall Pease. The byzantine generals problem. *Journal of TOPLAS. ACM*, 4(3):382–401, 1982.

15 Richard Gay, Heiko Mantel, and Barbara Sprick. Serveice automata. In *Proc. FAST*. Springer, 2011.

16 Sylvain Hall, Raphaël Khoury, Quentin Betti, Antoine El-Hokayem, and Yliès Falcone. Decentralized enforcement of document lifecycle constraints. *JIS. ACM*, 74(2), 2018.

17 Brinch Hansen. *The Origin of Concurrent Programming*. Springer-Verlag, 2002.

18 C.A.R. Hoare. *Communicating sequential processes*. Prentice-Hall International series in computer science, 1985.

19 Ilya Kolchinsky and Assaf Schuster. Efficient adaptive detection of complex event patterns. In *Proc. VLDB*. ACM, 2018.

20 Ajay D Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.

21 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *JCM. ACM*, 21(7):558–565, 1978.

22 Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *JIS. Springer*, 4(1):2–16, 2005.

23 Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *Proc. ESORICS*. Springer, 2010.

24 Carmen. T Lopez, Stefan Marr, Elisa Gonzalez, and Hanspeter Mössenböck. *A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs, Programming with Actors*. Programming with Actors, 2018.

25 Friedemann Mattern. *Virtual Time and Global States of Distributed Systems, Parallel and Distributed Algorithms*. North-Holland Press, 1988.

26 Barton. P Miller and J. D Choi. Breakpoints and halting in distributed programs. In *Proc. ICDCS*. IEEE, 1988.

27 Menna Mostafa and borzoo Bonakdarpour. Decentralized runtime verification of ltl specifications in distributed systems. In *Proc. IPDPS*. IEEE, 2015.

28 Saravana.M Palanisamy, Muhammad.A Tariq Frank Dürr, and Kurt Rothermel. Preserving privacy and quality of service in complex event processing through event reordering. In *Proc. DEBS*. ACM, 2018.

29 Srinivas Pinisetty, Viorel Preoteasa, Stavros Tripakis, Thierry Jéron, Yliès Falcone, and Hervé Marchand. Predictive runtime enforcement. *Journal of FMSD. Springer*, 51(3):154–199, 2017.

30 Yingmei Qi, Lei Cao, Medhabi Ray, and Elke A. Rundensteiner. Complex event analytics: online aggregation of stream sequence patterns. In *Proc. SIGMOD*. ACM, 2014.

31 Panos Rondogiannis, Georgios Pavlides, and A. Levy. Distributed algorithm for communication deadlock detection. *Journal of IST. Elsevier*, 33(7):483–488, 1991.

32 Mahboubeh Samadi, Fatemeh Ghassemi, and Ramtin Khosravi. Choreography-based runtime verification of message sequences in distributed message-based systems. available in http://rebeca-lang.org/assets/papers/archive/rv.pdf.

33 Fred. B Schneider. Enforceable security policies. *Journal of TISSEC. ACM*, 3(1):30–50, 2000.

34 César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, and et al. A survey of challenges for runtime verification from advanced application domains (beyond software). *Journal of FMSD. Springer*, 54(3):273–335, 2018.

A Soundness of The Algorithm

We aim to prove that the proposed algorithm is *sound* meaning that the output of a message-based system is correct and no given message sequences will be formed at runtime.

▶ **Lemma 9.** *For any sequence* $\mathfrak{m}_1 \dots \mathfrak{m}_{n-1}\mathfrak{m}_n \in \mathcal{L}(\mathcal{A})$*, the corresponding monitor of* $\mathfrak{m}_{n-1}$ *declares the formation of* $\mathfrak{m}_1 \dots \mathfrak{m}_{n-1}$ *correctly.*

Proof. It is trivial that if $\omega = \mathfrak{m}_1 \dots \mathfrak{m}_{n-1}\mathfrak{m}_n \in \mathcal{L}(\mathcal{A})$, then there exists at least a sub-sequence $\mathfrak{m}_1^i\mathfrak{m}_2^j\mathfrak{m}_3^w \dots \mathfrak{m}_l^h \dots \mathfrak{m}_n^k$, called $\widehat{\omega}$, where for the message $\mathfrak{m}_l^h$, h is the index of the message in ω and l is the index of the message in $\widehat{\omega}$, i. e., $0 \leq l \leq n$. For each pair of $\mathfrak{m}_{l'}^{i'}\mathfrak{m}_{l'+1}^{j'}$ of $\widehat{\omega}$, there is no message $\mathfrak{m}_o$ in ω, where $i' < o < j'$, that cancel the effect of $\mathfrak{m}_{l'}^{i'}$. In other words, the messages of $\widehat{\omega}$ comprise 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 $\mathfrak{m}_{n-1}$, declares the formation of $\mathfrak{m}_1 \dots \mathfrak{m}_{n-1}$ correctly.

The message $\mathfrak{m}_i$ has occurred before the message $\mathfrak{m}_j$, denoted by $\mathfrak{m}_i \rightarrow \mathfrak{m}_j$, if and only if $\neg(vc(\mathfrak{m}_j) < vc(\mathfrak{m}_i))$. So, it can be concluded that in $\widehat{\omega}$, either $\mathfrak{m}_1^i$ has happened before $\mathfrak{m}_2^j$ or $\mathfrak{m}_1^i$ is concurrent with $\mathfrak{m}_2^j$ i.e., $\mathfrak{m}_2^j \not\rightsquigarrow \mathfrak{m}_1^i$ in short. By running our algorithm, the monitor of $\mathfrak{m}_2^j$, namely M_2, checks the taken status of its pre-transitions and the vio-transitions of the pre-transitions ((3) in Figure 5). So, a transition labeled by $\mathfrak{m}_1^i$, called t, and its corresponding vio-transitions are investigated. If a message belonging to the vio-transitions of t, called $\mathfrak{m}_v$, has occurred in ω, it must have occurred before $\mathfrak{m}_1^i$, where $v < i$, in ω due to our condition on the sub-sequence. Two cases can be distinguished: either the message $\mathfrak{m}_v$ has happened before $\mathfrak{m}_1^i$ or $\mathfrak{m}_v$ is concurrent with $\mathfrak{m}_1^i$. In the first case, where the vector clock of $\mathfrak{m}_v$ is less than the vector clock of $\mathfrak{m}_1^i$, the effect of $\mathfrak{m}_i$ has not been canceled by $\mathfrak{m}_v$. So, M_2 checks the vector clocks of $\mathfrak{m}_1^i$ and $\mathfrak{m}_2^j$. If $\mathfrak{m}_1^i$ has happened before $\mathfrak{m}_2^j$, M_2 concludes that the sequence $\mathfrak{m}_1^i\mathfrak{m}_2^j$ has been formed so far and stores the "*Frm*" result in its history. If there is no relation between the vector clocks of $\mathfrak{m}_1^i$ and $\mathfrak{m}_2^j$, M_2 behaves conservatively and stores the result value "Frm_p" in its history ((8) in Figure 5). In the second case, where the vector clock of $\mathfrak{m}_1^i$ is concurrent with $\mathfrak{m}_v$, M_2 does not know whether $\mathfrak{m}_v$ has cancel the effect of $\mathfrak{m}_1^i$ and so it adds the result value "Frm_p" to its history since $\mathfrak{m}_2^j \not\rightsquigarrow \mathfrak{m}_1^i$.

Up to here, the result with "*Frm*" or "Frm_p" value has been correctly inserted into the M_2's history. With the same discussion, we select $\mathfrak{m}_3^w$ of $\widehat{\omega}$ and assume that the message $\mathfrak{m}_{v'}$, which cancels the effect of $\mathfrak{m}_2^j$, has occurred and due to our condition on the sub-sequence, it must have occurred before $\mathfrak{m}_2^j$ in ω. So, the message $\mathfrak{m}_2^j$ has happened before $\mathfrak{m}_3^w$ or $\mathfrak{m}_2^j$ is concurrent with $\mathfrak{m}_3^w$. By applying our algorithm, the monitor of $\mathfrak{m}_3^w$, namely M_3, inquiries about $\mathfrak{m}_2^j$ and $\mathfrak{m}_{v'}$. The monitor M_3 compares the received information about $\mathfrak{m}_2^j$ and $\mathfrak{m}_{v'}$ and decides whether $\mathfrak{m}_{v'}$ cancels the effect of $\mathfrak{m}_2^j$. If $\mathfrak{m}_{v'}$ has not canceled the effect of $\mathfrak{m}_2^j$, M_3 investigates the vector clocks of $\mathfrak{m}_2^j$ and $\mathfrak{m}_3^w$. If $\mathfrak{m}_2^j$ has happened before $\mathfrak{m}_3^w$, then M_3 stores the result value "Frm_p" or "*Frm*" sent by M_2 to the history. If there is no relation between the vector clocks of $\mathfrak{m}_2^j$ and $\mathfrak{m}_3^w$, it behaves conservatively and stores the "Frm_p" result to its history. These scenarios will be continued to reach the message $\mathfrak{m}_{n-1}$ and the "*Frm*" or "Frm_p" results values have been correctly propagated to the monitor of the message $\mathfrak{m}_{n-1}$ and hence declares the false verdict. ◀

▶ **Theorem 10.** *In a message-based system $D = \{P_1 \dots P_n\}$, no message sequence $\omega \in \mathcal{L}(\mathcal{A})$ will be formed at runtime.*

Proof. We prove by contradiction: suppose that the unwanted message sequence $\mathfrak{m}_1 \dots \mathfrak{m}_n$ is formed at runtime. Based on the second case of the proposed algorithm, the process P_n is blocked before sending the message $\mathfrak{m}_n$ and its corresponding monitor M_n inquires the sending status of $\mathfrak{m}_{n-1}$ from M_{n-1}. The process P_{n-1} cannot be blocked as $\mathfrak{m}_{n-1}$ is not the last message in the sequence. There are two cases depending on the response of M_{n-1}:

(1) The message $\mathfrak{m}_{n-1}$ has not been sent: The monitor M_{n-1} responds to M_n that $\mathfrak{m}_{n-1}$ has not been sent and adds $\mathfrak{m}_{n-1}$ to $waitinglist_{n-1}$. In this case, The message $\mathfrak{m}_{n-1}$ cannot be sent until $\mathfrak{m}_n$ has been sent and then M_n sends *notify* to M_{n-1}. As M_n finds that $\mathfrak{m}_{n-1}$ has not been sent, it informs P_n to send m_n safely ((10) in Figure 6). After sending $\mathfrak{m}_n$, the monitor M_n sends *notify* to M_{n-1} ((13) in Figure 6). Then, M_{n-1} removes $\mathfrak{m}_{n-1}$ from $waitinglist_{n-1}$ and after that P_{n-1} can send $\mathfrak{m}_{n-1}$. Hence, the message $\mathfrak{m}_n$ has been sent after $\mathfrak{m}_{n-1}$ and is contradicted by the assumption that $\mathfrak{m}_{n-1}\mathfrak{m}_n$ is formed.

(2) The message $\mathfrak{m}_{n-1}$ has been sent: The monitor M_{n-1} send the records of its history that are related to $\mathfrak{m}_{n-1}$ to M_n. By Lemma 1, the monitor M_{n-1} responds correctly if $\mathfrak{m}_1 \dots \mathfrak{m}_{n-1}$ has been formed. There are two cases depending on the result of the received records: If there is any record which its result is *Frm* or Frm_p, M_n finds that the sequence $\mathfrak{m}_1 \dots \mathfrak{m}_{n-1}$ is formed. Hence, it informs P_n to send an error message instead of $\mathfrak{m}_n$. So, the message $\mathfrak{m}_n$ has not been sent and the sequence $\mathfrak{m}_1 \dots \mathfrak{m}_n$ is not formed and hence it is a contradiction. Otherwise, since there is no record with the result of *Frm* or Frm_p, M_n finds that the sequence $\mathfrak{m}_1 \dots \mathfrak{m}_{n-1}$ is not formed and informs P_n to send $\mathfrak{m}_n$ safely. So, the message $\mathfrak{m}_n$ has been sent as the sequence $\mathfrak{m}_1 \dots \mathfrak{m}_{n-1}$ has not been formed. This is also contradicted by the formation of $\mathfrak{m}_1 \dots \mathfrak{m}_{n-1}$. ◀