Failure-aware Runtime Verification of Distributed Systems

David Basin[1], Felix Klaedtke[2], and Eugen Zălinescu[1]

1 **ETH Zürich, Department of Computer Science, Zürich, Switzlerland**
2 **NEC Labs Europe, Heidelberg, Germany**

Abstract

Prior runtime-verification approaches for distributed systems are limited as they do not account for network failures and they assume that system messages are received in the order they are sent. To overcome these limitations, we present an online algorithm for verifying observed system behavior at runtime with respect to specifications written in the real-time logic MTL that efficiently handles out-of-order message deliveries and operates in the presence of failures. Our algorithm uses a three-valued semantics for MTL, where the third truth value models knowledge gaps, and it resolves knowledge gaps as it propagates Boolean values through the formula structure. We establish the algorithm's soundness and provide completeness guarantees. We also show that it supports distributed system monitoring, where multiple monitors cooperate and exchange their observations and conclusions.

1998 ACM Subject Classification C.2.4 Distributed Systems, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.1 Mathematical Logic

Keywords and phrases Runtime verification, monitoring algorithm, real-time logics, multi-valued semantics, distributed systems, asynchronous communication

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2015.590

1 Introduction

Distributed systems are omnipresent and complex, and they can malfunction for many reasons, including software bugs and hardware or network failures. Runtime monitoring is an attractive option for verifying at runtime whether a system behavior is correct with respect to a given specification. But distribution opens new challenges. The monitor itself becomes a component of the (extended) system and like any other system component it may exhibit delays, finite or even infinite, when communicating with other components. Moreover, the question arises whether monitoring itself can be distributed, thereby increasing its efficiency and eliminating single points of failure. Distribution also offers the possibility of moving the monitors close to or integrating them in system components, where they can more efficiently observe local system behavior.

Various runtime-verification approaches exist for different kinds of distributed systems and specification languages [19, 3, 8, 15]. These approaches are of limited use for monitoring distributed systems where components might crash or network failures can occur, for example, when a component is temporarily unreachable and a monitor therefore cannot learn the component's behavior during this time period. Even in the absence of failures, monitors can receive messages about the system behavior in any order due to network delays. A naive solution for coping with out-of-order message delivery is to have the monitor buffer messages and reorder them before processing them. However, this can delay reporting a violation when

35th IARCS Annual Conf. Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015).
Editors: Prahladh Harsha and G. Ramalingam; pp. 590–603

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the violation is already detectable on some of the buffered messages. Another limitation concerns the expressivity of the specification languages used by these monitoring approaches. It is not possible to express real-time constraints, which are common requirements for distributed systems. Such constraints specify, for example, deadlines to be met.

In this paper, we present a monitoring algorithm for the real-time logic MTL [11, 1] that overcomes these limitations. Our algorithm accounts for out-of-order message deliveries and soundly operates in the presence of failures that, for example, cause components to crash. In the absence of failures, we also provide completeness guarantees, meaning that a monitor eventually reports the violation or the satisfaction of the given specification. Furthermore, our algorithm allows one to distributively monitor a system. To achieve this, the system is extended with monitoring components that receive observations from system components about the system behavior and the monitors cooperate and exchange their conclusions.

Our monitoring algorithm builds upon a timed model for distributed systems [7]. The system components use their local clocks to timestamp observations, which they send to the monitors. The monitors use these timestamps to determine the elapsed time between observations, e.g., to check whether real-time constraints are met. Furthermore, the timestamps totally order the observations. This is in contrast to a time-free model [9], where the events of a distributed system can only be partially ordered, e.g., by using Lamport timestamps [12]. However, since the accuracy of existing clocks is limited, the monitors' conclusions might only be valid for the provided timestamps. See Section 5 where we elaborate on this point.

We base our monitoring algorithm on a three-valued semantics for the real-time logic MTL, where the interpretation of the third truth value, denoted by $\bot$, follows Kleene logic [10]. For example, a monitor might not know the Boolean value of a proposition at a time point because a message from a system component about the proposition's truth value is delayed, or never sent or received. In this case, the monitor assigns the proposition the truth value $\bot$, indicating that the monitor has a knowledge gap about the system behavior. The truth value $\bot$ is also used by monitors to avoid issuing incorrect verdicts about the system behavior: a monitor only reports the satisfaction of the specification or its negation, and this verdict remains valid no matter how the monitor's knowledge gaps are later resolved when receiving more information about the system behavior. No verdict is output if under the current knowledge the specification evaluates to $\bot$.

To efficiently resolve knowledge gaps and to compute verdicts, each monitor maintains a data structure that stores the parts of the specification—a subformula and an associated time point—that have not yet been assigned a Boolean value. Intuitively speaking, the truth value of the subformula at the given time is $\bot$ under the monitor's current knowledge. These parts are nodes of an AND-OR-graph, where the edges express constraints for assigning a Boolean value to a node. When a monitor receives additional information about the system behavior, it updates its graph structure by adding and deleting nodes and edges, based on the message received. To compute verdicts, the monitors also propagate Boolean values between nodes when possible.

Our main contribution is a novel monitoring algorithm for MTL specifications. It is the first algorithm that efficiently handles observations that can arrive at the monitor in any order. This feature is essential for our approach to monitoring distributed systems, which is our second contribution. Our approach overcomes the limitations of prior runtime-verification approaches for distributed systems. Namely, it handles message delays, it allows one to distribute the monitoring process across multiple system components, and it soundly accounts for failures such as crashes of system components.

We proceed as follows. In Section 2, we introduce the real-time logic MTL with a three-valued semantics. In Section 3, we describe the system assumptions and the requirements.

Table 1 Truth tables for three-valued logical operators (strong Kleene logic [10]).

$\neg$	
t	f
f	t
$\bot$	$\bot$

$\vee$	t	f	$\bot$
t	t	t	t
f	t	f	$\bot$
$\bot$	t	$\bot$	$\bot$

$\wedge$	t	f	$\bot$
t	t	f	$\bot$
f	f	f	f
$\bot$	$\bot$	f	$\bot$

$\rightarrow$	t	f	$\bot$
t	t	f	$\bot$
f	t	t	t
$\bot$	t	$\bot$	$\bot$

In Section 4, we present our monitoring algorithm. In Section 5, we consider the impact of the accuracy of timestamps for ordering observations. In Section 6, we discuss related work. Finally, in Section 7, we draw conclusions. Details, omitted due to space restrictions, can be found in the full version of this paper, which is available from the authors or their webpages.

2 Three-Valued Metric Temporal Logic

For Σ an alphabet, we work with words that are finite or infinite sequences of tuples in $\Sigma \times \mathbb{Q}_+$, where $\mathbb{Q}_+$ is the set of positive rational numbers. We write $|w| \in \mathbb{N} \cup \{\infty\}$ to denote the length of w and (σ_i, τ_i) for the tuple at position i. A *timed word* w is a word where:

(i) $\tau_{i-1} < \tau_i$, for all $i \in \mathbb{N}$ with $0 < i < |w|$, and
(ii) If $|w| = \infty$ then for every $t \in \mathbb{Q}_+$, there is some $i \in \mathbb{N}$ such that $\tau_i > t$.

Observe that (1) requires that the sequence of the τ_is is strictly increasing rather than requiring only that the τ_is increase monotonically, as, e.g., in [1]. This means that there are no fictitious clocks that order tuples with equal τ_is. Instead, it is assumed that everything at time τ_i happens simultaneously and the τ_is already totally order the tuples that occur in w.

We denote the set of infinite timed words over the alphabet Σ by $TW^\omega(\Sigma)$. We often write a timed word $w \in TW^\omega(\Sigma)$ as $(\sigma_0, \tau_0)(\sigma_1, \tau_1)\ldots$. We call the τ_is *timestamps* and the indices of the elements in the sequence *time points*. For $\tau \in \mathbb{Q}_+$, let $\mathrm{tp}(w, \tau)$ be w's time point i with $\tau_i = \tau$ if it exists. Otherwise, $\mathrm{tp}(w, \tau)$ is undefined.

Let $\mathsf{3}$ be the set $\{\mathsf{t}, \mathsf{f}, \bot\}$, where t (true) and f (false) denote the Boolean values, and $\bot$ denotes the truth value "unknown." Table 1 shows the truth tables of some standard operators over $\mathsf{3}$. Observe that these operators coincide with the Boolean ones when restricted to the set $\mathsf{2} := \{\mathsf{t}, \mathsf{f}\}$ of Boolean values.

We partially order the elements in $\mathsf{3}$ by their knowledge: $\bot \prec \mathsf{t}$ and $\bot \prec \mathsf{f}$, and t and f are incomparable as they carry the same amount of knowledge. Note that $(\mathsf{3}, \prec)$ is a lower semilattice, where $\curlywedge$ denotes the meet.

Throughout the paper, let P be a set of atomic propositions. We extend the partial order $\prec$ over $\mathsf{3}$ to timed words over the alphabet $\Sigma := \mathsf{3}^P$, where X^Y is the set of functions with domain Y and range X. Let $v, v' \in TW^\omega(\Sigma)$, where (σ_i, τ_i) and (σ'_i, τ'_i) are the tuples at position i in v and v', respectively. We define $v \preceq v'$ if $|v| = |v'|$, $\tau_i = \tau'_i$, and $\sigma_i(p) \preceq \sigma'_i(p)$, for every i with $0 \leq i < |v|$ and every $p \in P$. Intuitively, some of the knowledge gaps about the propositions' truth values in v are resolved in v'.

The syntax of the real-time logic MTL is given by the grammar: $\varphi ::= \mathsf{t} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\, \mathsf{S}_I\, \varphi \mid \varphi\, \mathsf{U}_I\, \varphi$, where p ranges over P's elements and I ranges over intervals over $\mathbb{Q}_+$. For brevity, we omit the temporal connectives for "previous" and "next." MTL's three-valued

semantics is defined as follows. Let $i \in \mathbb{N}$ and $w \in TW^\omega(\Sigma)$ with $w = (\sigma_0, \tau_0)(\sigma_1, \tau_1)\ldots$.

$$\begin{aligned}
[\![w, i \models \mathsf{t}]\!] &:= \mathsf{t} \\
[\![w, i \models p]\!] &:= \sigma_i(p) \\
[\![w, i \models \neg\varphi]\!] &:= \neg[\![w, i \models \varphi]\!] \\
[\![w, i \models \varphi \vee \psi]\!] &:= [\![w, i \models \varphi]\!] \vee [\![w, i \models \psi]\!] \\
[\![w, i \models \varphi \,\mathsf{S}_I\, \psi]\!] &:= \bigvee_{j \in \{\ell \in \mathbb{N} \mid \tau_i - \tau_\ell \in I\}} \big([\![w, j \models \psi]\!] \wedge \textstyle\bigwedge_{j < k \leq i} [\![w, k \models \varphi]\!]\big) \\
[\![w, i \models \varphi \,\mathsf{U}_I\, \psi]\!] &:= \bigvee_{j \in \{\ell \in \mathbb{N} \mid \tau_\ell - \tau_i \in I\}} \big([\![w, j \models \psi]\!] \wedge \textstyle\bigwedge_{i \leq k < j} [\![w, k \models \varphi]\!]\big)
\end{aligned}$$

Furthermore, for $\tau \in \mathbb{Q}_+$, let $[\![w \models \varphi]\!]^\tau := [\![w, \mathrm{tp}(w, \tau) \models \varphi]\!]$ if $\mathrm{tp}(w, \tau)$ is defined, and $[\![w \models \varphi]\!]^\tau := \bot$, otherwise. Note that we abuse notation here and unify MTL's constant t with the Boolean value $\mathsf{t} \in \mathsf{3}$, and MTL's connectives $\neg$ and $\vee$ with the corresponding three-valued operators in Table 1. Also note that when propositions are only assigned to Boolean values, i.e., $w \in TW^\omega(\Gamma)$ with $\Gamma := 2^P$ then the above definition coincides with MTL's standard two-valued semantics.

We use standard syntactic sugar, e.g., $\varphi \to \psi$ abbreviates $(\neg\varphi) \vee \psi$, and $\Diamond_I \varphi$ ("eventually") and $\Box_I \varphi$ ("always") abbreviate $\mathsf{t}\,\mathsf{U}_I\, \varphi$ and $\neg\Diamond_I \neg\varphi$, respectively. The past-time counterparts $\blacklozenge_I \varphi$ ("once") and $\blacksquare_I \varphi$ ("historically") are defined as expected. The nonmetric variants of the temporal connectives are also easily defined, e.g., $\Box\varphi := \Box_{[0,\infty)}\varphi$. Finally, we use standard conventions concerning the connectives' binding strength to omit parentheses.

▶ **Example 1.** The formula $\Box\, \mathit{req} \to \Diamond_{[0,100)}\, \mathit{ack}$ expresses a simple deadline property of a request-response protocol between two system components. Namely, requests must be acknowledged within 100 milliseconds, assuming that the unit of time is milliseconds.

3 Monitoring Architecture

The target system we monitor consists of one or more system components. The objective of monitoring is to determine at runtime whether the system's behavior, as observed and reported by the system components, satisfies a given MTL specification φ at some or all time points. To this end, we extend the system with additional components called *monitors*. The system components communicate with the monitors and the monitors communicate with each other. Communication takes place over channels. In the following, we explain the system assumptions, sketch the design of our monitoring extension, and state the monitors' requirements.

System Assumptions. We make the following assumptions on our system model.

A 1. *The system is static.*
This means that no system components are created or removed at runtime. Furthermore, each monitor is aware of the existence of all the system components. Note that this assumption can easily be eliminated by building into our algorithm a mechanism to register components before they become active and unsubscribing them when they become inactive. To register components we can, e.g., use a simple protocol where a component sends a registration request and waits until it receives a message that confirms the registration.

A 2. *Communication between components is asynchronous and unreliable. However, messages are neither tampered with nor delivered to wrong components.*
Asynchronous, unreliable communication means that messages may be received in an order different from which they were sent, and some messages may be lost and therefore never received. Note that message loss covers the case where a system component

crashes without recovery. A component that stops executing is indistinguishable to other processes from one that stops sending messages or none of its messages are received. We explain in Remark 7 in Section 4 that it is also straightforward to handle the case where crashed processes can recover. The assumption ruling out tampering and improper delivery can be discharged in practice by adding information to each message, such as a recipient identifier and a cryptographic hash value, which are checked when receiving the message.

A 3. *System components, including the monitors, are trustworthy.*
This means, in particular, that the components correctly report their observations and do not send bogus messages.

A 4. *Observations about a proposition's truth value are consistent.*
This means that no components observe that a proposition $p \in P$ is both true and false at a time $\tau \in \mathbb{Q}_+$.

A 5. *The system components make infinitely many observations in the limit.*
This guarantees that the observable system behavior is an infinite timed word. Note that MTL formulas specify properties about infinite timed words. We would need to use another language if we want to express properties about finite system behavior. However, note that a monitor is always aware of only a finite part of the observed system behavior. Furthermore, since channels are unreliable and messages can be lost, a monitor might even, in the limit, be aware only of a finite part of the infinite system behavior.

System Design. The monitors are organized in a directed acyclic graph structure, where each monitor is responsible for some subformula of the given MTL specification φ. The decomposition of φ into the subformulas used for monitoring is system and application specific. However, we require that it respects the subformula ordering in that if the monitor M' is in the subgraph of the monitor M, then the formula that M' monitors is a subformula of the one monitored by M. Moreover, the monitor at the root is responsible for φ. It outputs verdicts of the form $(b, \tau) \in \mathsf{2} \times \mathbb{Q}_+$, with the meaning that φ has the truth value b at time τ. We also add a unidirectional communication channel from each monitor to its parent monitors and unidirectional communication channels from the system components to the monitors. The system components are instrumented to send their observations to the monitors. This instrumentation is also system and application specific, and irrelevant for the functioning of the monitors; hence we do not discuss it further.

Three types of messages are exchanged during monitoring: report, notify, and alive.

- A system component sends the message $\mathsf{report}(p, b, \tau)$ when it observes at time $\tau \in \mathbb{Q}_+$ that the Boolean value $b \in \mathsf{2}$ is assigned to the proposition $p \in P$. This message is only sent to the monitors that are responsible for a subformula ψ of φ in which p occurs in one of ψ's subformulas for which no other monitor is responsible. Analogously, a monitor responsible for ψ sends messages of the form $\mathsf{report}(\psi, b, \tau)$ to inform its parent monitors about verdicts (b, τ) for the subformula ψ of φ.
- A system component C sends the message $\mathsf{notify}(C, \tau, s)$ to all monitors to inform them about some observation at time $\tau \in \mathbb{Q}_+$. The need to send such messages originates from MTL's *point-based* semantics. Their purpose is that all monitors are aware of all the time points and their timestamps of the timed word representing the observed system behavior. This message includes a sequence number $s \in \mathbb{N}$, which is the number of notify messages that C has sent so far, including the current one. A monitor uses s to determine whether it knows all time points up to time τ.

- A system component C can also send the message $\mathsf{alive}(C, \tau, s)$ when it has not made any observations for a while. The sequence number s is the number of messages of the form $\mathsf{notify}(C, \tau', s')$ with $\tau' < \tau$ that have been sent by C. The alive messages help a monitor to determine whether it has received all notify messages over some time period. In particular, alive messages are handy when components have not made any observations for a while.

▶ **Remark 2.** In what follows, we assume that there is only a single monitor. By this assumption, there are no messages of the form $\mathsf{report}(\psi, b, \tau)$, which are sent by a monitor responsible for the subformula ψ of φ. This assumption is without loss of generality since we can replace ψ in φ by a fresh proposition p_ψ and consider the submonitor as yet another system component. Note that this component need not send notify messages about the existence of time points since they are already sent by the other system components.

Monitor Requirements. Let O be the set of messages corresponding to the observations made by the system components and therefore, by A3, sent to (but not necessarily received by) the monitors. We use the timed word $w(O)$ to model the observable system behavior. It satisfies the following conditions.

(i) For every $\mathsf{notify}(C, \tau, s) \in O$, there is a letter (σ, τ) in $w(O)$.
(ii) For every $\mathsf{report}(p, b, \tau) \in O$, there is a letter (σ, τ) in $w(O)$ with $\sigma(p) = b$.
(iii) For every letter (σ, τ) in $w(O)$, there is some $\mathsf{notify}(C, \tau, s) \in O$ and for all $p \in P$, if $\sigma(p) \neq \bot$ then $\mathsf{report}(p, \sigma(p), \tau) \in O$.

Note that $w(O)$ is uniquely determined by the notify and report messages in O. First, for each $\tau \in \mathbb{Q}_+$, there is at most one letter with the timestamp τ in a timed word. Hence, all notify and report messages that include the timestamp τ determine the letter in $w(O)$ with this timestamp. The letter's position in $w(O)$ is also determined by τ. Second, because of A4, $\mathsf{report}(p, b, \tau) \in O$ implies $\mathsf{report}(p, \neg b, \tau) \notin O$. Finally, by A5, $w(O)$ is infinite.

We state the requirements of our monitoring approach concerning its correctness with respect to $w(O)$. The messages are processed iteratively by a monitor M for the formula φ and it keeps state between iterations. M's input in an iteration is a message and its output is a set $V \subseteq 2 \times \mathbb{Q}_+$ of verdicts. We denote M's output after processing a message m by $M(m)$. Let $\bar{m} = m_0, m_1, \ldots$ be a sequence of messages from O of length $N \in \mathbb{N} \cup \{\infty\}$.

- A monitor M is *sound* for φ on $\bar{m}$ if for all $\tau \in \mathbb{Q}_+$ and $b \in 2$, if $(b, \tau) \in M(m_i)$ for some $i < N$, then $[\![w(O) \models \varphi]\!]^\tau = b$.
- A monitor M is *complete* for φ on $\bar{m}$ if for all $\tau \in \mathbb{Q}_+$, and $b \in 2$, if $[\![w(O) \models \varphi]\!]^\tau = b$ then $(b, \tau) \in M(m_i)$, for some $i < N$.

▶ **Remark 3.** Completeness together with soundness is not achievable in general. One reason is failures, cf. A2. For instance, if all messages are lost, it is only possible in trivial cases for a monitor to soundly output verdicts for every violation. We therefore require completeness of a monitor only under the assumption that every message in O is eventually received by the monitor and the monitor never crashes. Another reason is that not all formulas are "monitorable" [17]. For example, the formula $\Box\Diamond p$, which states that p is true infinitely often, can only be checked on $w(O)$. However, a monitor only knows finite parts of $w(O)$ at any time, which is insufficient to determine whether the formula is fulfilled or violated. To simplify matters, we focus in the forthcoming sections on *bounded* formulas, i.e., the metric constraint of any temporal future-time connective is a finite interval. Note that if the formula ψ is bounded then $\Box\psi$ describes a safety property. Many deadline requirements have this form. Since we consider verdicts for all $\tau \in \mathbb{Q}_+$ with $[\![w(O) \models \psi]\!]^\tau \in 2$, the outermost temporal connective $\Box$ is implicitly handled by a monitor.

▶ **Example 4.** Consider a system with a single component C. Let O be an infinite set of messages containing the messages $\mathsf{notify}(C, 0.5, 1)$, $\mathsf{report}(p, \mathsf{f}, 0.5)$, $\mathsf{notify}(C, 2.0, 2)$, and $\mathsf{report}(p, \mathsf{f}, 2.0)$, and no other message with a timestamp less than or equal to 2.0. Note that the sequence number of the first notify message that C sends is 1 since C's sequence-number counter is incremented before C sends the message. Furthermore, assume that the message $\mathsf{report}(p, \mathsf{f}, 0.5)$ is lost, while all other messages are received by the monitor. A sound monitor for the formula $\blacklozenge_{[0,1]}\, p$ can at most output the verdicts $(\mathsf{f}, 0.5)$ and $(\mathsf{f}, 2.0)$ for the time points 0 and 1, respectively. However, since a monitor does not know p's truth value at time 0.5, it cannot deduce the verdict $(\mathsf{f}, 0.5)$, and is therefore incomplete. Note that a monitor can deduce the verdict $(\mathsf{f}, 2.0)$ because, from the sequence numbers of the notify messages, it can infer that there is no other time point originating from the component C in $w(O)$ with a timestamp between 1.0 and 2.0.

4 Verdict Computation

In this section, we explain how a monitor processes a sequence of messages from the set O of messages sent by the system components and how it computes verdicts.

Main Loop. The monitor's main procedure Monitor, given in Figure 1, is invoked for each message received. It takes as input φ, the formula to be monitored, and a message. It updates the monitor's state, thereby computing verdicts, which it returns. The verdicts computed in an iteration of the monitor are stored in the global variable verdicts, which is set to the empty set at the start of processing the received message.

```
procedure Monitor(φ, msg)
  verdicts ← ∅
  case msg = notify(_, τ, _)
    NewTimePoint(φ, τ)
  case msg = report(p, b, τ)
    NewTimePoint(φ, τ)
    SetTruthValue((p, {τ}), b)
  foreach J in NewCompleteIntervals(msg) do
    NoTimePoint(φ, J)
  return verdicts
```

Figure 1 The monitor's main loop.

Intuitively speaking, with each received message the monitor gains knowledge about the infinite timed word $w(O)$. The monitor's partial knowledge about $w(O)$ is reflected in the monitor's state. The monitor's state is maintained by the procedures NewTimePoint, SetTruthValue, and NoTimePoint. When a $\mathsf{notify}(C, \tau, s)$ message is received, Monitor calls the NewTimePoint procedure, which makes the monitor aware of the existence of the time point with the timestamp τ in $w(O)$. When a $\mathsf{report}(p, \tau, b)$ message is received, Monitor calls the SetTruthValue procedure, which sets the proposition p's truth value at the time point with timestamp τ to the Boolean value b. It also deduces, whenever possible, the truth values of φ's subformulas at the known time points in $w(O)$. This deduction can result in new verdicts. Note that, prior to SetTruthValue, Monitor calls the NewTimePoint procedure, which ensures that the monitor is aware of the existence of the time point with timestamp τ in $w(O)$. Finally, Monitor accounts for the intervals that became complete by the received message. We say that an interval $J \subseteq \mathbb{Q}_+$ is *complete* if the monitor has received all notify messages with a timestamp in J from all system components. In particular, if J is incomplete, then the monitor does not yet know all the timestamps in J from letters in $w(O)$. The procedure NewCompleteIntervals returns new complete intervals, based on the sequence number of the received message and the monitor's state. Note that only notify and alive messages contain a sequence number; for a report message, NewCompleteIntervals does not return any intervals. For each of the returned intervals, the NoTimePoint procedure updates the monitor's state accordingly.

In the following, we provide some details about the monitor's state and how it is updated. We start by explaining the main data structure stored in the monitor's state.

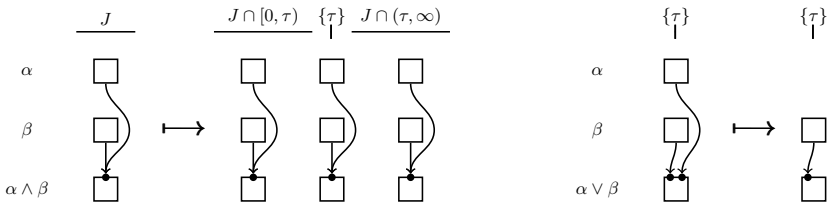
Data Structure. The main data structure is a graph structure. Its *nodes* are pairs of the form (ψ, J), where ψ is a subformula of the monitored formula φ and $J \subseteq \mathbb{Q}_+$ is an interval. The interval J is either a singleton $\{\tau\}$, where τ is the timestamp occurring in a received notify or report message (thus τ occurs in a letter in $w(O)$), or it is an incomplete interval. Initially, there is a node $(\psi, [0, \infty))$ for each subformula ψ of φ. The interval $[0, \infty)$ corresponds to the fact that no time points have been created yet. Each node is associated with a truth value, initially $\bot$. Furthermore, each node contains a set of *guards* and a set of outgoing pointers to guards, called *triggers*. We call the source node of an incoming pointer to a guard a *precondition*. Intuitively, a guard with no preconditions (i.e., no incoming pointers) is satisfied and we assign the Boolean value t to the guard's node; if a node has no guards, we assign the Boolean value f to the node; otherwise, if a node has guards with incoming pointers, the node is assigned the truth value $\bot$. Overall, the graph structure can be viewed as an AND-OR-graph, where intuitively a node's truth value is given by the disjunction over the node's guards of conjunctions of the truth values of each guard's preconditions.

Updates. The first time the monitor receives a notify or a report message with some timestamp τ, a new time point in $w(O)$ is identified and the data structure is updated. Note that the timestamp τ is necessarily in some incomplete interval J. Each node (ψ, J) in the graph is replaced by the nodes $(\psi, \{\tau\})$, $(\psi, J \cap [0, \tau))$, and $(\psi, J \cap (\tau, \infty))$. The links of the new nodes to and from the other nodes are created based on the links of the node (ψ, J). Links are used to propagate Boolean values from one node to another when, for example, receiving a report message. These two tasks, creating nodes and propagating truth values, are carried out by the procedures NewTimePoint and SetTruthValue, respectively. NewTimePoint also deletes a node (ψ, J) after creating the new nodes for the split interval J, and SetTruthValue deletes nodes when they are no longer needed for propagating truth values. A call to SetTruthValue$((\varphi, \{\tau\}), b)$, for some timestamp τ and Boolean value b, also adds the verdict (b, τ) to the set verdicts.

When the monitor infers that a nonsingular interval J is complete, it calls the procedure NoTimePoint, which deletes the nodes of the form (ψ, J) and updates triggers if necessary. Moreover, it calls the procedure SetTruthValue when a Boolean value can be assigned to a node. The monitor uses the sequence numbers in notify and alive messages to determine whether there are no time points with timestamps in J. For such a J, the monitor must have received from each component C messages of one of the forms: (1) notify(C, τ, s) with $\tau \leq \inf J$ and either notify$(C, \tau', s+1)$ or alive(C, τ', s) with $\tau' \geq \sup J$, or (2) alive(C, τ, s) with $\tau \leq \inf J$ and either notify$(C, \tau', s+1)$ or alive(C, τ', s) with $\tau' \geq \sup J$. In the latter case, we assume without loss of generality that the monitor has received at the beginning the message alive$(C, -1.0, 0)$ from each component C.

In the following, we explain how nodes are created and Boolean values are propagated. For a newly created node (ψ, J), its guards and their preconditions depend on ψ's main connective. We first focus on the simpler cases where the main connective is nontemporal.

A node for the formula $\psi = \alpha \vee \beta$ has two guards, each with a precondition for ψ's direct subformulas. Analogously, when considering $\wedge$ as a primitive, the node for $\psi = \alpha \wedge \beta$ has one guard with two preconditions for the two direct subformulas. A node for the formula $\psi = \neg\alpha$ has one guard with the node for the formula α as the precondition associated to the same time point or an incomplete interval. The first two cases are illustrated on the



(a) node creation for a new time point (b) propagation

Figure 2 Adjusting guards in case of (a) the creation of a new time point at $\tau \in J$ for the formula $\alpha \wedge \beta$, and (b) propagation, namely when the node $(\alpha, \{\tau\})$ is set to f, for the formula $\alpha \vee \beta$.

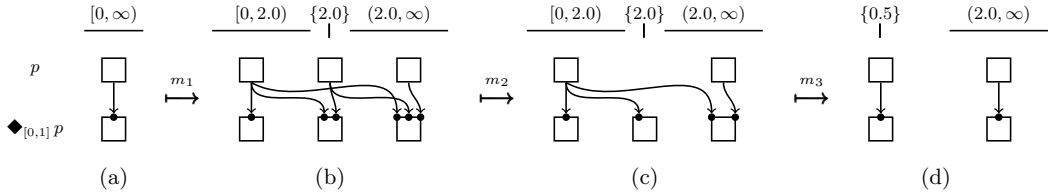


Figure 3 The data structure (a) before receiving any message, and after receiving the messages (b) $m_1 = \mathsf{notify}(C, 2.0, 2)$, (c) $m_2 = \mathsf{report}(p, \mathsf{f}, 2.0)$, and (d) $m_3 = \mathsf{notify}(C, 0.5, 1)$.

left-hand side of the arrow $\mapsto$ of Figure 2(a) and (b), respectively. A box corresponds to a node, where the node's formula is given by the row and the interval by the column of the box. Dots correspond to guards and arrows to triggers. Figure 2(a) also illustrates how the data structure is updated when a new time point is added; in the case of Boolean connectives, this is done by simply duplicating the nodes and their guards and triggers. The creation of the guards of a node for a formula with the main connective S_I or U_I is more complex as the preconditions are nodes that can be associated to time points or incomplete intervals different from the node's interval J. We first sketch how Boolean values are propagated before explaining these more complex cases.

When receiving a $\mathsf{report}(p, b, \tau)$ message, we set the truth value of the node $(p, \{\tau\})$ to the Boolean value b, provided that the node exists. This value is then propagated through the node's triggers to its successor nodes. However, for negation, the Boolean value propagated from a node (α, J) to the node $(\neg\alpha, J)$ is the complement of the Boolean value associated to the node (α, J). The propagation of the Boolean value t corresponds to deleting just the triggers, whereas the propagation of f corresponds to also deleting the guards that the triggers point to. If a guard of a successor node has no more preconditions, then we set the successor's nodes value to t; in contrast, if the set of guards of a successor node becomes empty, then we set its value to f. Figure 2(b) illustrates the propagation of a truth value through the data structure for the simple case where the formula is of the form $\alpha \vee \beta$.

Temporal Connectives. Before describing the general case of handling formulas ψ of the forms $\alpha\, \mathsf{S}_I\, \beta$ and $\alpha\, \mathsf{U}_I\, \beta$, we consider a simpler example where $\psi = \blacklozenge_I\, p$. In this case, a node $(\blacklozenge_I\, \psi, J)$ has a guard for every node (ψ, K) for which there are a $\tau \in J$ and $\kappa \in K$ such that $\tau - \kappa \in I$. Each guard has exactly one precondition, namely the corresponding node (ψ, K).

▶ **Example 5.** We reconsider the formula $\varphi = \blacklozenge_{[0,1]}\, p$ from Example 4, and a set O that contains the messages $m_1 := \mathsf{notify}(C, 2.0, 2)$, $m_2 := \mathsf{report}(p, \mathsf{f}, 2.0)$, and $m_3 := \mathsf{notify}(C, 0.5, 1)$. We assume that the monitor receives m_1, m_2, and m_3 in this order. Figure 3 illustrates how

the data structure is updated after receiving each of these messages. The updates performed after the first two messages are clear from the previous explanations, whereas the update performed for the message m_3 comprises the following update steps. First, the interval $[0, 2.0)$ is split at timestamp 0.5, thus deleting the nodes $(p, [0, 2.0))$ and $(\varphi, [0, 2.0))$, and creating six new nodes, together with their guards and triggers. Namely, a node is created for each of the formulas p and φ, and for each of the intervals $[0, 0.5)$, $\{0.5\}$, and $(0.5, 2.0)$. The preconditions of the other nodes are also updated accordingly to the intervals of the newly created nodes. Second, the four nodes corresponding to the intervals $[0, 0.5)$ and $(0.5, 2.0)$, together with their triggers, are deleted. This is because these intervals are complete, that is, no time point of $w(O)$ has a timestamp in these intervals. By deleting these four nodes, the node $(\varphi, \{2.0\})$ remains with no guards. Indeed, after the split, the node remains with only one guard, which has the precondition $(p, (0.5, 2.0))$. The other nodes (p, J) are not preconditions because no timestamp in those intervals J satisfies the temporal constraint. This single guard is deleted when its precondition $(p, (0.5, 2.0))$ is deleted. Finally, as the node $(\varphi, \{2.0\})$ is without guards, it is assigned the Boolean value f. These steps lead to the structure in Figure 3(d).

We now consider the general case where the formula ψ is $\alpha \,\mathsf{S}_I\, \beta$; the case for $\alpha \,\mathsf{U}_I\, \beta$ is dual. The node (ψ, J) has a guard for each node (β, K) with the truth value t or $\bot$, and with $(J \ominus K) \cap I \neq \emptyset$, where $J \ominus K := \{\tau - \kappa \mid \tau \in J \text{ and } \kappa \in K\}$. Moreover, for each such node (β, K) and all nodes (α, H), with H between K and J, we have that (α, H) is not assigned to the truth value f. We call the node (β, K) an *anchor node* for the node (ψ, J), and a node (α, H) a *continuation node* for the anchor (β, K). The node (α, H) must be strictly after (β, K) if K is a singleton, but we can have $H = K$ otherwise. Note too that a node can be a continuation node for multiple anchor nodes. A guard has a trigger from its anchor node if the truth value assigned to the anchor node is $\bot$. Furthermore, the guard has a trigger from the first continuation node after the anchor node that is assigned to the truth value $\bot$. If this continuation node is assigned to the Boolean value t at a later time, we move the trigger to the second such continuation node, and delete it if such a node does not exist. Alternatively, we could unroll ψ into a disjunction of conjunctions and use the guard constructions presented previously for $\wedge$ and $\vee$. However, each guard would have multiple continuation nodes as preconditions, which would result in an unnecessary overhead.

When splitting the interval J at time τ, we create the new nodes (ψ, J'), with $J' \in \{J \cap [0, \tau), \{\tau\}, J \cap (\tau, \infty)\}$, together with the nodes' guards. For this, we use the guards of the node (ψ, J). After creating the new nodes, we delete the node (ψ, J). The split preserves the invariant, stated in the previous paragraph, about the nodes' guards. This invariant is key in the algorithm's soundness proof. We illustrate the construction for the specific case depicted on the left-hand side of Figure 4. There are two guards of $(\alpha \,\mathsf{S}_I\, \beta, J)$, each with two preconditions. The first guard has the anchor node (β, K) and the continuation node (α, H). The second guard has the anchor node (β, J) and the continuation node (α, J). The triggers of the first guard are drawn with solid lines in Figure 4 and the triggers of the second guard are drawn with dashed lines. We assume that J, K, and H are pairwise disjoint. We also assume that $0 \in I$ and that the metric constraint is satisfied for the new nodes and their anchors. The right-hand side of Figure 4 shows the guards for the new nodes along with their triggers.

Initialization. The monitor's state is initialized by the procedure Initialize, which takes φ as argument. We assume that it is called before processing the received messages by the Monitor procedure. Initially, the nodes of the graph structure are $(\psi, [0, \infty))$, where ψ is a subformula

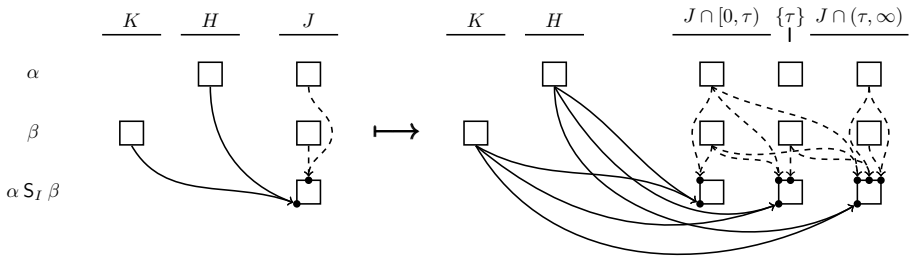


Figure 4 Adjusting guards when creating a new time point at $\tau \in J$ for the formula $\alpha \, \mathsf{S}_I \, \beta$.

of φ, with the corresponding guards and triggers. The truth value of a node $(\psi, [0, \infty))$ is $\bot$, except for the node $(\mathsf{t}, [0, \infty))$, which has the truth value t. Note that the node $(\mathsf{t}, [0, \infty))$ only exists if the constant t occurs in φ. For this node, we invoke the procedure SetTruthValue to propagate its Boolean value.

Correctness Guarantees. The correctness guarantees of the monitoring algorithm are given in the following theorem.

▶ **Theorem 6.** *Let $\bar{m} = m_0, m_1, \ldots$ be the sequence of messages in O received by the monitor.*

(i) *The monitor is sound for φ on $\bar{m}$.*

(ii) *The monitor is complete for φ on $\bar{m}$, if (a) all temporal future connectives in φ are bounded (i.e., their metric constraints are finite intervals), and (b) for every $m \in O$, there is some $i \in \mathbb{N}$ with $m_i = m$.*

▶ **Remark 7.** When a process crashes, its state is lost. To recover a process we must bring it into a state that is safe for the system. To safely restart a system component, we must restore its sequence number. We can use any persistent storage available to store this number. In case the component crashes while storing this number, we can increment the restored number by one. This might results in knowledge gaps for some monitors, since some intervals will never be identified as complete. However, the computed verdicts are still sound.

For the recovery of a crashed monitor, we just need to initialize it. In particular, the nodes of its graph structure are of the form $(\psi, [0, \infty))$, where ψ is a subformula of φ. A recovered monitor corresponds to a monitor that has not yet received any messages. This is safe in the sense that the recovered monitor will only output sound verdicts. When the monitor also logs received messages in a persistent storage, it can replay them to close some of its knowledge gaps. Note that the order in which these messages are replayed is irrelevant and they can even be replayed whenever the recovered monitor is idle.

5 Accuracy of Timestamps

The monitors' verdicts are computed with respect to the observations that the monitors receive from the system components. These observations might not match with the actual system behavior. In particular, the timestamp in a message $\mathsf{report}(p, b, \tau)$ may be inaccurate because τ comes from the clock of a system component that has drifted from the actual time. Nevertheless, we use these timestamps to determine the time between observations. Hence, one may wonder in what sense are the verdicts meaningful.

Consider first the guarantees we have under the additional system assumption that timestamps are precise and from the domain $\mathbb{Q}_+$. Under this assumption, $w(O) \preceq w$, where $w(O) \in TW^\omega(\Sigma)$ is the observed system behavior and $w \in TW^\omega(\Gamma)$ represents the real system behavior. Note that in w, all propositions at all time points are assigned Boolean values, which might not be the case in $w(O)$ since no system component observes whether a proposition is true or false at a time point. It follows from Lemma 8 that the verdicts computed from the observed system behavior $w(O)$ are also valid for the system behavior w.

▶ **Lemma 8.** *Let φ be an MTL formula, $v, v' \in TW^\omega(\Sigma)$, and $\tau \in \mathbb{Q}_+$. If $v \preceq v'$ then $[\![v \models \varphi]\!]^\tau \preceq [\![v' \models \varphi]\!]^\tau$.*

Assuming precise timestamps is however a strong assumption, which does not hold in practice since real clocks are imprecise. Moreover, each system component uses its local clock to timestamp observations and these clocks might differ due to clock drifts. In fact, assuming synchronized clocks boils down to having a synchronized system at hand.

Nevertheless, we argue that for many kinds of policies and systems, relying on timestamps from existing clocks in monitoring is good enough in practice. First, under stable conditions (like temperature), state-of-the-art hardware clocks already achieve a high accuracy and their drifts are, even over a longer time period, rather small [7]. Moreover, there are protocols like the Network Time Protocol (NTP) [16] for synchronizing clocks in distributed systems that work well in practice. For local area networks, NTP can maintain synchronization of clocks within one millisecond [14]. Overall, with state-of-the-art techniques, we can obtain timestamps that are "accurate enough" for many monitoring applications, for instance, for checking whether deadlines are met when the deadlines are in the order of seconds or even milliseconds. Furthermore, if the monitored system guarantees an upper bound on the imprecision of timestamps, we can often account for this imprecision in the policy formalization. For example, if the policy stipulates that requests must be acknowledged within 100 milliseconds and the imprecision between two clocks is always less than a millisecond, then we can use the MTL formula $\Box\, \mathit{req} \rightarrow \blacklozenge_{[0,1)} \Diamond_{[0,101)}\, \mathit{ack}$ to avoid false alarms.

6 Related Work

Multi-valued semantics for temporal logics are widely used in monitoring, see e.g., [5, 4, 3, 18, 15]. Their semantics extend the classical LTL semantics by also assigning non-Boolean truth values to finite prefixes of infinite words. The additional truth values differentiate whether some or all extensions of a finite word satisfy a formula. However, in contrast to the three-valued semantics of MTL used in this paper, the Boolean and temporal connectives are not extended over the additional truth values. Furthermore, the partial order $\prec$ on the truth values, which orders them in knowledge, is not considered. Note that having the third truth value $\bot$ at the logic's object level and the partial order $\prec$ is at the core of our monitoring approach, namely it is used account for a monitor's knowledge gaps. Multi-valued semantics for temporal logics have also been considered in other areas of system verification. For example, Chechik et al. [6] describe a model-checking approach for a multi-valued extension for the branching-time temporal logic CTL. Their CTL extension is similar to our extension of MTL in the sense that it allows one to reason about uncertainty at the logic's object level. However, the considered tasks are different. Namely, in model checking, the system model is given—usually finite-state—and correctness is checked offline with respect to the model's described executions; in contrast, in runtime verification, one checks online the correctness of the observed system behavior.

Several monitoring algorithms have been developed for verifying distributed systems at runtime [19, 18, 3, 8, 15]. They make different assumptions on the system model and thus target different kinds of distributed systems. Furthermore, they handle different specification languages. None of them account for network failures or handle specifications with real-time constraints. Sen et al. [19] use an LTL variant with epistemic operators to express distributed knowledge. The verdicts output by the monitors are correct with respect to the local knowledge the monitors obtained about the systems' behavior. Since their LTL variant only comprises temporal connectives that refer to the past, only safety properties are expressible. Scheffel and Schmitz [18] extend this work to also handle some liveness properties by working with a richer fragment of LTL that includes temporal connectives that also refer to the future. The algorithm by Bauer and Falcone [3] assumes a lock-step semantics and thus only applies to synchronous systems. Falcone et al. [8] weaken this assumption. However, each component must still output its observations at each time point, which is determined by a global clock. The observations are then received by the monitors at possibly later time points. The algorithm by Mostafa and Bonakdarbour [15] assumes lossless FIFO channels for asynchronous communication. Logical clocks are used to partially order messages.

Various monitoring algorithms have been developed, analyzed, and used to verify real-time constraints at runtime, see e.g., [20, 5, 13, 2]. All of them, however, fall short for monitoring distributed systems. For instance, they do not account for out-of-order message deliveries and the monitor's resulting knowledge gaps about the observed system behavior. It is this shortcoming of prior work that motivated us to develop the monitoring algorithm presented in this paper.

7 Conclusion

We have presented a monitoring algorithm for verifying the behavior of a distributed system at runtime, where properties are specified in the real-time logic MTL. Our algorithm accounts for failures and out-of-order message deliveries. The monitors' verdicts are sound with respect to the observed system behavior. In particular, timestamps originating from local clocks determine the time between the observations made by the system components and sent to the monitors. Note that the ground truth for system behavior is not accessible to the monitors because the monitors themselves are system components.

There are several directions for extending our work. First, we have considered a monitor's completeness from a global perspective, i.e., the observable system behavior. An alternative would be with respect to the knowledge a monitor can infer from the messages it receives. We intend to investigate when a monitor is complete under this perspective. Second, we have opted for a point-based semantics for MTL. An alternative is to use an interval-based semantics, which can be more natural but it also makes monitoring more complex, see [2]. Future work is to adapt the presented monitoring algorithm to an interval-based semantics. Finally, we plan to evaluate our monitoring algorithm on a substantial case study.

Acknowledgments. We thank Srdjan Marinovic for his input and for many helpful discussions in the early phase of this work.

References

1 R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Proceedings of the 1991 REX Workshop on Real Time: Theory in Practice*, volume 600 of *Lect. Notes Comput. Sci.*, pages 74–106. Springer, 1992.

2 D. Basin, F. Klaedtke, and E. Zălinescu. Algorithms for monitoring real-time properties. In *Proceedings of the 2nd International Conference on Runtime Verification (RV)*, volume 7186 of *Lect. Notes Comput. Sci.*, pages 260–275. Springer, 2011.

3 A. Bauer and Y. Falcone. Decentralised LTL monitoring. In *Proceedings of the 18th International Symposium on Formal Methods (FM)*, volume 7436 of *Lect. Notes Comput. Sci.*, pages 85–100. Springer, 2012.

4 A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *J. Logic Comput.*, 20(3):651–674, 2010.

5 A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Meth.*, 20(4):14, 2011.

6 M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. Multi-valued symbolic model-checking. *ACM Trans. Softw. Eng. Meth.*, 12(4), 2003.

7 F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, 1999.

8 Y. Falcone, T. Cornebize, and J.-C. Fernandez. Efficient and generalized decentralized monitoring of regular languages. In *Proceedings of the 34th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, volume 8461 of *Lect. Notes Comput. Sci.*, pages 66–83. Springer, 2014.

9 M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

10 S. C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, Princeton, 1950.

11 R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, 1990.

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

13 O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Proceedings of the Joint International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS) and on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, volume 3253 of *Lect. Notes Comput. Sci.*, pages 152–166. Springer, 2004.

14 D. L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Trans. Netw.*, 3(3):245–254, 1995.

15 M. Mostafa and B. Bonakdarbour. Decentralized runtime verification of LTL specifications in distributed systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 494–503. IEEE Computer Society, 2015.

16 Network time protocol. `www.ntp.org`, webpage accessed on March 26, 2015.

17 A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In *Proceedings of the 14th International Symposium on Formal Methods (FM)*, volume 4085 of *Lect. Notes Comput. Sci.*, pages 573–586. Springer, 2008.

18 T. Scheffel and M. Schmitz. Three-valued asynchronous distributed runtime verification. In *Proceedings of the 12th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMCODE)*, pages 52–61. IEEE Computer Society, 2014.

19 K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 418–427. IEEE Computer Society, 2004.

20 P. Thati and G. Roşu. Monitoring algorithms for metric temporal logic specifications. In *Proceedings of the 4th Workshop on Runtime Verification (RV)*, volume 113 of *Elec. Notes Theo. Comput. Sci.*, pages 145–162. Elsevier Science Inc., 2005.