Failure-aware Runtime Verification of Distributed Systems

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

1 ETH Zurich, Department of Computer Science, Zurich
2 NEC Labs Europe, Heidelberg

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.xxx.yyy.p

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 [3, 8, 15, 19]. 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 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 [1, 11] 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 $\perp$, 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 $\perp$, indicating that the monitor has a knowledge gap about the system behavior. The truth value $\perp$ 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 $\perp$.

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 $\perp$ 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.

The remainder of this paper is structured as follows. In Section 2, we introduce the

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	$\bot$	$\bot$

real-time logic MTL with a three-valued semantics. In Section 3, we describe the system assumptions and the requirements. 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. Additional details, omitted in the main text, are given in Appendices A and B.

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 (i) 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 $0 \le i < |v|$ 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 $w = (\sigma_0, \tau_0)(\sigma_1, \tau_1)\dots$.

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

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.

A1 *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.

A2 *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.

A3 *System components, including the monitors, are trustworthy.*
This means, in particular, that the components correctly report their observations and do not send bogus messages.

A4 *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 $\tau \in \mathbb{Q}_+$.

A5 *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 $(b, \tau) \in 2 \times \mathbb{Q}_+$, 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 $b \in 2$ 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. See Appendix A.5 for details.

– 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.

```
procedure Monitor($\varphi$, msg)
  verdicts $\leftarrow \emptyset$
  case msg = notify(_, $\tau$, _)
    NewTimePoint($\varphi$, $\tau$)
  case msg = report($p, b, \tau$)
    NewTimePoint($\varphi$, $\tau$)
    SetTruthValue(($p$, $\{\tau\}$), $b$)
  foreach $J$ in NewCompleteIntervals(msg) do
    NoTimePoint($\varphi$, $J$)
  return verdicts
```

Figure 1 The monitor's main loop.

▶ **Example 4.** Consider a system with a single component C. Let O be an infinite set of messages containing the messages notify$(C, 0.5, 1)$, report$(p, \mathsf{f}, 0.5)$, notify$(C, 2.0, 2)$, and 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 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.

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 notify(C, τ, 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 report(p, τ, 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. However, we only sketch the procedures NewTimePoint, SetTruthValue, and NoTimePoint here and refer to Appendix A for further algorithmic details. 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 $\perp$. 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 $\perp$. 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.

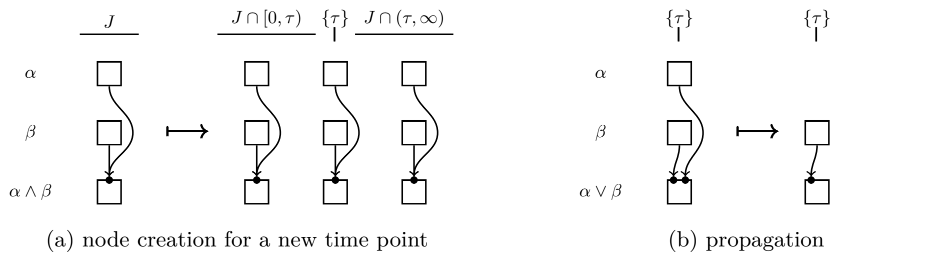

Figure 2 Adjusting guards in case of (a) the creation of a new time point at $\tau \in J$ for the formula $\alpha \wedge \beta$, and (b) propagation, namely when the node $(\alpha, \{\tau\})$ is set to f, for the formula $\alpha \vee \beta$.

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 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 $[0, 2.0)$

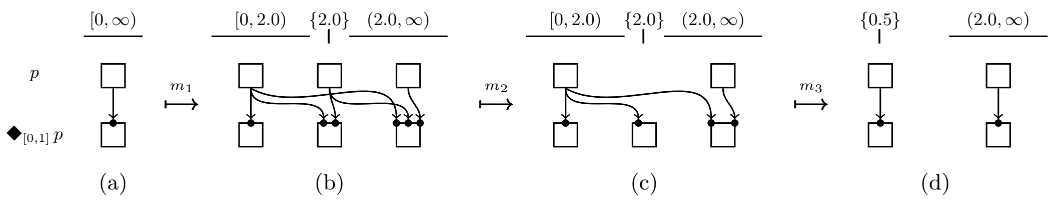

Figure 3 The data structure (a) before receiving any message, and after receiving the messages (b) $m_1 = \mathsf{notify}(C, 2.0, 2)$, (c) $m_2 = \mathsf{report}(p, \mathsf{f}, 2.0)$, and (d) $m_3 = \mathsf{notify}(C, 0.5, 1)$.

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 $\perp$, 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 $\perp$. Furthermore, the guard has a trigger from the first continuation node after the anchor node that is assigned to the truth value $\perp$. 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. See Appendix B.1. 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

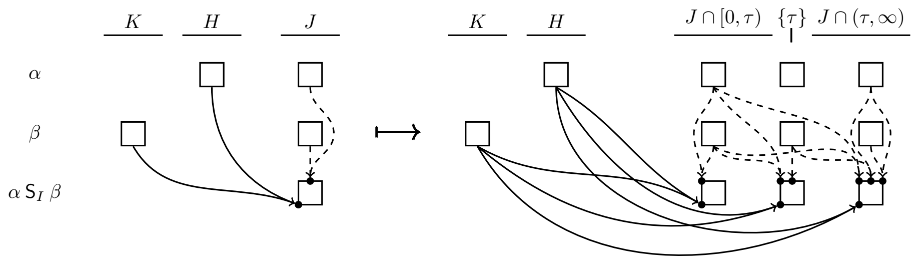

Figure 4 Adjusting guards when creating a new time point at $\tau \in J$ for the formula $\alpha \,\mathsf{S}_I\, \beta$.

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 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. We refer to Appendix B for proof details.

▶ **Theorem 6.** *Let $\bar{m} = m_0, m_1, \dots$ 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 $m_i = m$.

We remark that if the formula φ contains unbounded temporal future connectives the monitor may still output verdicts and these verdicts are sound by Theorem 6(i). Note that completeness together with soundness for all MTL specifications cannot be achieved in general; see Remark 3.

▶ **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 $\square\, \mathit{req} \rightarrow \blacklozenge_{[0,1)} \lozenge_{[0,101)} \mathit{ack}$ to avoid false alarms.

6 Related Work

Multi-valued semantics for temporal logics are widely used in monitoring, see e.g., [3–5,15,18]. 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 $\perp$ 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 [3, 8, 15, 18, 19]. 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., [2,5,13,20]. 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.

Table 2 Auxiliary procedures.

Procedure	Description / Return Value
Initialize(φ)	initializes the graph structure for the formula φ
ExistsTimePoint(τ)	returns true if the time point with timestamp τ exists
GetInterval(τ)	returns the interval J with $\tau \in J$
ExistsNode(ψ, J)	returns true if the node (ψ, J) exists
GetParent(ψ)	returns the parent formula of ψ (nil if $\psi = \varphi$)
GetDirectSubformulas(ψ)	returns the set of the direct subformulas of ψ
CreateNode(ψ, J)	creates a node with value $\perp$ and no guards and triggers
InsertBefore(ψ, n, n')	inserts the node n before the node n' in ψ's list of nodes
Split(J, τ)	returns the pair $\big(J \cap [0,\tau), J \cap (\tau, \infty)\big)$
IsBefore(L, K)	returns true if $L \neq K$ and $(K \ominus L) \cap \mathbb{Q}_+ \neq \emptyset$
IsSingleton(J)	returns true if J is a singleton
IsConstrSatisfiable(∇_I, L, K)	returns true if $\ominus_\nabla(L, K) \cap I \neq \emptyset$
IsConstrValid(∇_I, L, K)	returns true if $\ominus_\nabla(L, K) \subseteq I$
IsNode(prec)	returns true if the precondition prec represents a node
GetConstrPrec(grd)	returns the temporal constraint precondition of grd
HasAnchor(grd)	returns true if the guard grd has an anchor
GetAnchor(grd)	returns the anchor of the guard grd (nil if anchor does not exist)
HasCont(grd)	returns true if the guard grd has a continuation
GetCont(grd)	returns the continuation of the guard grd (nil if continuation does not exist)
Report(ψ, b, τ)	reports that the node $(\psi, \{\tau\})$ has value b (i.e., it updates the global variable verdicts)

Table 3 Record types and their fields.

Record Type	Field Name	Field Type
node	Formula	formula
	Interval	interval
	Value	truth value (in 3)
	Guards	set of guards
	Triggers	set of triggers
	Next	node
	Prev	node
guard	GuardOf	node
	Precs	set of preconditions

A Additional Algorithmic Details

Recall the monitor's main loop and its pseudocode, which is shown in Figure 1. In the following, we present additional algorithmic details of our monitoring algorithm. Namely, we present the procedures NewTimePoint (Section A.2), SetTruthValue (Section A.3), and NoTimePoint (Section A.4) in detail. Finally, we explain how the monitor infers whether an interval is complete (Appendix A.5), i.e., we provide details about the procedure NewCompleteIntervals. We start with details on the monitor's data structure (Section A.1).

Throughout this section, we use the following conventions. Variable names start with a lower-case letter, while procedure names start with an upper-case letter. Furthermore, ∇ ranges over the temporal connectives, i.e., $\nabla \in \{\mathsf{S}, \mathsf{U}\}$. Table 2 summarizes the auxiliary procedures that we use in the following and for which we do not provide pseudocode.

A.1 Data Structure

We represent nodes and guards using records. Their fields are given in Table 3. We initialize a record variable r using the notation r $\leftarrow$ {Field=value, …}, and we use a dot to access a

```
procedure NewTimePoint($\psi$, $\tau$)
  if not ExistsTimePoint($\tau$) then
    $J \leftarrow$ GetInterval($\tau$)
    $(J_1, J_2) \leftarrow$ Split($J$, $\tau$)
    AddNodes($\psi$, $J$, $J_1$, $J_2$, $\tau$)
    if $(\psi, J)$.Value $= \bot$ then
      for each $\alpha$ in GetDirectSubformulas($\psi$) do
        NewTimePoint($\alpha$, $\tau$)
      AddGuards($\psi$, $J$, $J_1$, $J_2$, $\tau$)
    $\gamma \leftarrow$ GetParent($\psi$)
    case $\gamma = \_\nabla_I \psi$
      AddTriggersForAnchor($\gamma$, $J$, $\tau$, $J_1$, $J_2$)
    case $\gamma = \psi \nabla_I \_$
      AddTriggersForCont($\gamma$, $J$, $J_1$)
    if $(\psi, J)$.Value $\neq \bot$ then
      foreach $J'$ in $\{J_1, \{\tau\}, J_2\}$ do
        SetTruthValue($(\psi, J')$, $(\psi, J)$.Value)
    DeleteNode($(\psi, J)$)
```

Figure 5 The procedure NewTimePoint.

record's field, as in r.Field.

As in Section 4, we identify a node with a tuple consisting of a formula and an interval. That is, a variable n of the node type is written as the tuple (n.Formula, n.Interval). Note that by maintaining a map from tuples of formulas and intervals to nodes, we can access a node in constant time.

In addition to linking nodes via triggers, we store the nodes for each subformula ψ of φ in a doubly-linked list. These lists are sorted ascendingly according to the nodes' intervals. Recall that different nodes for the same subformula are associated with disjoint intervals, so we have a total order. The lists are used to access a node's successors and predecessors, e.g., when updating continuation nodes once the truth value of the current continuation node becomes true: in general, the new continuation node is the next or previous node in the list, depending on whether the main connective of the parent subformula is S or U, respectively.

For a uniform treatment of the propagation of truth values, we refine preconditions to also be temporal constraints, not just nodes, as in Section 4. Formally, we call a *temporal constraint* a triple of the form (∇_I, L, K) and we say that it is *satisfiable* if there are timestamps $\tau \in L$ and $\tau' \in K$ such that (1) $\tau - \tau' \in I$ if $\nabla = \mathsf{S}$, and (2) $\tau' - \tau \in I$ if $\nabla = \mathsf{U}$. The constraint is *valid* when the conditions hold for each $\tau \in L$ and each $\tau' \in K$. These two conditions are equivalent with $\ominus_\nabla(L, K) \cap I \neq \emptyset$ and respectively $\ominus_\nabla(L, K) \subseteq I$, where $\ominus_\mathsf{S}(L, K) := (L \ominus K)$ and $\ominus_\mathsf{U}(L, K) := (K \ominus L)$. Also, note that if (∇_I, L, K) is valid then (∇_I, L', K') is valid for any $L' \subseteq L$ and $K' \subseteq K$. We extend these satisfiability notions to nodes. We say that a node n is satisfiable if n.Value $= \bot$, it is valid if n.Value $=$ t, and it is unsatisfiable if n.Value $=$ f.

A.2 The NewTimePoint Procedure

The procedure NewTimePoint (Figure 5) is called whenever the monitor is informed about the existence of a time point. It first checks whether the time point is new for the monitor. If this is not the case, nothing needs to be done. If the time point is new to the monitor, the given timestamp τ must be in an incomplete interval J. Note that the monitor's knowledge about the complete intervals is updated later in the procedure Monitor. For each subformula ψ of φ, there is a node (ψ, J) if $\psi = \varphi$ or if for the parent formula γ of ψ the value associated with the node (γ, J) is $\bot$. For each such a formula ψ, the procedure NewTimePoint(ψ, τ) builds the nodes (ψ, J_1) and (ψ, J_2) for the two new intervals $J_1 = J \cap [0, \tau)$ and $J_2 = J \cap (\tau, \infty)$, and the node $(\psi, \{\tau\})$ for the new time point. The guards and triggers of these new nodes

```
procedure AddNodes($\psi$, $J$, $J_1$, $J_2$, $\tau$)
  CreateNode($\psi$, $J_1$)
  CreateNode($\psi$, $J_2$)
  CreateNode($\psi$, $\{\tau\}$)
  InsertBefore(($\psi$, $J_1$), ($\psi$, $J$))
  InsertBefore(($\psi$, $\{\tau\}$), ($\psi$, $J$))
  InsertBefore(($\psi$, $J_2$), ($\psi$, $J$))
  RemoveFromList(($\psi$, $J$))

procedure AddTrigger(node, grd)
  node.Triggers $\leftarrow$ node.Triggers $\cup$ {grd}

procedure AddGuard(node, precs)
  grd $\leftarrow$ {GuardOf = node, Precs = precs}
  foreach prec in precs do
    if IsNode(prec) then AddTrigger(prec, grd)
  node.Guards $\leftarrow$ node.Guards $\cup$ {grd}
```

```
procedure AddGuards($\psi$, $J$, $J_1$, $J_2$, $\tau$)
  foreach $J'$ in $\{J_1, \{\tau\}, J_2\}$ do
    case $\psi = \neg\alpha$
      AddGuard(($\psi$, $J'$), $\{(\alpha, J')\}$)
    case $\psi = \alpha \vee \beta$
      AddGuard(($\psi$, $J'$), $\{(\alpha, J')\}$)
      AddGuard(($\psi$, $J'$), $\{(\beta, J')\}$)
  case $\psi = \alpha \nabla_I \beta$
    AddGuardsSU($\psi$, $J$, $\tau$, $J_1$, $J_2$)
    foreach $J'$ in $\{J_1, \{\tau\}, J_2\}$ do
      if ($\psi$, $J'$).Guards = $\emptyset$ then
        SetTruthValue(($\psi$, $J'$), f)
```

Figure 6 The AddNodes and AddGuards procedures.

are built from the ones of the (ψ, J) node by the AddTriggersForAnchor, AddTriggersForCont, and AddGuards procedures. The procedures AddTriggersForAnchor and AddTriggersForCont are only relevant when ψ is a direct subformula of a formula γ with a temporal connective.

Before building the guards, NewTimePoint calls itself recursively to build the new nodes corresponding to ψ's subformulas. However, this is done only when the truth value associated to the node (ψ, J) is $\perp$. If (ψ, J) is already associated to a Boolean value then the new nodes inherit this truth value through a call to the SetTruthValue procedure, which might propagate the truth value to other nodes. Finally, the node (ψ, J) is deleted.

In the following, we give more details about the creation of the nodes, including their guards and triggers.

A.2.1 Node Creation.

The first part of the NewTimePoint procedure creates the nodes (ψ, J_1), (ψ, J_2), and $(\psi, \{\tau\})$. Their guards and triggers are created later. Initially, the truth value of the created nodes is $\perp$. Note that the NewTimePoint procedure later calls the SetTruthValue procedure if the node's (ψ, J) truth value is a Boolean value. Furthermore, it updates the list of nodes associated with the subformula ψ. These two steps are done by the AddNodes procedure, given on the left-hand side of Figure 6.

A.2.2 Guard and Trigger Creation.

We describe next how guards and triggers are built for the new nodes (ψ, J_1), (ψ, J_2), and $(\psi, \{\tau\})$. The guards together with their incoming triggers are built by the AddGuards procedure. The triggers of these nodes are generally also built by the same procedure, when called on the parent formula. Indeed, recall that there is a one-to-one correspondence between triggers and node preconditions. The triggers of a node (ψ, J) point to nodes (γ, K), where γ is ψ's parent. In case $J = K$, the triggers of the new nodes (ψ, J') with $J' \in \{J_1, J_2, \{\tau\}\}$ are added when adding the guards of the nodes (γ, J'). However, this is not the case when $J \neq K$. Furthermore, note that $J \neq K$ only when γ's main connective is a temporal connective. Thus, in order to add all triggers for the news nodes for such formulas γ, in NewTimePoint, we additionally call the two AddTriggers procedures.

The AddGuards procedure is given on the right-hand side of Figure 6. It uses the procedure AddGuard, which is also given in Figure 6, for the cases where ψ is $\neg\psi'$ and $\psi' \vee \psi''$. These two cases are as explained in Section 4. The case $\psi = \alpha \nabla_I \beta$ is handled by the procedure

```
// $\gamma = \alpha \nabla_I \beta$
procedure AddGuardsSU($\gamma$, $L$, $\tau$, $L_1$, $L_2$)
  $(L_f, L_c)$ ← GetFarClose($L_1$, $L_2$, $\nabla$)

  foreach grd in $(\gamma, L)$.Guards do
    precs ← NodePrec(grd)
    case precs = $\{(\beta, K), (\alpha, H)\}$
      if $K = L$
        AddGuardAC($(\gamma, L_f)$, $(\beta, L_f)$, $(\alpha, L_f)$)
        AddGuardAnchor($(\gamma, \{\tau\})$, $(\beta, \{\tau\})$)
        AddGuardAC($(\gamma, \{\tau\})$, $(\beta, L_f)$, $(\alpha, L_f)$)
        AddGuardAC($(\gamma, L_c)$, $(\beta, L_f)$, $(\alpha, L_f)$)
        AddGuardAC($(\gamma, L_c)$, $(\beta, \{\tau\})$, $(\alpha, L_c)$)
        AddGuardAC($(\gamma, L_c)$, $(\beta, L_c)$, $(\alpha, L_c)$)
      else
        if $H = L$ then $G \leftarrow L_f$ else $G \leftarrow H$
        foreach $L'$ in $\{L_f, \{\tau\}, L_c\}$ do
          AddGuardAC($(\gamma, L')$, $(\beta, K)$, $(\alpha, G)$)
    case precs = $\{(\beta, K)\}$
      if $K = L$
        AddGuardAnchor($(\gamma, L_f)$, $(\beta, L_f)$)
        AddGuardAnchor($(\gamma, \{\tau\})$, $(\beta, L_f)$)
        AddGuardAnchor($(\gamma, \{\tau\})$, $(\beta, \{\tau\})$)
        AddGuardAnchor($(\gamma, L_c)$, $(\beta, L_f)$)
        AddGuardAnchor($(\gamma, L_c)$, $(\beta, \{\tau\})$)
        AddGuardAnchor($(\gamma, L_c)$, $(\beta, L_c)$)
      else
        foreach $L'$ in $\{L_f, \{\tau\}, L_c\}$ do
          AddGuardAnchor($(\gamma, L')$, $(\beta, K)$)
    case precs = $\{(\alpha, H)\}$
      if $H = L$ then $G \leftarrow L_f$ else $G \leftarrow H$
      foreach $L'$ in $\{L_f, \{\tau\}, L_c\}$ do
        AddGuardCont($(\gamma, L')$, $(\alpha, G)$, grd)
    case precs = $\emptyset$
      foreach $L'$ in $\{L_f, \{\tau\}, L_c\}$ do
        AddGuardConstr($(\gamma, L')$, grd)
```

```
procedure GetFarClose($J_1$, $J_2$, $\nabla$)
  if $\nabla$ = S then return $(J_1, J_2)$
  else return $(J_2, J_1)$


// $\gamma = \alpha \nabla_I \beta$
procedure AddTriggersForAnchor($\gamma$, $K$, $\tau$, $K_1$, $K_2$)
  $(K_f, K_c)$ ← GetFarClose($K_1$, $K_2$, $\nabla$)
  foreach grd in $(\beta, K)$.Triggers do
    $L$ ← grd.GuardOf.Interval
    if $L \neq K$ then
      if HasCont(grd) then
        $H$ ← GetCont(grd).Interval
        if $H = K$ then
          AddGuardAC($(\gamma, L)$, $(\beta, K_f)$, $(\alpha, K_f)$)
          AddGuardAC($(\gamma, L)$, $(\beta, \{\tau\})$, $(\alpha, K_c)$)
          AddGuardAC($(\gamma, L)$, $(\beta, K_c)$, $(\alpha, K_c)$)
        else
          foreach $K'$ in $\{K_f, \{\tau\}, K_c\}$ do
            AddGuardAC($(\gamma, L)$, $(\beta, K')$, $(\alpha, H)$)
      else
        foreach $K'$ in $\{K_f, \{\tau\}, K_c\}$ do
          AddGuardAnchor($(\gamma, L)$, $(\beta, K')$)
      DeleteGuard(grd)

// $\gamma = \alpha \nabla_I \beta$
procedure AddTriggersForCont($\gamma$, $H$, $H_1$, $H_2$)
  $(H_f, \_)$ ← GetFarClose($H_1$, $H_2$, $\nabla$)
  foreach grd in $(\alpha, H)$.Triggers do
    $L$ ← grd.GuardOf.Interval
    if $L \neq H$ then
      if HasAnchor(grd) then
        if GetAnchor(grd).Interval $\neq H$ then
          UpdateContPrec(grd, $(\alpha, H_f)$)
      else
        UpdateContPrec(grd, $(\alpha, H_f)$)
```

Figure 7 The procedures for adding triggers for continuation and anchor nodes.

AddGuardsSU, which is explained next.

AddGuardsSU. This procedure is given on the left-hand side of Figure 7. In the pseudocode L and β correspond to J and ψ, respectively. L_f, L_c are J_1, J_2 if $\nabla = \mathsf{S}$ and to J_2, J_1 if $\nabla = \mathsf{U}$. The subscript f stands for "far" and the subscript c for "close" from the point of view of $\sup J$ if $\nabla = \mathsf{S}$ and $\inf J$ if $\nabla = \mathsf{U}$.

The procedure AddGuardsSU iterates through the guards of (ψ, J) to build the corresponding guards for the nodes (ψ, J'). For each guard grd, it distinguishes whether the guard has both an anchor and a continuation precondition, just one of these preconditions, or none of these. When the guard has an anchor precondition, it further distinguishes whether the anchor node (β, K) is the node (β, J) or not, that is, whether $J = K$. In the former case, at most six guards are added (one for the node (γ, L_f), two for $(\gamma, \{\tau\})$, and three for (γ, L_c)), depending on whether the corresponding temporal constraints are satisfied or not. These guards are represented by the upper dots in Figure 4. In the latter case (when $J \neq K$), for each new node at most one guard is added and the anchor precondition of the new guard is the same as the anchor precondition of the old guard, namely (β, K).

We now look at one of the cases in more detail, namely, the case when the guard has both preconditions, say (β, K) and (α, H), and furthermore $K \neq J$. If $H \neq J$, then the new guards and their incoming pointers are built as in Figure 4 (see lower dots), assuming that the temporal constraint for each new node is still satisfied. If $H = J$ the construction is similar, except that the node (α, L_f) is the new continuation node, because it is the farthest possible continuation node among the three new nodes. In this way, the invariant on anchor and continuation nodes is again satisfied.

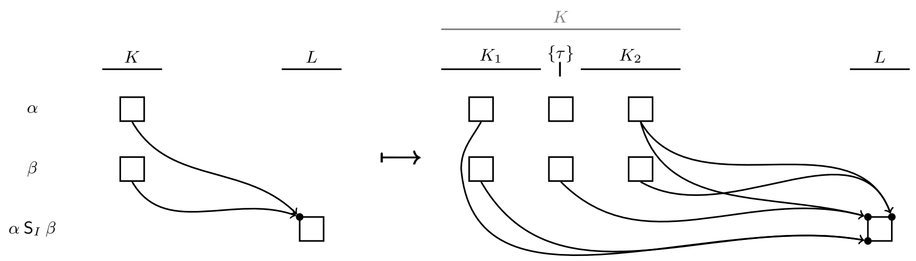

Figure 8 Illustration of AddTriggerForAnchor when $L \neq K$ and $H = K$.

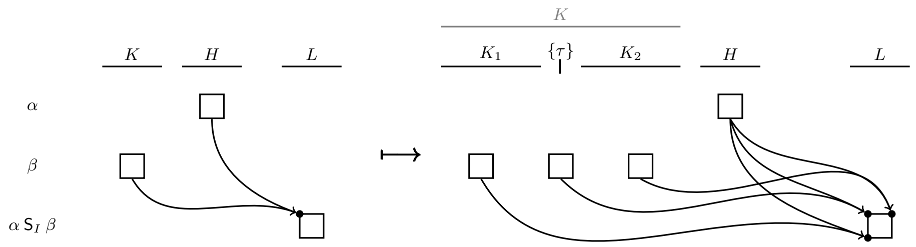

Figure 9 Illustration of AddTriggerForAnchor when K, L, H are pairwise disjoint.

The case when the guard grd only has an anchor as a node precondition is similar to the previous case. The other two cases are also similar, except that we use the guard grd to recover the temporal constraint, because in the absence of the anchor we cannot reconstruct the temporal constraint as done by the AddGuardAC and AddGuardAnchor procedures. Finally, note that if grd has no node precondition, then it has a temporal constraint precondition, because otherwise the guard would have been previously deleted.

AddTriggersForAnchor. This procedure is called when ψ is the anchor of γ, that is, $\gamma = \alpha \nabla_I \psi$ for some formula α. In the pseudocode, given on right-hand side of Figure 7, K, K_f, K_c, and α correspond to J, J_1, J_2, and ψ in the same way as for AddGuardsSU. The procedure iterates through the triggers of (ψ, J) and considers the guard grd to which each trigger points to. Let (γ, L) be the node to which grd belongs. When $L = J$ the triggers for the nodes (ψ, J') are added by the AddGuardsSU procedure, as explained previously. Otherwise, we distinguish the following cases.

Assume first that there is a corresponding continuation node (α, H). The new guards of the (γ, L) node and the triggers of the new nodes to these guards are shown in Figure 8 for the case when $H = J$, and in Figure 9 for the case when $H \neq J$. As before, the left and right hand sides of these figures show the relevant nodes before and after the call to AddTriggersForAnchor, respectively. Furthermore, the figures assume that the temporal constraints are satisfied for the new anchor nodes. Thus, actually fewer guards and triggers may be built than shown in the figures.

Assume now that there is no corresponding continuation node. This can happen when all possible continuation nodes were set to t before and were deleted. The node preconditions of the new guards are as in Figures 8 and 9, except of course that there are no continuation nodes nor their triggers.

Finally, after building the new guards, the old guard grd is deleted by the DeleteGuard procedure, given on the right-hand side of Figure 11.

AddTriggersForCont. This procedure is called when ψ is the continuation of γ, that is, $\gamma = \psi \nabla_I \beta$ for some formula β. In the pseudocode, given on the right-hand side of Figure 7,

```
procedure AddGuardSU(node, precs, constr)
  if IsConstrValid(constr) then
    if precs = $\emptyset$ then
      SetTruthValue(node, t)
    else
      AddGuard(node, precs)
  else if IsConstrSatisfiable(constr) then
    AddGuard(node, precs $\cup$ {constr})

procedure AddGuardAC(($\gamma, L$), ($\beta, K$), ($\alpha, H$))
  precs $\leftarrow$ {($\beta, K$), ($\alpha, H$)}
  AddGuardSU(($\gamma, L$), precs, ($\nabla_I, L, K$))

procedure AddGuardAnchor(($\gamma, L$), ($\beta, K$))
  precs $\leftarrow$ {($\beta, K$)}
  AddGuardSU(($\gamma, L$), precs, ($\nabla_I, L, K$))
```

```
procedure AddGuardCont(($\gamma, L$), ($\alpha, H$), grd)
  if HasConstrPrec(grd) then
    ($\nabla_I$, _, $K$) $\leftarrow$ GetConstrPrec(grd)
    AddGuardSU(($\gamma, L$), {($\alpha, H$)}, ($\nabla_I, L, K$))
  else
    AddGuard(($\gamma, L$), {($\alpha, H$)})

procedure AddGuardConstr(($\gamma, L$), grd)
  ($\nabla_I$, _, $K$) $\leftarrow$ GetConstrPrec(grd)
  AddGuardSU(($\gamma, L$), $\emptyset$, ($\nabla_I, L, K$))

procedure UpdateContPrec(grd, ($\alpha, H$))
  $H'$ $\leftarrow$ GetCont(grd).Interval
  grd.Precs $\leftarrow$ grd.Precs \ {($\alpha, H'$)}
  grd.Precs $\leftarrow$ grd.Precs $\cup$ {($\alpha, H$)}
```

Figure 10 Specialized procedures for adding and updating guards, where $\gamma = \alpha \nabla_I \beta$.

```
procedure DeleteNode(node)
  DeleteTriggers(node)
  DeleteGuards(node)

procedure DeleteTriggers(node)
  foreach grd in node.Triggers
    grd.Precs $\leftarrow$ grd.Precs \ {node}

procedure DeleteGuards(node)
  foreach grd in node.Guards
    DeleteGuard(grd)
```

```
procedure DeleteGuard(grd)
  foreach prec in grd.Precs
    if IsNode(prec) then
      prec.Triggers $\leftarrow$ prec.Triggers \ {grd}
      if prec.Triggers = $\emptyset$ then
        DeleteNode(prec)
  grd.GuardOf.Guards = grd.GuardOf.Guards \ {grd}
```

Figure 11 The deletion procedures.

H, H_f, H_c, and α correspond to J, J_1, J_2, and ψ in the same way as for AddGuardsSU. As before, the procedure iterates through the triggers of (ψ, J) and updates the corresponding guard grd and its node (γ, L). And again, when $L = J$ the triggers for the nodes (ψ, J') are added by the AddGuardsSU procedure. Otherwise, we distinguish the following cases.

Consider first that there is a corresponding anchor node (β, K). The case when $K = J$ was treated by the AddTriggersForAnchor procedure when adding triggers for the anchor nodes (β, J'); see also Figure 8. When $K \neq J$ we update the guard grd by calling the UpdateContPrec, which replaces the continuation precondition (ψ, H) with (ψ, H_f). This is in line with the invariant on the anchor and continuation nodes. Finally, the case when there is no corresponding anchor node is treated in the same way as the previous one.

A.2.3 Node Deletion.

After the new nodes (ψ, J_1), (ψ, J_2), and $(\psi, \{\tau\})$, and their guards and triggers are created, the node (ψ, J) is deleted by the DeleteNode procedure, given in Figure 11. Nodes are also deleted after their truth values have been propagated, see Section A.3. The DeleteNode procedure is recursive, as it can lead to other nodes being deleted. Indeed, when deleting a node we also delete all of its triggers and all of its guards and their incoming pointers. If a predecessor node in the graph structure remains without triggers, then it is deleted as well. Successor nodes however never remains without preconditions, as explained in Appendix B.

A.3 The SetTruthValue Procedure

The pseudocode of the SetTruthValue procedure is given on the left-hand side of Figure 12. The arguments are the node (ψ, J) to which a Boolean value is associated and the associated value b. The procedure first sets the Value field of the node to the Boolean value b. Next,

```
procedure SetTruthValue(($\psi$, $J$), $b$)
  if ExistsNode($\psi$,$J$) then
    ($\psi$, $J$).Value ← $b$
    $\gamma$ ← GetParent($\psi$)
    if $\gamma$ = nil then
      case $J = \{\tau\}$
        verdicts ← verdicts ∪ {($b$, $\tau$)}
        Report($\psi$, $b$, $\tau$)
        DeleteNode(($\psi$, $J$))
      default
        DeleteGuards(($\psi$, $J$))
    else
      case $\gamma = \psi \nabla_I \_$
        if $b$ then
          UpdateContNode(($\psi$, $J$), $\nabla$)
          Propagate(($\psi$, $J$), $b$)
          DeleteNode(($\psi$, $J$))
        else if IsSingleton($J$) then
          PropagateCont(($\psi$, $J$), $\nabla$)
          DeleteNode(($\psi$, $J$))
      case $\gamma = \_ \nabla_I \psi$ and $b$ and not IsSingleton($J$)
        DeleteGuards(($\psi$, $J$))
      default
        Propagate(($\psi$, $J$), $b$)
        DeleteNode(($\psi$, $J$))
```

```
procedure NoTimePoint($\psi$, $J$)
  foreach $\alpha$ in GetDirectSubformulas($\psi$) do
    NoTimePoint($\alpha$, $J$)
  $\gamma$ ← GetParent($\psi$)
  case $\gamma = \psi \nabla_I \_$
    UpdateContNode(($\psi$, $J$), $\nabla$)
    Propagate(($\psi$, $J$), t)
  case $\gamma = \_ \nabla_I \psi$
    Propagate(($\psi$, $J$), f)
  DeleteNode(($\psi$, $J$))
```

Figure 12 The SetTruthValue and NoTimePoint procedures.

the truth value is reported if ψ has no parent (that is, ψ is φ) and J is a singleton $\{\tau\}$ for some timestamp τ. Note that in this case we have obtained a verdict (b, τ) for the formula φ. Concretely, the verdict is added to the set stored in the global variable verdicts, and the procedure Report is called. This procedure sends a corresponding report message to the parent monitor, if any, when several monitors are deployed. It also performs application-specific reporting, for instance, writing the verdict to a file. After the report, the node is deleted. If J is not a singleton (and thus it is an incomplete interval), only the node's guards are deleted. The node may still be used later by the NewTimePoint procedure to create three new nodes when a notify message arrives.

If ψ is a proper subformula of φ, we distinguish several cases for propagating the Boolean value to other nodes. Namely, if (ψ, J) is a continuation node, then we restore the invariant concerning continuation nodes. This is done as follows. If b is the value t, then first a trigger from the next continuation node, if any, is added to each guard to which the current continuation node's triggers point to. This is done by the UpdateContNodes procedure, shown on the left-hand side of Figure 13. Next, the value t is propagated to the successor nodes, and finally the current node (ψ, J) is deleted. If b is the value f and J is a singleton, then f is propagated to all guards for which this continuation node is relevant. This is done by the procedure PropagateCont, also shown on the left-hand side of Figure 13. Note that the monitor cannot propagate f when J is not a singleton as it does not know whether there will be a time point with a timestamp in J. Propagating f when there is no time point with a timestamp in J would lead to incorrect results. Finally, if (ψ, J) is an anchor node, then a similar reasoning applies when b is t. That is, t is propagated and the node is deleted only when J is a singleton, otherwise only its guards are deleted. However, the value f is always propagated for anchor nodes. For all nodes that are neither anchor, nor continuation nodes, the truth value is propagated in all cases and the node is deleted. Finally, we note that propagation is done by the procedure Propagate, which closely follows the intuition presented in Section 4 (see right-hand side of Figure 13).

```
procedure UpdateContNode(($\alpha$, $H$), $\nabla$)
  contNode $\leftarrow$ GetFwdNeighbor(($\alpha$, $H$), $\nabla$)
  if contNode $\neq$ nil then
    $H'$ $\leftarrow$ contNode.Interval
    foreach grd in ($\alpha$, $H$).Triggers do
      $L$ $\leftarrow$ grd.GuardOf.Interval
      if IsRelevant($H'$, $L$, $\nabla$) then
        AddTrigger(($\alpha$, $H'$), grd)

procedure PropagateCont(($\alpha$, $H$), $\nabla$)
  contNode $\leftarrow$ ($\alpha$, $H$)
  do
    if contNode.Triggers = $\emptyset$ then
      relevant $\leftarrow$ true
    else
      relevant $\leftarrow$ false
      foreach grd in contNode.Triggers do
        ($\gamma$, $L$) $\leftarrow$ grd.GuardOf
        if IsRelevant($H$, $L$, $\nabla$) then
          relevant $\leftarrow$ true
          DeleteGuard(grd)
          if ($\gamma$, $L$).Guards = $\emptyset$ then
            SetTruthValue(($\gamma$, $L$), f)
    contNode $\leftarrow$ GetBwdNeighbor(contNode, $\nabla$)
  until contNode = nil or not relevant
```

```
procedure GetFwdNeighbor(($\alpha$, $J$), $\nabla$)
  if $\nabla$ = S then return ($\alpha$, $J$).Next
  else return ($\alpha$, $J$).Prev

procedure GetBwdNeighbor(($\alpha$, $J$), $\nabla$)
  if $\nabla$ = S then return ($\alpha$, $J$).Prev
  else return ($\alpha$, $J$).Next

procedure IsRelevant($H$, $L$, $\nabla$)
  if $H$ = $L$ return true
  else if $\nabla$ = S then return IsBefore($H$, $L$)
       else return IsBefore($L$, $H$)

procedure Propagate(($\psi$, $J$), $b$)
  foreach grd in ($\psi$, $J$).Triggers do
    ($\gamma$, $L$) $\leftarrow$ grd.GuardOf
    if $\gamma = \neg\psi$ then $b' \leftarrow \neg b$ else $b' \leftarrow b$
    if $b'$ then
      grd.Precs $\leftarrow$ grd.Precs \ {($\psi$, $J$)}
      if grd.Precs = $\emptyset$ then
        SetTruthValue(($\gamma$, $L$), t)
    else
      ($\gamma$, $L$).Guards $\leftarrow$ ($\gamma$, $L$).Guards \ {grd}
      if ($\gamma$, $L$).Guards = $\emptyset$ then
        SetTruthValue(($\gamma$, $L$), f)
```

Figure 13 Auxiliary procedures used by SetTruthValue and NoTimePoint.

```
procedure NewCompleteIntervals(msg)
  ret $\leftarrow$ $\emptyset$
  case msg = alive($C$,$\tau$,$s$)
    $K$ $\leftarrow$ UpdateCompleteIntervals($C$, $\tau$, $s$, false)
    ret $\leftarrow$ UpdateIncompleteIntervals($C$, $K$)
  case msg = notify($C$,$\tau$,$s$)
    $K$ $\leftarrow$ UpdateCompleteIntervals($C$, $\tau$, $s$, true)
    ret $\leftarrow$ UpdateIncompleteIntervals($C$, $K$)
  return ret
```

Figure 14 The procedure NewCompleteIntervals.

A.4 The NoTimePoint Procedure

The procedure NoTimePoint, shown on the right-hand side of Figure 12, simply deletes the node corresponding to an incomplete interval that has become complete and for which we know that there are no more time points inside. However, in case the node is a continuation node, then the procedure first updates the guards to which the node's triggers point to. This is done by calling the UpdateContNodes procedure, because we would update the continuation node in the same way if the node's truth value were indeed set to t. Similarly, in case the node is an anchor node, then the procedure deletes each guard to which triggers of this node point to. This is done by propagating the truth value f.

A.5 Complete Intervals

In the following, we provide algorithmic details about the procedure NewCompleteIntervals. See Figure 14. It comprises two steps, which are carried out by the two procedures UpdateCompleteIntervals and UpdateIncompleteIntervals. Their pseudocode is given in Figure 15.

We start by explaining the first step. The monitor maintains for each system component C a doubly-linked list complete$_C$, which stores triples of the form (c, I, c'), where $c, c' \in \mathbb{N}$ with $c \leq c'$ and $I \subseteq \mathbb{Q}_+$ is a nonempty interval. The intuition is that all timestamps from the component C in I have been received by the monitor and the first and last timestamp in I

have the sequence numbers c and c', respectively. The intervals are maximal (with respect to set inclusion). The triples in $\mathsf{complete}_C$ are ordered by their intervals. This is possible since they are maximal and hence nonoverlapping. Additionally, for each triple (c, I, c') we store two flags that record whether c and c' originate from notify messages.

Initially, $\mathsf{complete}_C$ is the singleton list $\langle(0, \{-1.0\}, 0)\rangle$. The triple $(0, \{-1.0\}, 0)$ can be understood as follows. The component C initially notifies the monitor that it is alive, i.e., it sends the message $\mathsf{alive}(C, -1.0, 0)$, which the monitor receives before the monitoring process is started.

The list $\mathsf{complete}_C$ is updated as follows when receiving alive and notify messages. If $\mathsf{complete}_C$ contains a triple (c, I, c') with $c \leq s \leq c'$, where s is the message's sequence number, we expand I. That is, we replace the triple by $(c, \{\tau\} \Cup I, c')$, where $A \Cup B$ is the smallest interval that contains the sets $A \subseteq \mathbb{Q}_+$ and $B \subseteq \mathbb{Q}_+$. If no such triple exists, we add the triple $(s, \{\tau\}, s)$ to $\mathsf{complete}_C$. In a second step, we merge triples when possible. Let (c, I, c') be the expanded or newly added triple.

- If the predecessor triple is of the form (d, J, d') with $d' = c - 1$ then we replace both triples by $(d, J \Cup I, c')$, provided that the monitor received a notify message with the sequence number c from the component C. To see the necessity of this provision, assume that the monitor has not received such a notify message yet. By the existence of the triple (c, I, c') in $\mathsf{complete}_C$, the monitor must have received at least one alive message with the sequence number c. It follows that the component C has sent a notify message with a timestamp between the right margin of J and the left margin of I. However, this message has not been received by the monitor. That is, the interval $J \Cup I$ is not complete. We note that the predecessor triple always exists because $\mathsf{complete}_C$ initially contains the triple $(0, \{-1.0\}, 0)$ and all timestamps contained in the received messages are greater than -1.0.
- Analogously, if the successor triple is of the form (d, J, d') with $d = c' + 1$ then we replace the two triples by $(c, I \Cup J, d')$, provided that the successor triple exists and the monitor received a notify message with the sequence number $c' + 1$ from the component C.

Note that if both cases apply, we merge the triple (c, I, c') with both its predecessor and its successor.

This update of the list $\mathsf{complete}_C$ is done by the procedure UpdateCompleteIntervals, see the left-hand side of Figure 15. The procedure UpdateCompleteIntervals uses the following auxiliary procedures.

- ContainsTriple returns a pointer to the triple (c, I, c') in the list $\mathsf{complete}_C$ with $c \leq s \leq c'$, if such a triple exists. Otherwise, it returns nil.
- AddTriple adds the triple $(s, \{\tau\}, s)$ to the list $\mathsf{complete}_C$. It returns the position of the added triple.
- ExpandTriple expands the triple to which pos points to in the list $\mathsf{complete}_C$.
- MergeTriples merges the triple at position pos in the list $\mathsf{complete}_C$ with the neighboring triples when possible. It also returns the pointer to the merged triple.
- UpdateTripleSequenceNumber stores for the triple at position pos that a notify message with the sequence number s was received.
- GetCompleteInterval returns the interval of the triple to which pos points to.

The implementation of these procedures is straightforward and we do not provide pseudocode for them here.

The second step of the procedure NewCompleteIntervals is implemented by the procedure UpdateIncompleteIntervals and is as follows. See also the right-hand side of Figure 15. The procedure UpdateIncompleteIntervals determines the new complete intervals, which it returns.

```
procedure UpdateCompleteIntervals($C$, $\tau$, $s$, isnotify)
  pos ← ContainsTriple($C$, $s$)
  if pos = nil then
    pos ← AddTriple($C$, $(s, \{\tau\}, s)$)
  else
    ExpandTriple(pos, $\tau$)
  if isnotify then
    UpdateTripleSequenceNumber(pos, $s$)
  pos ← MergeTriples(pos)
  return GetCompleteInterval(pos)
```

```
procedure UpdateIncompleteIntervals($C$, $K$)
  ret ← $\emptyset$
  if NonSingular($K$) then
    intervals ← GetIncompleteIntervals($K$)
    foreach $J$ in intervals do
      MarkCompleteForComponent($C$, $J$)
      if IsIntervalComplete($J$) then
        RemoveInterval($J$)
        ret ← ret ∪ $\{J\}$
  return ret
```

Figure 15 The auxiliary procedures UpdateCompleteIntervals and UpdateIncompleteIntervals.

For this, the monitor maintains a balanced tree of intervals. In this tree, the monitor stores the nonsingleton intervals J of nodes (ψ, J). Note that these intervals are nonoverlapping and hence we can order them and store them in a balanced search tree. For each of these intervals J, we maintain a set of components. This set stores the components from which the monitor might receive a notify message with a timestamp $\tau \in J$. When the set becomes empty, we know that the interval J is complete. Initially, the set of components of the interval $[0, \infty)$ contains all components. Furthermore, whenever an interval J' is split into the intervals $J' \cap [0, \tau)$, $\{\tau\}$ and $J' \cap (\tau, \infty)$, the intervals $J' \cap [0, \tau)$ and $J' \cap (\tau, \infty)$ inherit the set of components of J'. Note that when splitting J', we remove J' from the tree and add the intervals $J' \cap [0, \tau)$ and $J' \cap (\tau, \infty)$.

Suppose the interval K is nonsingular and all notify messages with a timestamp in K from the component C have been received by the monitor. For this interval K, the procedure UpdateIncompleteIntervals first obtains all the stored intervals J in the balanced search tree that are covered by K (GetCompleteIntervals), i.e., $J \subseteq K$. Then, UpdateIncompleteIntervals removes the component C from the sets of all these intervals (MarkCompleteForComponent). If one of these sets becomes empty (IsIntervalComplete), UpdateIncompleteIntervals removes the corresponding interval from the search tree (RemoveInterval) and adds it to the returned list of intervals. We omit the straightforward pseudocode of these auxiliary procedures.

B Additional Proof Details

B.1 Proof of Theorem 6

B.1.1 Termination.

We first note that each procedure terminates. First, there are no unbounded loops. Second, the recursive procedures visit one node of the graph structure per call. The procedures NewTimePoint, SetTruthValue, NoTimePoint only iterate through either triggers or preconditions and the subgraphs induced by triggers and (pointers to) preconditions, respectively, contain no cycles. The UpdateContNodes procedure only iterates in one direction through the doubly linked list of nodes corresponding to a subformula, and thus it also terminates. The DeleteNode procedure iterates through both triggers and preconditions. However, after traversing a link, it immediately deletes it, and thus cannot visit the same node twice. There are no other recursive procedures.

B.1.2 Soundness.

In the following, we prove Theorem 6(i). We first fix notation. Let O be the set of messages sent by the components. Furthermore, let $N \in \mathbb{N} \cup \{\infty\}$ be 1 plus the number of messages received by the monitor. Note that the monitor might only receive finitely many messages, i.e., $N \in \mathbb{N}$ and $N > 0$. Otherwise, when the monitor receives infinitely many messages, we have that $N = \infty$. For $0 < i < N$, we denote by m_i the ith received message. We recall that the monitor works iteratively, processing message m_i at iteration i, for each i with $0 < i < N$. By convention, iteration 0 is the execution of the initialization procedure Initialize. In iteration $i > 0$, the monitor receives and processes the message m_i, i.e., Monitor(φ, m_i) is executed.

The main invariant of our monitoring algorithm is as follows.

For any iteration i with $0 \leq i < N$ and any node (ψ, J), if (ψ, J).Value $\in$ 2, then $[\![w(O) \models \psi]\!]^\tau = (\psi, J)$.Value, for any $\tau \in J$ for which $\mathrm{tp}(w(O), \tau)$ is defined. (VAL)

The monitor's soundness follows easily from the invariant (VAL). Recall that the Monitor procedure returns the set verdicts at the end of an iteration $i > 0$, and this set is only updated with a new verdict (b, τ) during a call to SetTruthValue((φ, $\{\tau\}$), b). Before updating verdicts this procedure also sets (ψ, J).Value to b. Thus, by the invariant (VAL), if (b, τ) is a verdict at iteration $i > 0$, then $[\![w(O) \models \varphi]\!]^\tau = b$. Note that $\mathrm{tp}(w(O), \tau)$ is defined.[1] We note that no verdicts are obtained in iteration 0, that is, during initialization. Although the procedure Initialize might call the SetTruthValue procedure in some special cases, it is only called for nodes with the interval $[0, \infty)$.

It remains to establish the invariant (VAL). We first recall and introduce some terminology for the temporal connectives. For a formula $\gamma = \alpha \, \mathsf{S}_I \, \beta$, we call β the anchor of γ, and α the continuation of γ. We also recall what we mean by anchor nodes and continuation nodes. An anchor node for a node (γ, J) is any node (β, K) such that the constraint (S_I, J, K) is satisfiable. A continuation node for a node (γ, J) and an anchor node (β, K) of (γ, J) is any node (α, H) with H equal or before J, and H strictly after K, if K is a singleton, and equal with K or after K otherwise. We often omit and infer from the context the nodes (γ, J) and (β, K) with respect to which a continuation node is defined. The definitions are dual for $\gamma = \alpha \, \mathsf{U}_I \, \beta$, and are thus omitted.

In the following, we consider a virtual execution of our algorithm in which nodes together with their guards and triggers are not deleted by the SetTruthValue procedure. Instead, such nodes are marked with a color. Namely, instead of deleting a node, it is colored with green if its truth value is set to t and with red if its value is set to f. The virtual execution also adds triggers in some very special cases. Namely, the PropagateCont procedure adds a trigger (and the corresponding precondition) from the node (α, H) to the guard grd of the node (γ, L) when the IsRelevant(H, L, ∇) returns true. Note that this change would have no effect in the real execution because this trigger would be deleted by the call to DeleteGuard(grd). However, in the virtual execution the trigger remains, because, as mentioned previously, the guard grd is not deleted in the virtual execution. Propagation of Boolean values is performed as if nodes and their guards and triggers have been deleted. We emphasize that not every node on which SetTruthValue is called is colored, but only the nodes that would be deleted in the real execution. Furthermore, nodes are still deleted by the procedures NewTimePoint and NoTimePoint. That is, a node (ψ, J) for which J is split or for which J is complete is deleted. We reason about both the real execution and such a virtual execution. The execution mode we are referring to will be clear from the context if not mentioned explicitly. Finally, note that we can recover at any moment the real data structure from the virtual one

[1] The definedness follows from the fact that for any node (ψ, J), the interval J is a singleton $\{\tau\}$ iff $\mathrm{tp}(w(O), \tau)$ is defined (see the procedure NewTimePoint, in particular, the intervals J_1 and J_2 returned by the Split procedure are never singletons).

by deleting all colored nodes together with their guards and triggers. The following invariant characterizes the colored nodes of the data structure.

A node (ψ, J) is colored iff $(\psi, J).\mathsf{Value} \in 2$ and (a) if $\psi = \varphi$ then J is a singleton, (b) if ψ is an anchor then $(\psi, J).\mathsf{Value} = \mathsf{f}$, or $(\psi, J).\mathsf{Value} = \mathsf{t}$ and J is a singleton, and (c) if ψ is a continuation then $(\psi, J).\mathsf{Value} = \mathsf{t}$, or $(\psi, J).\mathsf{Value} = \mathsf{f}$ and J is a singleton. (COL)

The following invariant states that the intervals associated with the nodes of the virtual data structure cover the timestamps of all time points of $w(O)$.

For any subformula ψ of φ, and any $\tau \in \mathbb{Q}_+$ for which $\mathrm{tp}(w(O), \tau)$ is defined, there is a node (ψ, J) such that $\tau \in J$. (COV)

The invariant (COV) follows easily by induction on the number of calls to the NewTimePoint and NoTimePoint procedures. Note that these procedures are the only procedures that add and delete nodes in the virtual data structure in an iteration $i > 0$. The base case holds since the data structure initially consists of the nodes $(\psi, [0, \infty))$, where ψ is a subformula of φ. After a call to the NewTimePoint procedure, when a node (ψ, J) is deleted it is replaced by three new nodes. Their intervals partition the interval J. So, the invariant holds by the induction hypothesis. Consider a call to the NoTimePoint procedure where the node (ψ, J) is deleted. Then the interval J is complete, and thus there is no $\tau \in J$ such that $\mathrm{tp}(w(O), \tau)$ is defined (see Appendix A.5). This concludes the induction proof. Note that (COV) does not hold in an execution with the real data structure. There, nodes of the form $(\psi, \{\tau\})$ that are set to a Boolean value by the SetTruthValue procedure are deleted. For such nodes $\mathrm{tp}(w(O), \tau)$ is defined.

Since nodes are not deleted when assigning a Boolean value to them, we obtain the following structural properties of the virtual data structure.

(S1) A node $(\neg\psi', J)$ has one guard with one precondition, namely the node (ψ', J).

(S2) A node $(\psi_1 \vee \psi_2, J)$ has two guards with one precondition each, namely the node (ψ_1, J) and (ψ_2, J), respectively.

(S3) Let (ψ, J) be a node with $\psi = \alpha \,\mathsf{S}_I\, \beta$. (i) Every guard of (ψ, J) has at least one node precondition. There is exactly one node precondition of the form (β, K), and at most one corresponding constraint precondition. The constraint precondition is of the form (S_I, J, K), with K matching the mention node precondition, and it is present if and only if it is satisfiable and not valid. All the other preconditions are of the form (α, H). (ii) For any node (β, K), the node (ψ, J) has a guard with (β, K) as a precondition iff (β, K) is an anchor node for (ψ, J). (iii) For any guard of (ψ, J), if there is a red continuation node for the guard's anchor, then there is also a red continuation node among the guard's preconditions. Otherwise, the guard's preconditions of the form (α, H) consist of all green continuation nodes from the left most one up to and including the first uncolored continuation node, if any.[2]

(S4) Dual to (S3) for nodes (ψ, J) with $\psi = \alpha \,\mathsf{U}_I\, \beta$.

We prove that the invariant (VAL) holds in a virtual execution. We do this by first showing in Lemma 9 that the invariants (S1) to (S4) hold. Afterwards, in Lemma 10, we conclude—with the help of the other invariants—that (VAL) holds.

[2] That is, if there is no red or uncolored precondition, then the guard's preconditions of the form (α, H) consist of all continuation nodes.

▶ **Lemma 9.** *The invariants (S1) to (S4) hold.*

Proof. We only consider (S3). (S4) is dual to (S3), and (S1) and (S2) are straightforward and omitted. In the following, let (ψ, J) be a node with $\psi = \alpha \mathsf{S}_I \beta$. We show that if the node (ψ, J) satisfies (S3) before a change to the data structure, then it also satisfies (S3) after the change.

It suffices to consider the changes made to the virtual data structure after each call to the main procedures NewTimePoint, SetTruthValue, and NoTimePoint. We first note that (S3) is maintained when one of the nodes of the form (β, K), (α, H), or (ψ, J) is split, that is, after a call to NewTimePoint. Next, we note that (S3) is also maintained when one of the nodes of the form (β, K), (α, H), or (ψ, J) is deleted by the NoTimePoint procedure. We consider in detail the most interesting case, namely when SetTruthValue((ψ, J), b) is called from the procedures Propagate or PropagateCont. The other cases, namely when SetTruthValue is called by NewTimePoint, AddGuards, or AddGuardSU, are straightforward and thus omitted.

We assume that (ψ, J) satisfies (S3) before a call to SetTruthValue((ψ, J), b), for some $b \in 2$, and we show that it is satisfied after this call, but before the next call to SetTruthValue. We first note that the node (ψ, J) is not colored before the call. This is because we reason about the calls to SetTruthValue and Propagate in the real execution. If this node were colored, then it is deleted in the real execution and the calls cannot be made. We distinguish two cases, based on whether the procedure that calls SetTruthValue (i.e., Propagate or PropagateCont) is called on nodes of the form (β, K) or of the form (α, H).

Call on nodes of the form (β, K). The calling procedure is Propagate, as PropagateCont is only called on nodes of the form (α, H). In turn, Propagate is called from SetTruthValue or from NoTimePoint. If it is called from SetTruthValue, the only possible change to the virtual data structure is that the node (ψ, J) gets colored. As none of the three sub-invariants of (S3) is about the color of (ψ, J), it follows that (S3) is maintained. If Propagate is called from NoTimePoint then $b = \mathsf{f}$ and Propagate deletes the guard for which (β, K) is an anchor node. Since (ψ, J) has thus fewer guards than before the change, (S3i) and (S3iii) are clearly satisfied. (S3ii) is also satisfied, because the anchor node (β, K) is deleted. Note that it is deleted in the real execution by NoTimePoint and not by SetTruthValue; hence it is also deleted in the virtual execution.

Call on nodes of the form (α, H). We first focus on (S3iii), and consider (S3i) and (S3ii) afterwards. We distinguish two cases, based on the value of b.

- Suppose that $b = \mathsf{t}$. Then SetTruthValue((ψ, J), t) was called by Propagate((α, H), t). It follows that (α, H) is a precondition of some guard grd of (ψ, J). The call to Propagate must have been made either from NoTimePoint(α, H) or from SetTruthValue((α, H), t). In both cases, UpdateContNode((α, H), S) was called before Propagate. UpdateContNode adds a trigger to (ψ, J) from the real continuation node (α, H') at the right of (α, H), when such a node exists. Suppose that it does exist. We note that all the virtual continuation nodes (α, H'') with H'' between H and H' must be colored because they have been deleted in the real execution. Furthermore, they must be green because otherwise the guard grd would have been deleted in the real execution. For the same reason, there is no other red continuation node for guard grd. Then, since (S3iii) holds before the call to SetTruthValue, it also holds afterwards. The case where (α, H') has no real continuation node at its right is similar.
- Suppose that $b = \mathsf{f}$. Then SetTruthValue((ψ, J), f) is called by PropagateCont((α, H), S). Note that it could not have been called by Propagate((α, H), f). This can be easily seen by inspecting the places where Propagate is called from, namely SetTruthValue and

NoTimePoint. Let contNode be the node in the do-until loop of PropagateCont that has a trigger to a guard grd of (ψ, J). From the definition of the virtual execution, it follows that (α, H) is added as a precondition of grd. (Recall that the virtual execution adds extra preconditions.) Moreover, from the implementation of SetTruthValue and as $b = \mathsf{f}$, we have that (α, H) is red. (PropagateCont is called from SetTruthValue((α, H), f) which deletes the node (α, H)). Thus, to show (S3iii), it is sufficient to prove that (α, H) is a continuation node for the guard grd. The guard grd has the node contNode as a precondition. As contNode is equal or to the left of (α, H), it follows that (α, H) is to the right of the anchor node of grd. Since IsRelevant(H, J, S) returns true, we have that H equals or is at the left of J. Thus (α, H) is a continuation node for grd.

By reinspecting the previously analyzed changes to the virtual data structure, we see that guards and nodes of the form (β, K) are not affected. Thus (S3i) and (S3ii) are maintained. ◀

▶ **Lemma 10.** *The invariant (VAL) holds.*

Proof. Let i, ψ, and J be as in the invariant (VAL). It is sufficient to consider the case where i is the minimum among all iterations i' at which (ψ, J).Value $\in 2$. This is because a node's value is never changed after it has been set to a Boolean value.

Observe that (ψ, J).Value is only set to a Boolean value by a call SetTruthValue((ψ, J), b), where b is the value that is assigned to (ψ, J).Value. We thus analyze the calls to SetTruthValue. For any call SetTruthValue((ψ, J), b), one of the following call cases applies. The call was made: (1) during Initialize, when $\psi = \mathsf{t}$, $J = [0, \infty)$, and $b = \mathsf{t}$, (2) when receiving a report message, (3) when inheriting the truth value from a node (ψ, J') that was split, and (4) when propagating a truth value from preconditions. In the last call case, the call of SetTruthValue is done during the execution of one of the procedures Propagate, PropagateCont, AddGuards, or AddGuardSU.

We reason by induction over the lexicographic ordering on tuples $(i, |\psi|)$, where $|\psi|$ denotes the size of ψ. Note that for $i = 0$, only the call cases (1) and (4) are possible.

The base cases are where $i = 0$ and the formula ψ is atomic, that is, it is either the constant t or a proposition $p \in P$. Since $i = 0$, SetTruthValue is called from the Initialize procedure. Since the nodes $(p, [0, \infty))$ have no guards, the call case (4) does not apply. Thus $\psi = \mathsf{t}$ and $b = \mathsf{t}$. The invariant trivially holds.

For the inductive step, the call case (1) does not apply. It remains to consider the other call cases. For this, let $\tau \in J$ be an arbitrary timestamp for which $\mathrm{tp}(w(O), \tau)$ is defined.

We start with the call case (2). That is, the message report(p, b, τ) has been received at iteration i and we have $\psi = p$ and $J = \{\tau\}$. From the definition of $w(O)$ we have that $w(O)$ contains a letter (σ, τ) with $\sigma(p) = b$. In particular $\mathrm{tp}(w(O)), \tau)$ is defined. Thus $[\![w(O) \models \psi]\!]^\tau = b$ and the invariant is maintained for this case.

We now consider the call case (3). That is, the Boolean value (ψ, J).Value was set when splitting the node (ψ, J') for some J' with $J \subseteq J'$. In turn, (ψ, J').Value was set by a previous call to SetTruthValue((ψ, J'), b) during some iteration $i' < i$. By the induction hypothesis, we have that $[\![w(O) \models \psi]\!]^{\tau'} = b$, for any $\tau' \in J'$ such that $\mathrm{tp}(w(O), \tau')$ is defined. As $J \subseteq J'$, we conclude that $[\![w(O) \models \psi]\!]^\tau = b$.

In the remainder of the proof, we consider the call case (4). We make a case distinction on the form of ψ. The cases where ψ is atomic are trivial, since the nodes of atomic subformulas have no guards (and no preconditions) and thus truth values are not propagated to them. For the remaining cases, note that the SetTruthValue is called from PropagateCont, AddGuards, or AddGuardSU only if ψ is of the form $\alpha \nabla_I \beta$. Hence, for the nontemporal cases, SetTruthValue is called by Propagate.

Case $\psi = \neg\psi'$. From (S1) we have that the node (ψ, J) has only one guard, with (ψ', J) as the only precondition. $\mathsf{SetTruthValue}((\psi, J),\ b)$ was thus called from the execution of $\mathsf{Propagate}((\psi', J),\ b')$, during the same iteration i. From the implementation of $\mathsf{Propagate}$ it follows that $b' = \neg b$. From the induction hypothesis we have that $[\![w(O) \models \psi']\!]^\tau = b'$. From MTL's semantics, it follows that $[\![w(O) \models \psi]\!]^\tau = b$.

Case $\psi = \psi_1 \vee \psi_2$. From (S2) we have that the node (ψ, J) has exactly two guards, each with one precondition, namely (ψ_1, J) and (ψ_2, J) respectively. $\mathsf{SetTruthValue}((\psi, J),\ b)$ was thus called from the execution of $\mathsf{Propagate}((\psi_j, J),\ b_j)$, during the same iteration i, for some $j \in \{1, 2\}$ and $b_j = (\psi_j, J).\mathsf{Value}$. We have that $b_j \in \mathsf{2}$. If $b_j = \mathsf{t}$ then the guard with the precondition (ψ_j, J) remains with no precondition in the real execution, and thus $b = \mathsf{t}$. (VAL) then follows easily from the induction hypothesis. If $b_j = \mathsf{f}$ then the guard with the precondition (ψ_j, J) is removed in the real execution. Since b is propagated it must be the case that (ψ, J) remains with no guards in the real execution and that $b = \mathsf{f}$. Thus the other guard of (ψ, J) has been previously been removed in the real execution. Then it must be that the case that the node $(\psi_{j'}, J)$ is red, where $j' = 3 - j$. Otherwise, the Boolean value t would have been propagated earlier. Again, (VAL) then follows easily from the induction hypothesis.

Case $\psi = \alpha\ \mathsf{S}_I\ \beta$. Let $j \in \mathbb{N}$ be $w(O)$'s time point with timestamp τ. We consider the state of the (virtual and real) data structures after the propagation has been done, that is at the end of the execution of $\mathsf{SetTruthValue}$ that called the $\mathsf{Propagate}$ procedure. We distinguish two cases, based on b's value.

- Suppose that $b = \mathsf{t}$. We have a guard grd of (ψ, J) that remains without preconditions in the real data structure. From (S3i) and (S3ii), this guard has an anchor node (β, K) as a precondition in the virtual data structure. This anchor node must be green. Indeed, if it is red then at the iteration when it has turned red the Boolean value f would have been propagated and the guard grd would have been deleted in the real data structure—this is a contradiction. If the anchor node is uncolored then grd remains with a precondition in the real data structure—again a contradiction. For the same reason the constraint (S_I, J, K) must be valid. Otherwise, this constraint remains a precondition of grd in the real data structure.
  We now show that all continuation nodes for (β, K) are green. If there is a red continuation node, then from (S3iii) grd has a red precondition and, as above, this results in a contradiction. So all grd's preconditions of the form (α, H) consist of all green continuation nodes up to and including the first uncolored continuation node, if any. As above, there can be no uncolored precondition. We conclude that all continuation nodes are green.
  As the anchor node (β, K) is green, it follows from (COL) that K is a singleton, that is, $K = \{\kappa\}$ for some $\kappa \in \mathbb{Q}_+$. Then, as noted before (see footnote 1), we have that $\mathrm{tp}(w(O), \kappa)$ is defined. Let k be a time point in $w(O)$ with the timestamp κ. Let $h \in \mathbb{N}$ be a time point with $k < h \leq j$ and let η be its timestamp. From (COV) there is a node (α, H) such that $\eta \in H$. As $\kappa < \eta \leq \tau$, we have that (α, H) is a continuation node. Hence it is green. Then $(\alpha, H).\mathsf{Value} = \mathsf{t}$. From the induction hypothesis, we have that $[\![w(O) \models \alpha]\!]^\eta = \mathsf{t}$. From the induction hypothesis, we also have that $[\![w(O) \models \beta]\!]^\kappa = \mathsf{t}$. As h was chosen arbitrarily, it follows from MTL's semantics that $[\![w(O) \models \psi]\!]^\tau = \mathsf{t}$.
- Suppose that $b = \mathsf{f}$. The node (ψ, J) remains without guards in the real data structure. We assume, by absurdity, that $[\![w(O) \models \psi]\!]^\tau \neq \mathsf{f}$. Then, there is a $k \leq j$ such that $[\![w(O), k \models \beta]\!] \neq \mathsf{f}$, $\kappa - \tau \in I$, and $[\![w(O), h \models \alpha]\!] \neq \mathsf{f}$ for all h with $k < h \leq j$. Let κ be the timestamp of k in $w(O)$. From (COV), we have that there is a node (β, K) at iteration i such that $\kappa \in K$. From the induction hypothesis this node cannot be red, as

otherwise $[\![w(O) \models \beta]\!]^\kappa = \mathsf{f}$. Since $\kappa - \tau \in I$, it follows that the constraint (S_I, J, K) is satisfiable and thus (β, K) is an anchor node for some guard grd of (ψ, J) (in the virtual data structure). We can easily deduce that all its continuation nodes are not red (if they would be then a contradiction follows with $[\![w(O), h \models \alpha]\!] \neq \mathsf{f}$ for all h with $k < h \leq j$, from the induction hypothesis). From (S3iii) we obtain that all grd's preconditions of the form (α, H) consist of all green continuations nodes up to the first uncolored continuation node, if any. If grd has at least one uncolored node precondition or (S_I, J, K) is not valid, then the guard remains with a precondition and it is thus not deleted in the real data structure—a contradiction. If all grd's node preconditions are green and the constraint (S_I, J, K) is valid, then the node (ψ, J) is green and thus already deleted in the real data structure—again a contradiction. As we have reached a contradiction in all cases, we conclude that the assumption was false, and thus $[\![w(O) \models \psi]\!]^\tau = \mathsf{f}$.

Case $\psi = \alpha\, \mathsf{U}_I\, \beta$. This case is dual to the previous case. ◀

B.1.3 Completeness.

For proving Theorem 6(ii), let φ be a bounded formula. Furthermore, let $\tau \in \mathbb{Q}_+$. Assume that $[\![w(O) \models \varphi]\!]^\tau \in 2$. Since φ is bounded, only a finite prefix of $w(O)$ is relevant for the verdict at time τ. Since every message is eventually received, we have that the monitor has eventually complete knowledge about this prefix. This includes that the monitor knows that there are no further time points in this prefix.

If the verdict at time τ has not been reported yet, then the node $(\varphi, \{\tau\})$ must still contain at least one guard. However, since the monitor knows that there are no further time points in this prefix, all guards have triggers from a node associated to an existing time point. It is now easy to see by induction over the formula structure that the algorithm propagates the Boolean value $[\![w(O) \models \varphi]\!]^\tau$ to the node $(\varphi, \{\tau\})$.

B.2 Proof of Lemma 8

Note that the timestamp τ either occurs at the same time point $i \in \mathbb{N}$ in v and v', or it does not occur at all in v and v'. In the latter case, we have by definition that $[\![v \models \varphi]\!]^\tau = [\![v' \models \varphi]\!]^\tau = \bot$.

The other case is proved by induction over the formula structure, namely, for all formulas φ, for all $v, v' \in TW^\omega(\Sigma)$, and all $i \in \mathbb{N}$, if $v \preceq v'$ then $[\![v, i \models \varphi]\!] \preceq [\![v', i \models \varphi]\!]$. The base cases t and $p \in P$ are obvious. The step cases for the connectives $\neg$ and $\vee$ are also straightforward, since the corresponding logical operators are monotonic in $\prec$. The step cases for the temporal connectives S_I and U_I follow easily from their definition. Recall that their semantics is based on the logical operators $\wedge$ and $\vee$, which are both monotonic in $\prec$.