9-2017

Monitoring Time Intervals

Teng Zhang
University of Pennsylvania, tengz@cis.upenn.edu

John Wiegley
john.wiegley@baesystems.com

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Recommended Citation

Teng Zhang, John Wiegley, Insup Lee, and Oleg Sokolsky, "Monitoring Time Intervals", *The 17th International Conference on Runtime Verification (RV 2017)* . September 2017. http://dx.doi.org/10.1007/978-3-319-67531-2_20

The 17th International Conference on Runtime Verification (RV 2017), Seattle, USA, September 13 - 16, 2017

Monitoring Time Intervals

Abstract

Run-time checking of timed properties requires to monitor events occurring within a specified time interval. In a distributed setting, working with intervals is complicated due to uncertainties about network delays and clock synchronization. Determining that an interval can be closed - i.e., that all events occurring within the interval have been observed - cannot be done without a delay. In this paper, we consider how an appropriate delay can be determined based on parameters of a monitoring setup, such as network delay, clock skew and clock rate. We then propose a generic scheme for monitoring time intervals, parameterized by the detection delay, and discuss the use of this monitoring scheme to check different timed specifications, including real-time temporal logics and rate calculations.

Keywords

Runtime verification, Time interval monitoring, Real-time properties

Disciplines

Computer Engineering | Computer Sciences

Comments

The 17th International Conference on Runtime Verification (RV 2017), Seattle, USA, September 13 - 16, 2017

Monitoring Time Intervals*

Teng Zhang[1], John Wiegley[2], Insup Lee[1], and Oleg Sokolsky[1]

[1] University of Pennsylvania, Philadelphia PA, USA,
{tengz,sokolsky,lee}@cis.upenn.edu
[2] BAE Systems, Burlington MA, USA,
john.wiegley@baesystems.com

Abstract. Run-time checking of timed properties requires to monitor events occurring within a specified time interval. In a distributed setting, working with intervals is complicated due to uncertainties about network delays and clock synchronization. Determining that an interval can be closed – i.e., that all events occurring within the interval have been observed – cannot be done without a delay. In this paper, we consider how an appropriate delay can be determined based on parameters of a monitoring setup, such as network delay, clock skew and clock rate. We then propose a generic scheme for monitoring time intervals, parameterized by the detection delay, and discuss the use of this monitoring scheme to check different timed specifications, including real-time temporal logics and rate calculations.

Keywords: Runtime verification, Time interval monitoring, Real-time properties

1 Introduction

In this paper, we consider runtime verification of timing properties, such as one event occurring after another event within certain time bound or counting the number of events that occur during an interval of time. In both cases, a monitor needs to not only evaluate the logic of the property but also determine whether events fall within a given time interval. We consider the situation when the system being monitored (referred to as the target system or just system, when the context is clear) and the monitor are deployed in an asynchronous environment. On the one hand, the asynchronous approach makes monitoring more difficult, due to uncertain delays introduced by the network delivering events from the system to the monitor and also due to the differences between the system and monitor clocks. On the other hand, by using the monitor clock that is different from the system clock, we may be able to detect that timing behavior of the target system is incorrect because the system clock is wrong.

We propose a monitor architecture that clearly separates monitoring of time intervals from the rest of property checking. The property is checked in an event-driven fashion similar to common approaches to runtime verification. To enable

* This work is supported in part by DARPA BRASS program under contract FA8750-16-C-0007 and by ONR SBIR contract N00014-15-C-0126.

checking of the timing in this way, we extend the set of events with a new kind of event that represents the end of a time interval, which we call *interval closure*. Now, we can reduce time checking to *temporal ordering*: if a system event arrives before the closure event, it occurred within the time interval, while if the closure event arrives first, the system event is outside of the interval. In order to produce closure events in the right order, we introduce the *interval handler* module into the monitor.

The second aspect addressed in the paper is the design of the interval handler. We note two particular design considerations for the handler: one is *correctness* and the other is *timeliness*. On the one hand, the handler needs to correctly monitor intervals, in the sense that it should close an interval – that is, raise the closure event – only after any event occurring within the interval has been received. In the presence of uncertainty, correct monitoring is possible only if the handler waits long enough to make sure it has seen all relevant events. On the other hand, closing the interval too late may increase unnecessary resource consumption for monitoring, which should be avoided. Moreover, we should know what the tight one is, in order to be certain that the deadline set is larger than the tight one. It is therefore important to set the monitoring *deadline* as small as possible under the premise that correctness of the closure is guaranteed.

To summarize, this paper addresses the following problem: *"Given an asynchronous environment with uncertain communication delay and imperfect clock synchronization between target system and monitor, under what conditions can correctness of monitoring time intervals be ensured and how to achieve it?"*

In this paper, we consider three parameters of monitoring setup, network delay, clock skew and clock rate, and study how they influence monitoring time intervals. We explore the parameter space and present a scheme for setting the deadline of monitoring for each interval. We then introduce an algorithm that the interval handler uses to monitor intervals.

Related Works. In [1], Sammapun considered properties represented with time-bound operators and analyzed several different implementations of checking properties based on timer and heartbeats with bounded or unbounded network delay. However, clock rate and clock skew were not taken into consideration. Moreover, properties using aggregate operators were not studied. In [2], Lee and Davidson proposed algorithms for implementing timed synchronous communication among processes having different clocks such that all processes will decide whether the communication is successfull within their own absolute deadlines and they agree on the same decision. Two communication schemes, multiple senders with one receiver and N-way communication were analyzed. In [3], they further analyzed the performance of two algorithms of timed synchronous communication using probabilistic models. We do not consider synchronous communication in this paper so the method of setting the deadline is different. In [4], Pinisetty et al. proposed a paradigm of runtime enforcement using time retardants on events to ensure that a system satisfies timed properties. While we are not aiming at an enforcement scenario, we will rely on a similar technique in the case when events may be delivered from the system to the monitor out of

order. Jahanian et al. studied the runtime monitoring of time constraints specified by Real-Time Logic (RTL) in the distributed real-time system [5]. However, the monitoring procedure of time intervals was not discussed. Finkbeiner et al. presented a query language for asynchronously collecting statistic information and proposed algebraic alternating automata for evaluating the queries in [6]. Colombo et al. presented the tool LarvaStat [7], which supports statistic operations based on the incrementally computable statistics. It also supports the specification of intervals opened and closed by special events. However, the issue of monitoring intervals was not discussed in detail. Basin et al. extended MFOTL (metric first-order temporal logic) with SQL-like aggregate operators over time and corresponding monitoring algorithms [8]. In [9], they further raised the problem of imprecise timestamps of traces influencing the correct verification of the properties specified by MTL formulas. The paper gave the conclusion that certain MTL fragments can be verified by existing monitors for precise traces over traces with imprecise timestamps. In our work, we do not verify properties using a specific logic but rather focus on studying the issue of monitoring intervals. To summarize, little work has been done in monitoring time intervals in a distributed environment based on the analysis of parameters of the monitoring setup.

Paper organization. The paper is organized as follows. Section 2 provides motivation for the work and gives examples of timed property specifications that can be checked in the proposed fashion. Based on this motivation, we introduce the monitor architecture in Section 3 and lay out requirements for the interval handler. We then introduce the system model and definitions of parameters considered: network delay, clock skew and clock rate. Based on these, Section 4 presents the exploration of the parameter space and addresses the problem of setting deadline for monitoring time intervals. Section 5 proposes a procedure for monitoring intervals. Section 6 gives a further discussion on cases when the correctness of interval monitoring cannot be guaranteed and introduces future work.

2 Motivation

Several kinds of commonly used timed specifications involve reasoning over time intervals. We note that, while the logic of evaluating these properties over a stream of events is different, it invariably involves reasoning about intervals of time given in the specification and whether the timestamp of a given observation falls within an interval or outside of it. As we discuss below, parameters of the monitoring setup, such as clock skew or the latency of delivering observations to the monitor, have an impact on how this reasoning should be performed. We therefore want to separate the logic of property evaluation, which depends only on the semantics of the specification language, and interval management, which depends on properties of the monitoring setup.

To illustrate our approach, we first briefly revisit two of them: LTL with interval operators and interval statistics.

LTL with time-bound operators. In LTL, operator Until(U), Weak-until(W) and R(Release) are used to specify properties in a trace. For instance, property $\phi_1 U \phi_2$ is satisfied in a trace if ϕ_1 is satisfied at each location of the trace until ϕ_2 is satisfied at a certain point. The verdict can not be given to this property until getting the result from the verification of ϕ_2. To restrict the time of getting the result, the time-bound operator is utilized [1]. If we want to express the property that ϕ_2 becomes satisfied within 5 time unit from ϕ_1 becoming true, the formula is written as $\phi_1 U_{[0,5]} \phi_2$.

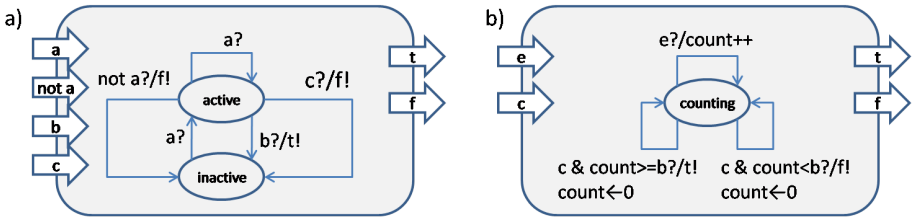

Fig. 1. Evaluation of interval operators

In many runtime verification approaches [10–12], temporal operators are evaluated in an event-driven fashion. Arriving events, which could be observations from the target system or results of sub formula evaluation, trigger changes in the operator evaluation status. We want to extend the same approach to interval operators. Consider, for example, evaluation of the bounded-until $aU_{[0,t_1]}b$, where a and b are target system observations. As Fig. 1, a) shows, evaluation of the operator is a state machine that takes as inputs events a, b, and c. Event *not* a represents the absence of a. We refer to the event c as the *interval closure*, which denotes that t_1 time units have elapsed. Note that t_1 is measured in the sense of perfect clock, which may be different from the clock on the system and the monitor side due to the clock skew. Evaluation is activated by an arrival of a, and while further occurrences of a arrive, the state of the evaluation is unresolved. As soon as *not* a arrives, or if the interval is closed, the operator evaluates to false, denoted by raising an event f. But if b arrives before the interval is closed, the operator evaluates to true and an event t is raised. In this way, evaluation of the operator does not depend on the value of the time bound and does not need direct access to the clock. It is straightforward to extend this scheme to cover intervals of the form $[t_1, t_2]$, as well as cover other commonly used temporal operators. Note that to monitor $aU_{[t_1,t_2]}b$ we consider intervals $[0, t_1)$ and $[t_1, t_2]$. When b arrives, we determine, which of the two interval it falls into, or if it is outside of both. For technical reasons that will be discussed later, we open both intervals when a arrives.

Interval statistics. Some properties needs to collect statistics over a time interval. These properties can be represented in a similar way as SQL queries using aggregate operators [8]. For instance, $Sum_{[0,t_1]}(occur(e)) >= b$) specifies

the property of the number of occurrences of event e over the time interval [0,t_1] is equal or greater than b. Fig. 1,b) shows the evaluation scheme for this operator in a fashion similar to the previous case. Variable *count* increases with arrivals of event e. When *interval closure* event c arrives, the interval is closed. An event t is raised if *count* is greater or equal than b; otherwise an event f is raised.

In contrast to interval operators discussed above, calculation of interval statistics is different in the sense that intervals are *recurrent*. On the system side, once an interval ends, the next one is immediately started and statistics calculation continues for the next interval, effectively partitioning the time line into intervals of the same size, starting from some initial event. We can view recurrent intervals as an extension of the two-interval case above.

Checking example. Fig. 2 shows a concrete scenario for monitoring of $aU_{[t_1,t_2]}b$ when system events can be delivered with a delay. Assume first that the clocks in both the system and the monitor are perfect. On the monitor side, we begin processing when the event a arrives at relative time 0. To correctly evaluate this property, the monitor needs to tell whether b falls within $i_1 = [0, t_1)$ or within $i_2 = [t_1, t_2]$. Suppose an event b is raised before t_1 but is delayed more than a was and thus arrives after the time t_1 on the monitor side. Thus, at t_1 the monitor cannot yet conclude that i_1 has expired. From the monitor perspective, i_1 and i_2 *overlap*; that is, an incoming event may belong to either interval. However, once we see the timestamp of b, we can tell whether it belongs to i_1 or i_2. Therefore, we do not need to measure duration of i_1 or i_2 on the monitor side. Now consider the case when b does not arrive within i_2. In order to conclude that b did not arrive in time, the monitor has to wait. Eventually, another event with a large enough timestamp may arrive and the monitor may be able to make the conclusion based on that. But what if it arrives after a very long time or, worse, if the missing b was meant to be the last observation? To proceed in a more timely fashion, the monitor has to use a timer. This timer, essentially, sets the *deadline* for b to arrive. This observation underlies our monitoring approach: we use the timer only to safely close the interval, while all other conclusions – whether the interval has started and whether an event is within the interval – are made based on event timestamps.

Apart from the network delay, the clock rate of the system and the monitor also influence interval monitoring. Using the same example above, assume first that there is no clock skew and delivery delay is ranged from 0 to 1 in the sense of the perfect clock. Suppose the clock rate of the perfect clock r_p is 1, clock rate of the system r_s is 0.5 and clock rate of the monitor r_m is within range $[0.8, 1.5]$. Interval i_1 to be monitored is $[0, 6]$ measured by the perfect clock and the monitor begins monitoring it at time 0. To guarantee that all events occurring in i_1 arrive before the monitor finishes monitoring this interval, the deadline of monitoring is set at time 10.5 of the monitor clock, as in the worst case, an event occurs at 3 of system clock (corresponding to time 6 in the sense of the perfect clock as the clock rate is 0.5) arrives at time 10.5 of the monitor clock with the largest delay. If the actual rate of r_m is 1.5, when an event b happens at time 3.1 of the system clock and the network delay is 0.2 then, it arrives at the monitor at

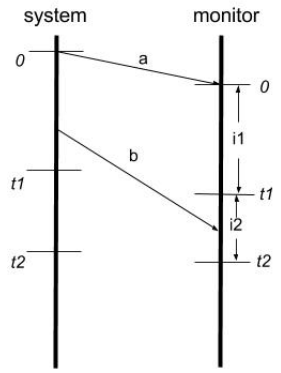

Fig. 2. Monitoring time intervals of $aU_{[t_1,t_2]}b$

time 9.6 (calculated by 3.1*3 + 0.2*1.5). However, since we know the clock rate of the system is 0.5, the time on the perfect clock will be 6.2, which is larger than 6. Therefore, even if b arrives when the monitor is monitoring the interval, the monitor can still determine b does not belong to it. This example suggests that the deadline for monitoring an interval depends only on the duration of the interval and the relationship between the monitoring clock and the perfect clock, but not on the system clock. At the same time, to determine whether an event is within an interval depends on the relationship between the system clock and the perfect clock, but not on the monitor clock. We will make this intuition precise in Sections 4 and 5.

3 System Architecture and preliminaries

In this section, we will present the architecture for monitoring time intervals. Then some preliminaries are given, including definitions of some key concepts and parameters of monitoring setup to be explored in the paper.

3.1 Architecture

Fig. 3 illustrates the architecture for monitoring time intervals. To separate the logic of time management, a module $IntervalHandler$ is introduced into the monitor between the target system and the property checker. Both the IntervalHandler and checker run under the monitor clock. The checker implements the logic the property to be checked so the implementation detail is out of scope of this paper. It receives two types of events from the IntervalHandler, one is the original events for property evaluation. Another is a special event *interval closure* introduced above, which is used to acknowledge to the checker the end of a time interval.

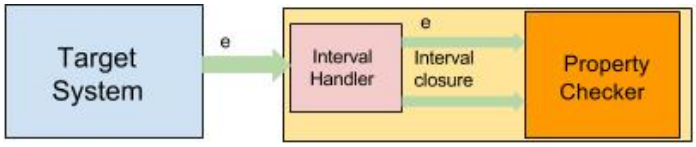

Fig. 3. Architecture for monitoring time intervals

A checker correctly evaluates the property for a time interval i if all events occurring in i are delivered to the checker when the property is being evaluated. In the ideal situation, when events are delivered from the system to the monitor immediately and there is no timing uncertainty, this can be easily achieved by setting the timer in the IntervalHandler for the duration of i. Any event arriving before the timer expires would be within i, while any event arriving after it expires is outside i. Expiration of the timer immediately raises the closure event. If events can be delayed, however, this approach may clearly result in incorrect checking. The closure event must be delayed to accommodate for late events. In order to close the interval in a timely manner, we need to set a *deadline* for raising the closure event that would, on the one hand, guarantee correct monitoring and, on the other hand, minimize the delay in closing the interval. According to the duration on the time interval and parameters of the monitoring setup, the IntervalHandler can calculate the deadline for each interval when the monitoring process begins. When the current time at the monitor reaches the deadline, the IntervalHandler will send interval closure event to the checker to finish the evaluation of the property for this interval.

The deadline discussed above is useful in another way. If events can arrive out of order, they also should be re-ordered according to their timestamps before being passed on to the checker which, as we discussed above, does not reason about time. In our approach, the IntervalHandler is storing events in a queue in the timestamp order and uses the same deadline to release events from the queue to the checker. We discuss event reordering further in Section 5.

3.2 Preliminaries

Time model. There are three time domains assumed: T_m for the monitor clock, T_s for the system clock and T_p for the perfect clock. The monitor takes streams of events as input. Events are observations originated from the system. They are timestamped using the *system clock* in the time domain T_s. The event and monitor clocks can be skewed and run at different rates. In addition, there may be unpredictable delays in delivering events from the target system to the monitor. As a result, event timestamps are not directly comparable with readings of the system clock. Moreover, elements in the time domain T_m and T_s are totally ordered. An event stream E_T is a sequence of timestamped observations $\langle (o_1, t_1), (o_2, t_2), ... \rangle$, where o_i is a value observed at time $t_i \in T_m$. The perfect clock c_p in T_p is used to measure the length of the time interval being monitored.

Time interval is a period of time between two events, the duration of which is measured by the perfect clock. In the remainder of the paper, when we refer the interval on the system, we use "start" and "end" to denote the beginning and ending of the interval. On the monitor side, an interval is "opened" or "closed" by the monitor. A closed interval i that starts at t_1 and ends at t_2 is denoted as $i_{[t_1,t_2]}$. For an event e originated from the system and an interval i, if $t_1 \leq t_e \leq t_2$, then $e \in i_{[t_1,t_2]}$ where t_e is the timestamp of e. Note that if we don't care about events occurring on the bound(s), the interval could also be half-open or open and the denotation will be modified accordingly.

Network delay, denoted as nd, represents the time to send the event from the system to the monitor. The absolute value of the delay is measured in the sense of perfect clock.

Clock rate is the interval of the finest time unit. It is assumed that the clock rate of c_p, denoted as r_p, is 1. The clock rate of the system and the monitor are respectively denoted as r_s and r_m. If r_s (r_m) is greater than 1, then the system (monitor) clock runs ahead of the perfect clock.

Clock skew, denoted as ts, represents the time difference $t_m - t_s$ between the monitor and the system where t_s is the time of the system and t_m is the time of the monitor. In this paper, we assume that time synchronization is periodically conducted between (1) the system clock and the perfect clock and (2) the monitor clock and the perfect clock.

4 Setting the Interval Deadline

In this section, we explore the parameter space of network delay, clock skew and clock rate and identify several cases where correctness of monitoring can be ensured. For each case, we describe how to calculate the deadline for closing the interval. The monitor uses this deadline to set the timer; when the timer expires, we can be certain that no further events belonging to this interval can arrive and the closure event is sent to the checker. Patterns of setting the timer for non-recurrent and recurrent intervals are presented respectively. Case analysis on the three parameters is conducted.

4.1 Patterns of setting timer

We rely on timers to determine when an interval can be closed. The timers are set differently based on whether the interval is recurrent or non-recurrent, shown in Fig. 4. Note that the clock rate of the system rs is used to calculate the actual time on the system side.

Non-recurrent intervals. Here we only consider the case involving two consecutive intervals such as the property $aU_{[t_1,t_2]}b$. In $aU_{[t_1,t_2]}b$, two intervals, $[0, t_1)$ and $[t_1, t_2]$, are involved. The monitor begins checking $[0, t_1)$ and $[t_1, t_2]$ when a arrives and two corresponding timers are set to close the intervals.

Recurrent intervals. As the number of intervals to be monitored is unbounded, only the timer for the first interval is set. Then every time an interval is closed, the timer for closing the next interval is set with a proper monitoring deadline. In the following section, we will denote the duration of the recurrent interval as d.

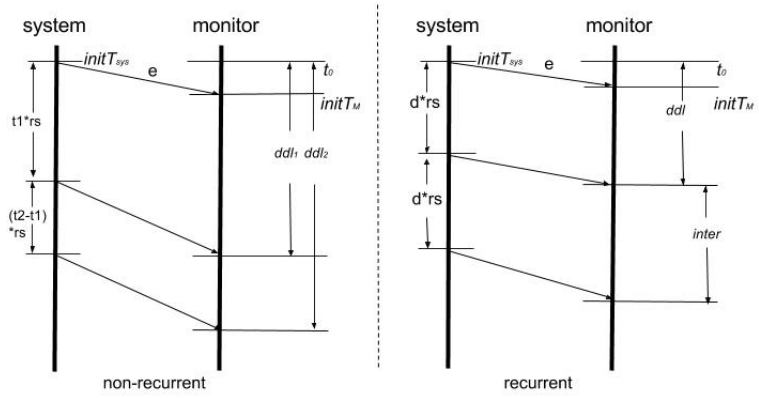

Fig. 4. Scheme of setting deadlines for non-recurrent and recurrent intervals

In order to set the deadline as accurate as possible, two steps have to be done. The first step is to estimate the time on the monitor side when e occurs on the system side, denoted as t_0 in Fig. 4. The second step is to calculate deadlines for each monitor based on t_0, which is introduced below.

4.2 Scheme of setting deadline

Here we give the case analysis with varying the values of the clock rate, network delay and clock skew with the assumption of bounded network delay. Fig. 4 illustrates the scheme of setting deadline for non-recurrent and recurrent intervals. The time when the initial event e occurs on the system side is denoted as $initT_{sys}$, measured by the system clock and $initT_M$ is the time at the monitor when e arrives at the monitor.

The monitor begins the monitoring process at $initT_M$. For the non-recurrent case, ddl_1 and ddl_2 for interval $[0, t_1)$ and $[t_1, t_2]$ need to be calculated. Then, as the timers are set at $initT_M$ with a relative value, deadline for $[0, t_1)$ is set with value $ddl1 - initT_M + t_0$ and deadline for $[t_1, t_2]$ is set with value $ddl2 - initT_M + t_0$. For the recurrent case, the deadline for the first interval can be calculated in a similar way to the non-recurrent case: ddl is calculated according to the duration of interval and the monitoring setup and the deadline for the first interval is $ddl - initT_M + t_0$. From the second interval, timers are set with a period $inter$. The reason the first interval is different from the rest of them is that for the initial event e, we know the exact time when e arrives at the monitor, but for

the rest of intervals, we only consider the worst case where the last event for a interval occurs at the boundary and the delay for the delivery is the maximum value of the network delay. In the following case analysis, we will estimate the value of t_0 and calculate ddl_1 and ddl_2 for the non-recurrent case; ddl and $inter$ for the recurrent case.

Case 1 : $r_s = 1$, $r_m = 1$, $nd = 0$. In this case, interval durations of the system and monitor are identical and there is no delay, so $t_0 = initT_M$. For the case of non-recurrent intervals, ddl_1 and ddl_2 are respectively t_1 and t_2. For the case of recurrent intervals, ddl and $inter$ have the same value d since there is no network delay.

Case 2 : $r_s = 1$, $r_m = 1$, nd is fixed and known. In this case, clock skew ts can be directly calculated by $initT_M - initT_{sys} - nd$ and $t_0 = initT_{sys} + ts$. For the case of non-recurrent intervals, ddl_1 and ddl_2 are respectively $t_1 + nd$ and $t_2 + nd$ since events occurring at the boundary of these two intervals have the delay of nd. For the case of recurrent intervals, ddl is set to $d + nd$, similar to the case of the non-recurrent interval. The value of $inter$ is set to d because the interval is of length d and the network delay has already been taken into consideration when calculating the deadline of the first interval.

Case 3 : $r_s = 1$, $r_m = 1$, $nd \in [b1, b2]$, ts is known. As ts is known, $t_0 = initT_{sys} + ts$. We only need to consider the worst case in which network delay has the maximum value, which is when an event e with timestamp t arrives on the monitor side at $t + b2$. The least delay $b1$ is not relevant for computing deadlines. For the case of non-recurrent intervals, ddl_1 and ddl_2 are respectively $t_1 + b2$ and $t_2 + b2$. For the case of recurrent intervals, ddl is $d + b2$ and $inter$ has value d.

Case 4 : $r_s = 1$, $r_m = 1$, $nd \in [b1, b2]$, ts is unknown. The analysis is similar to the case 4 but t_0 cannot be determined precisely since ts is unknown and network delay is not fixed. Consequently, we approximate its value using the network delay. The worst case is when the value of t_0 is as late as possible. Therefore, we set $t_0 = initT_M - b1$. The same formulas setting deadlines used in case 3 are also used here.

Case 5: r_s is fixed, $r_m \in [r3, r4]$, $nd \in [b1, b2]$, ts at time $initT_{sys}$ is known. Like in case 3, t_0 is calculated using the formula $t_0 = initT_{sys} + ts$. Because of the clock rate difference between the system and the monitor, clock skew may change. However, since we do not compare time values between the system and the monitor anywhere else, the value of the clock skew does not affect calculations of the deadline value. To cover the worst case of event arrival when calculating the deadline, r_m and nd need to be at their upper bounds. For the case of non-recurrent intervals, ddl_1 and ddl_2 are respectively $(t_1 + b2) * r4$ and $(t_2 + b2) * r4$. For the case recurrent intervals, ddl has value $(d + b2) * r4$ and $inter$ has value $d * r4$.

Case 6: r_s is fixed, $r_m \in [r3, r4]$, $nd \in [b1, b2]$, ts is unknown. Similar with case 4, we need to approximate t_0 using its maximum value: $initT_M - b1 * r3$. The formulas used in case 6 are used in this case.

One can observe that case 5 and 6 are generalization of special cases 1 to 4 and there is no conflicts between them. The summary of case analysis on deadline setting is shown in Table 1. We can prove that given monitoring setup in case 5 and 6, correctness of monitoring intervals can be guaranteed, shown in Lemma 1.

Table 1. Summary of deadline setting scheme

Monitoring setup		Non-recurrent		Recurrent	
	t_0	$ddl1$	$ddl2$	ddl	$inter$
$r_s = 1$, $r_m = 1$, $nd = 0$	$initT_M$	t_1	t_2	d	d
$r_s = 1$, $r_m = 1$, nd is fixed and known, ts is known	$initT_{sys} + ts$	$t_1 + nd$	$t_2 + nd$	$d + nd$	
$r_s = 1$, $r_m = 1$, $nd \in [b1, b2]$, ts is known	$initT_{sys} + ts$	$t_1 + b2$	$t_2 + b2$	$d + b2$	
$r_s = 1$, $r_m = 1$, $nd \in [b1, b2]$, ts is unknown	$initT_M - b1$				
r_s is fixed, $r_m \in [r3, r4]$, $nd \in [b1, b2]$, ts at time $initT_{sys}$ is known	$initT_{sys} + ts$	$(t_1 + b2) * r4$	$(t_2 + b2) * r4$	$(d + b2) * r4$	$d * r4$
r_s is fixed, $r_m \in [r3, r4]$, $nd \in [b1, b2]$, ts is unknown	$initT_M - b1 * r3$				

Lemma 1 (Correctness of Monitoring Interval for Setup in Case 5 and 6). *If r_s is fixed, $r_m \in [r3, r4]$ and $nd \in [b1, b2]$, we can always set a deadline for monitored intervals as illustrated in Table 1, such that all events of the interval will fall within the deadline.*

Proof Sketch. Based on whether ts is known at the beginning of monitoring process, we split into two cases corresponding to case 5 and 6 above. Here we give the sketch for proving the case of monitoring non-recurrent intervals $[0, t_1)$. The proof for interval $[t_1, t_2]$ and recurrent intervals is similar. Recall that t_0 is the estimated time, by the monitor clock, when the initial event occurs on the system side. The deadline is set in two steps, illustrated in Fig 4, and we argue correctness of these two steps separately. First, we compute the largest possible value for t_0 and this is correct because 1) if ts known, we can calculate the accurate time t_0 of the monitor given the timestamp of $initTsys$ when the initial event occurs on the system side; and 2) if ts is not known, we compute t_0 having the maximum value using the $initT_M$ and the lower bound of nd. Then, we set the deadline relative to t_0 and we do it correctly because we over-estimate the deadline with the upper bound of r_m and nd. Finally, we compare the deadline with t_r, the relative time between $initT_{sys}$ and the latest possible arrival time of the event occurring at t_1 at the monitor. The value of t_r is $t_1 + b2$ in the sense of perfect clock. Translating deadline to the perfect time scale, the

value would be $(t_1 + b2) * r4/r_m$, which is greater than or equal to t_r. Since t_0 is equal to or greater than the time when the initial event occurring within the interval, we can always ensure that all events will fall within the deadline.

Lemma 1 can be extended to Theorem 1 describing sufficient condition for correctly monitoring time intervals.

Theorem 1 (Correctness of Monitoring Interval). *If r_s is fixed, r_m is bounded and nd is bounded, we can set a deadline for each monitored interval as illustrated in Table 1 such that all events of the interval will arrive at the monitor within the deadline.*

Proof Sketch. The proof proceeds by case analysis of entries in Table 1. Note that cases 1-4 are special cases of 5 and 6 and need not be considered separately. The union of the monitoring setup conditions in Table 1 is exactly the premise of the theorem. Therefore, correctness of cases 5 and 6, established by Lemma 1, proves the theorem.

5 Monitoring Procedure

This section presents the procedure for monitoring time intervals using the scheme of setting monitoring deadline proposed in the previous section. The procedure describes operation of the IntervalHandler introduced in Section 3.1.

The procedure relies of two key functions. First, *calculateDeadline* sets the deadline for each interval according to Section 4. Second, procedure *getInterval* is given an event and returns an interval to which this event belongs, as follows. Given an event e with the timestamp t and $initT_{sys}$ which indicates the occurring time of the initial event, we need to get the interval that e belongs to. With the condition that the rate of the system r_s is fixed, the interval can be determined. For the non-recurrent interval, if $t - initT_{sys} < t_1 * rs$, e belongs to the interval $[0, t_1)$; if $t_1 * rs \leq t - initT_{sys} \leq t_2 * rs$, e belongs to the interval $[t_1, t_2]$; otherwise, e falls out of these two intervals. For the recurrent interval, the interval is calculated using the formula $\lfloor (t - initT_{sys})/(d * rs) \rfloor$.

Fig. 5 shows the detailed structure of the *IntervalHandler* and how it connects to the *PropertyChecker*. IntervalHandler is responsible for managing intervals and the checker evaluates the logic of the property. Note that the monitoring process is slightly different between the cases of in-order-delivery and out-of-order delivery. *IntervalList* is the data structure representing intervals of interest. In the non-recurrent case, these are the two intervals $[0, t_1)$ and $[t_1, t_2]$. In the recurrent case, if in-order delivery is assumed, we just need to remember the earliest non-closed interval. For out-of-order delivery, *IntervalList* needs to remember all non-closed intervals for which at least one event has been received. We also associate a data structure $eventQueue(i)$ for each interval i in the *IntervalList*: each arrived event is put into corresponding *eventQueue* ordered by the timestamp. Once the interval i is closed — that is, no more events from this interval can arrive, — the IntervalHandler sends all events in the $eventQueue(i)$ to the checker, followed by the interval closure event.

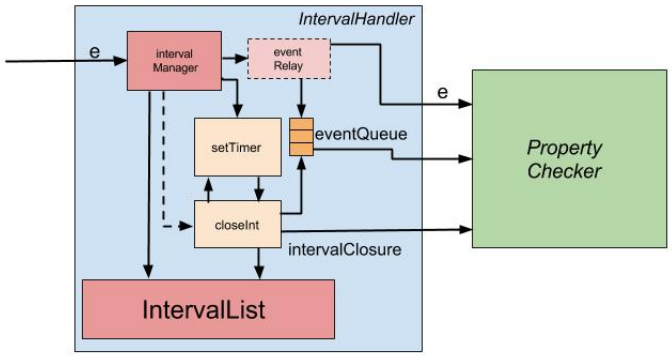

Fig. 5. Structure for the IntervalHandler

In the IntervalHandler, *intervalManager* is used to relay events from the system and manage intervals. It first examines whether the received event e is the initial event arriving at the monitor. If so, it computes the deadline and sets the timer for the first interval. Note that in the case of properties involving two non-recurrent intervals, two timers with corresponding deadlines need to be set. Then, the interval i that e belongs to is computed. If in-order delivery is assumed, the current interval being evaluated by the checker, denoted as i', is obtained from *IntervalList* by calling the procedure $getLeastOpenedInt$. If i is not equal to i', i' is closed and corresponding closing timer will also be unset. Event e is then sent to the checker. If out-of-order delivery is assumed, it is put into corresponding $eventQueue(i)$.

```
void intervalManager (){
   while(true) {
      Interval i;
      Event e = receiveEvent();
      if (initialEvent (e)){
         initialTS = e.getSystemTimeStamp();
         initialTM = getcurrentTime();
         deadline = calculateDeadline();
         setTimer(deadline,0);
      }
      i = getInterval(e);
      if (out-of-order-delivery){
          addQueue(e,eventQueue(i));
      }else{
         i' = getLeastOpenedInt();
         if(i != i'){
            closeInt(i');
            unsetTimer(i');
```

```
        }
        PropertyChecker.handlingEvents(e);
    }
}
```

Procedure $closeInt(i)$ is responsible for closing the interval i, which is called when the corresponding timer is up or an event for the next interval has arrived in the case of in-order delivery. It first calculates the deadline for the next interval $i+1$ to be evaluated and sets the corresponding timer. For the case of non-recurrent interval, the timer will not be not set. Then the queued events from $eventQueue(i)$ is be sent to the checker. For in-order delivery, there is no action on event queue. Finally, interval i is closed by sending $intervalClosure(i)$ to the checker. For the case of recurrent interval, interval i is removed from $IntervalList$ and $i+1$ is set as the earliest non-closed interval.

```
void closeInt(integer i){
     ddl = calculateDeadline();
     setTimer(ddl, i+1);
     liste = getEventsForQueue(eventQueue(i));
     PropertyChecker.handlingEvents(liste);
     intervalClosure(i);
 }
```

6 Discussion and Conclusions

This paper presented an approach to monitoring of time intervals in an event-driven fashion. To do this, we introduced an interval closure event, with the property that all events that fall into the interval occur before the interval closure. The two challenges are (1) correctness of the procedure and (2) timeliness of the event closure. To address these two challenges, we offer a procedure to determine when all events that can fit into the interval have been observed. The answer to this question depends on parameters of monitoring setup, namely network delay, clock skew between the system and the monitor and clock rates of the two. We perform case analysis and show how to close intervals in different cases.

This work has two limitations. First, we can exactly determine when we have seen all the events only if the network delay is bounded. Second, we assumed that we can precisely determine whether a given event is within the bounds of an interval or outside. In general, neither of these two assumptions are true. This means that the monitoring procedure needs to be augmented to accommodate the uncertainty. Below, we offer preliminary remarks on what extensions may be needed.

Unbounded network delay. If the monitoring system is built on a complicated network environment, the network delay can be unbounded. This means we cannot guarantee that the deadline will be sufficiently large to receive all

events from the system. We have to accept that, occasionally, an interval will be closed prematurely. A possible approach is to quantify the probability of error.

For example, we consider the delay distribution where events are independent with each other. To simplify the analysis, following assumptions are also made: (1) the system and the monitor has perfect clocks, (2) there is no clock skew, and (3) the event occurrence is distributed uniformly. With these assumptions, we can set a deadline that ensures the probability for one event occurring in the current interval falls out of it is less than $1 - p$. This relation can be represented as the formula below. According to the assumption, $P_1(t)$ is $1/d$ and P_2 is the CDF of delay distribution.

$$\int_0^d P_1(t)P_2(delay \geq d + deadline - t) < 1 - p$$

For a more realistic approach, we can consider more widely used *self-similar* traffic models [13] such as Pareto distribution and Weibull distribution. Different from the memoryless Poisson distribution, self-similar traffic models can perform better in modeling burstiness of traffic in the multiple time scales [14]. Events are not independent with each other and arrivals of events will heavily influence the model.

With this approach we can have a monitoring procedure with probabilistic guarantees of correctness. Moreover, once we discover an event that belongs to an already-closed interval, we know that a monitoring error has occurred. The property checker needs to be notified of the error, which may invalidate some of the checking results. In the case of interval statistics, it may be acceptable to discard the statistics for one of the intervals. In the case of temporal monitoring, we need to determine, which parts of the formula are affected by the error. We will consider a three-valued semantics for the temporal logic, with the "unknown" value corresponding to an error.

Uncertainty in the system clock rate. We need to determine whether an event, timestamped with the system clock, falls within an interval, whose boundaries are determined by the perfect clock. However, if r_s can vary, then an event close enough to an interval boundary cannot be precisely placed. In this case, we can also use a 3-valued semantics, with the third value representing the uncertainty whether the event occurs before or after the interval closure event.

It remains to be seen whether the two three-valued approaches – the one capturing an error and the one capturing the ordering uncertainty – can be combined together in an effective checking procedure. We will explore these questions in future work.

References

1. Sammapun, U.: Monitoring and checking of real-time and probabilistic properties. PhD thesis, University of Pennsylvania (2009)

2. Lee, I., Davidson, S.B.: Adding time to synchronous process communications. IEEE transactions on computers **100**(8) (1987) 941–948
3. Lee, I., Davidson, S.B.: A performance analysis of timed synchronous communication primitives. IEEE Transactions on Computers **39**(9) (1990) 1117–1131
4. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Timo, O.N.: Runtime enforcement of timed properties revisited. Formal Methods in System Design **45**(3) (2014) 381–422
5. Jahanian, F., Rajkumar, R., Raju, S.C.: Runtime monitoring of timing constraints in distributed real-time systems. Real-Time Systems **7**(3) (1994) 247–273
6. Finkbeiner, B., Sankaranarayanan, S., Sipma, H.B.: Collecting statistics over runtime executions. Formal Methods in System Design **27**(3) (2005) 253–274
7. Colombo, C., Gauci, A., Pace, G.J.: Larvastat: Monitoring of statistical properties. In: International Conference on Runtime Verification, Springer (2010) 480–484
8. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. In: International Conference on Runtime Verification, Springer (2013) 40–58
9. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: On real-time monitoring with imprecise timestamps. In: International Conference on Runtime Verification, Springer (2014) 193–198
10. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M.: Java-MaC: a runtime assurance approach for Java programs. Formal Methods in Systems Design **24**(2) (March 2004) 129–155
11. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. Journal of Logic and Computation **20**(3) (2010) 675–706
12. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Automated Software Engineering **12**(2) (2005) 151–197
13. Leland, W.E., Taqqu, M.S., Willinger, W., Wilson, D.V.: On the self-similar nature of ethernet traffic. In: ACM SIGCOMM Computer Communication Review. Volume 23., ACM (1993) 183–193
14. Becchi, M.: From poisson processes to self-similarity: a survey of network traffic models. Washington University in St. Louis, Tech. Rep (2008)