10-17-2005

Checking Correctness At Runtime using Real-Time Java

Usa Sammapun
University of Pennsylvania

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

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

Proceedings of the 3rd Workshop on Java Technologies for Real-time and Embedded Systems (JTRES'05), San Diego, CA, October 17, 2005.

Checking Correctness At Runtime using Real-Time Java

Abstract
Correctness of a real-time system depends on its computation as well as its timeliness. In recent years, research has been focusing on verifying the correctness of a real-time system during runtime by monitoring its runtime execution and checking it against its formal specifications. Such verification method is called Runtime Verification. While a few existing runtime verification tools verify both computational correctness and timeliness correctness, those that provide timeliness correctness fail to detect timeliness violations as soon as violations occur. In this paper, we investigate the verification of timeliness correctness by providing quantitative property specifications, address the problem why those tools fail to detect as soon as violations occur, provide an efficient solution, and present how to implement it in Real-Time Java.

Comments
Proceedings of the 3rd Workshop on Java Technologies for Real-time and Embedded Systems (JTRES'05), San Diego, CA, October 17, 2005.

Checking Timeliness Correctness At Runtime using Real-Time Java

Usa Sammapun, Insup Lee, and Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389 USA
{usa,sokolsky,lee}@cis.upenn.edu

ABSTRACT
Correctness of a real-time system depends on its computation as well as its timeliness. In recent years, research has been focusing on verifying the correctness of a real-time system during runtime by monitoring its runtime execution and checking it against its formal specifications. Such verification method is called Runtime Verification. While a few existing runtime verification tools verify both computational correctness and timeliness correctness, those that provide timeliness correctness fail to detect timeliness violations as soon as violations occur. In this paper, we investigate the verification of timeliness correctness by providing quantitative property specifications, address the problem why those tools fail to detect as soon as violations occur, provide an efficient solution, and present how to implement it in Real-Time Java.

1. INTRODUCTION
Real-time systems are distinct from other kinds of computer systems in that not only do they need to perform their computations correctly, but the results of the computation also need to be delivered in a timely manner. The need to consider both computation and timeliness correctness is an additional obstacle to the design of correct and reliable real-time systems. In recent years, researches have been focusing on verifying the correctness of a real-time system during runtime by monitoring its runtime execution and checking it against its formal specifications. Such verification method, called Runtime Verification [3, 4, 5, 6, 9], is more practical than other verification methods such as model checking and testing. Runtime verification verifies directly on the implementation of the system rather than on the system model as done in model checking. While testing is done in an ad hoc manner, runtime verification is based on formal logics.

In this paper, we explore the use of runtime verification tools to verify real-time systems. Most existing runtime verification tools verify computation correctness of a system using qualitative property specifications but do not provide verification of timeliness correctness. One of such tools is MaC (Monitoring and Checking) [8, 9]. MaC verifies computation correctness of a system by providing specification of qualitative properties in a language based on Linear Temporal Logic (LTL) [14]. During runtime, it extracts necessary observations from the system's runtime execution, and checks the observations against the system properties. In this paper, we extend MaC with the capability to verify timeliness correctness by providing quantitative property specifications and call the extension RT-MaC. The additional quantiative property specification is acheived by introducing time-bounded temporal operators. Such operators can specify a time limit in which a property must hold. The resulting language is similar to Metric Temporal Logic (MTL) [10].

There are a few existing runtime verification works aiming at verifying timeliness correctness, but their implementation fails to detect a timeliness violation as soon as it occurs. Computation correctness depends on changes only in system computation while timeliness correctness depends on changes both in system computation and in time. The existing works evaluate properties in response to only computation changes but not time changes. We call the evaluation in response to computation changes "event-driven" and the evaluation in response to time changes "time-driven". In this paper, we study what kinds of problems we encounter when doing time-driven evaluation, how we resolve such problems, and how we extend the current MaC system written in Java into the RT-MaC system written in Real-Time Java.

The paper is organized as follows. Section 2 presents related work. Section 3 provides background on MaC. Section 4 describes the RT-MaC language, which provides syntax and semantics for quantitative properties. Section 5 addresses problems when doing time-driven evaluation, provides solutions to the problems, and presents how to implement them in Real-Time Java. Finally, Section 6 concludes the paper.

2. RELATED WORK
There are a few existing runtime verification researches for quantitative properties. Mok and Liu [16] present an efficient runtime monitoring mechanism to detect violations for timing constraints using graph theories. The successors of the paper [15, 13] present the monitoring of timing con-

straints on time intervals and timing constraints with confidence threshold requirements. Their works only provide checking for timeliness correctness but not computation correctness while RT-MaC aims at both. Thati and Rosu [17] study monitoring algorithms for MTL [10], a logic that allows quantitative property specification. Their algorithms evaluate by transforming MTL formulae into a "canonical form" after some changes in computation. The transformed formulae specify what need to be held at the next state. The evaluation algorithm for EAGLE specification by Barringer et al. [2] is similar. Their specification language allows one to express temporal logics, extended regular expressions, and quantitative properties, among others. The works by Thati and Rosu [17] and by Barringer et al. [2] implement only the event-driven evaluation, but not the time-driven one.

Other runtime verification of quantitative properties are studied by Jayaputera et al. [7] and by Kristoffersen et al. [12]. Jayaputera et al. [7] provides quantitative property verification using Windows Management Instrumentation and .NET. The work by Kristoffersen et al. [12] is similar to Thati and Rosu [17]. They also transform quantitative properties based on Timed-LTL [1] into a canonical form called disjunctive normalized equation. Timed-LTL also allows one to express quantitative properties. Both of these works take a different approach. They evaluate properties every time step. In practice, correctness of properties does not usually change every time step, and therefore, such evaluation is considered redundant and inefficient. Kristoffersen et al. [11]'s following work improves and allows both event-driven and time-driven evaluations. Their work however does not consider the possibility of data race occurring during time-driven evaluation. Finally, none of the runtime verification of quantitative properties attempt to use Real-Time Java to implement their tools.

3. MAC OVERVIEW

3.1 MaC Architecture

The MaC system has been developed to ensure that a program runs correctly with respect to its formal requirement specification. Fig. 1 shows the overall structure of the MaC architecture. The system works as follows. A user specifies a requirement of a target program in a formal MaC specification language. Given a target program and the specification, the MaC system inserts a collection of probes or a *filter* into the target program to extract relevant observations such as assignments to program variables and function calls and returns. The specification is compiled into a MaC verifier specialized for the target program.

During runtime, the execution of the probed program is verified by the MaC verifier. An *event recognizer* detects primitive events and changes to primitive conditions from low-level information received from the filter. The primitive events are variable updates, method calls, and method returns. The primitive conditions are predicates over variables in the target program. These events and conditions are then sent to a *runtime checker*, which determines whether or not the current execution history satisfies the requirement specification. The execution history is captured from a sequence of events sent by the event recognizer. If the checker detects any violation, it notifies the user.

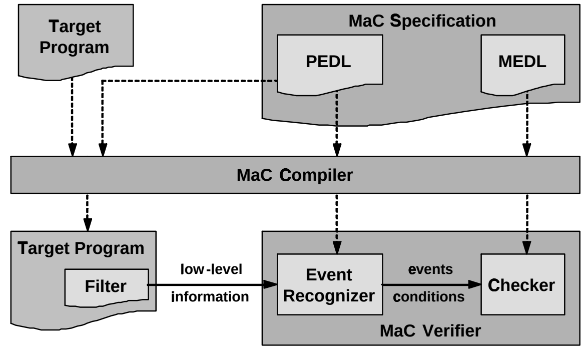

Figure 1: Overview of the MaC architecture

3.2 Meta-Event Definition Language (MEDL)

MaC provides two languages, PEDL and MEDL, shown in Fig. 1. The requirement specification or Meta-Event Definition Language (MEDL), based on a linear temporal logic (LTL) [14], allows one to express qualitative properties. A monitoring script or Primitive Event Definition Language (PEDL), defines which application-dependent information is extracted, and how it is transformed into events and conditions used in MEDL. PEDL is tied to a particular implementation while MEDL is independent of an implementation. In this paper, we do not discuss PEDL. See [8] for details.

3.2.1 Events and Conditions.

The underlying foundation for MEDL is the logic of events and conditions [8]. It deals with two kinds of objects, *events* and *conditions*. Events occur instantaneously during a system execution, whereas conditions represent system states that hold for a duration of time. For example, an event denoting a call to method *init* occurs at the instant the control is passed to the method, while a condition $v < 5$ holds as long as the value v does not exceed 5. The syntax of events and conditions is shown below where e and c represent primitive events and primitive conditions described in Section 3.1, respectively.

$$\begin{array}{lcl} E & ::= & e \mid E\&\&E \mid E||E \mid start(C) \mid end(C) \mid E\ when\ C \\ C & ::= & c \mid !C \mid C\&\&C \mid C||C \mid C \rightarrow C \mid defined(C) \mid [E, E) \end{array}$$

The boolean connectives used in events and conditions are defined in the usual way. $start(c)$ and $end(c)$ events occur when the condition c becomes true and false, respectively. $e\ when\ c$ event occurs if e occurs at the time when c is true. $defined(c)$ condition is true when the condition c is defined. $[e_1, e_2)$ is true from the time of an occurrence of e_1 until the first occurrence of e_2 after that. This condition $[e_1, e_2)$ is a variant of an *until* operator in LTL [14].

The model M for a MEDL formula is a time-stamped trace of observations, that is, a tuple $\langle S, \tau, L_C, L_E \rangle$, where $S = \{s_0, s_1, \ldots\}$ is a set of states, τ is a mapping from S to an absolute discrete time domain, L_C is a total function from $S \times \mathcal{C}$ to $\{true, false, \Lambda\}$ where $\mathcal{C}$ denotes a set of primitive conditions and Λ denotes undefined, and L_E is a partial function from $S \times \mathcal{E}$ to a value domain where $\mathcal{E}$ denotes a set of primitive events. M specifies, for each time instance, the value of each condition and the set of events that occur at that moment. The semantic definition is given in a form of the function $M, t \models \phi$, where ϕ is an event or condition and t is a time instance. The function $\models$ relies on a mutually recursive (but well-defined) definition of a denotation for conditions $\mathcal{D}^t_M(c)$ that assigns to a condition c a value drawn from the set $\{true, false, \Lambda\}$. For formal semantics of events, conditions, see [9].

4. RT-MEDL EXTENSION OF MAC SPECIFICATION LANGUAGE

This section describes an extension of the MaC language called RT-MEDL and introduces time-bound conditions for expressing quantitative properties.

4.1 Syntax

RT-MEDL adds time-bound conditions $[E, E)_{\{\leq d\}}$, $[E, E)_{\{<d\}}$, and $[E, E)_{\{=d\}}$ for expressing quantitative properties. The condition $[e_1, e_2)_{\{\leq d\}}$ indicates that an event e_2 must occur after an occurrence of e_1 within d time units. The conditions $[e_1, e_2)_{\{<d\}}$ and $[e_1, e_2)_{\{=d\}}$ have similar meanings. Here, d is a non-negative constant. The resulting language is similar to Metric Temporal Logic (MTL) [10]. MTL has a time-bound *until* operator that allow one to specify a time bound where a given temporal property must hold. Just as the condition $[e_1, e_2)$ is a variant of an *until* operator in LTL [14], the conditions $[e_1, e_2)_{\{\leq d\}}$, $[e_1, e_2)_{\{<d\}}$, and $[e_1, e_2)_{\{=d\}}$ are variants of a time-bound *until* operator in MTL [10].

4.2 Semantics

The semantics of the quantitative conditions uses the model presented in Section 3. A model M models a condition c when the value of $\mathcal{D}^t_M(c)$ has a $true$ value. $\mathcal{D}^t_M(c)$ represents a value of a condition c at time t. Formally, $M, t \models c$ iff $\mathcal{D}^t_M(c) =$ true. The notation $\mathcal{D}^t_M(c)$ for the new conditions is presented below.

$$\mathcal{D}^t_M([e_1, e_2)_{\{\leq d\}}) = \begin{cases} \Lambda & \text{if } \nexists t' < t \text{ s.t. } M, t' \models e_1 \\ false & \text{if } M, t-d \models e_1 \text{ and} \\ & \forall t' : t-d \leq t' \leq t \text{ s.t. } M, t' \not\models e_2 \\ true & \text{otherwise} \end{cases}$$

$$\mathcal{D}^t_M([e_1, e_2)_{\{< d\}}) = \begin{cases} \Lambda & \text{if } \nexists t' < t \text{ s.t. } M, t' \models e_1 \\ false & \text{if } M, t-d \models e_1 \text{ and} \\ & \forall t' : t-d \leq t' < t \text{ s.t. } M, t' \not\models e_2 \\ true & \text{otherwise} \end{cases}$$

$$\mathcal{D}^t_M([e_1, e_2)_{\{= d\}}) = \begin{cases} \Lambda & \text{if } \nexists t' < t \text{ s.t. } M, t' \models e_1 \\ false & \text{if } M, t-d \models e_1 \text{ and } M, t \not\models e_2 \\ true & \text{otherwise} \end{cases}$$

The condition $[e_1, e_2)_{\{\leq d\}}$ is undefined when an event e_1 has never occurred in the system. When e_1 occurs, it stays true unless the event e_2 does not occur within d time units. If e_1 occurs but e_2 does not occur within d time units, $[e_1, e_2)_{\{\leq d\}}$ becomes false. The conditions $[e_1, e_2)_{\{<d\}}$ and $[e_1, e_2)_{\{=d\}}$ are defined similarly. An example of a quantitative property *"real-time tasks must execute within time limits"* can be expressed using the quantitative condition as $[startT, endT)\{\leq 80\}$. The events $startT$ and $endT$ are events indicating the start and the end of an execution of a task T, respectively. The quantitative condition indicates $endT$ must occur after $startT$ within 80 time units.

5. IMPLEMENTATION

The RT-MaC system is obtained by extending the existing MaC implementation. We first modify parts of the MaC system, which is currently implemented in Java, to Real-Time Java. We then add a time-driven evaluation for quantitative properties to the Real-Time Java RT-MaC system.

5.1 Existing Implementation

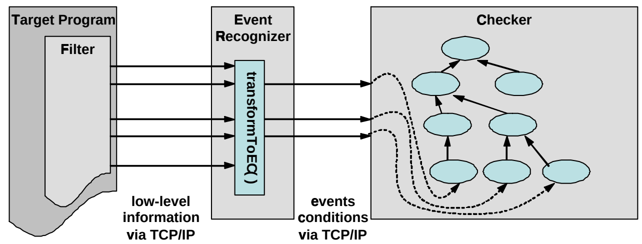

Figure 2: MaC Implementation

The current implementation for the MaC system shown in Figure 2 operates in an on-line fashion, that is, without storing the trace of observations. The target program, the event recognizer, and the checker, each running on a separate Java program, communicate with one another via a TCP/IP socket[1]. The target program sends its observation with an absolute timestamp in milliseconds to the event recognizer. The event recognizer, then, transforms it to an abstract observation in terms of events and conditions and forwards them to the checker. Upon receiving the new abstract observation, the checker updates the values of conditions and occurrence times of events. Without new observations, the checker remains idle. Then, the observation can be discarded and the algorithm proceeds to the next observation. The simplified algorithm is presented in Figure 3.

Event Recognizer:
```
while (true) {
  observe = receiveTarget();
  abstractObserve = transformToEC(observe);
  sendChecker(abstractObserve);
}
```
Checker:
```
while (true) {
   abstractObserve = receiveER();
   evaluate(abstractObserve);
}
```

Figure 3: MaC Implementation in Java

The main data structure used by the checker shown in Figure 2 is the property graph of dependencies between events, conditions, and auxiliary variables used in the formula. The nodes in the property graph keep the necessary information about the values of the conditions or variables, or the

[1]See[8] for the reasons why we choose to run the MaC system on a different machine.

timestamp of the last event. The algorithm acyclically traverses the property graph, starting from the nodes corresponding to primitive events and conditions, traversing up to the root updating the values in the nodes along the path. Other events and conditions not on the path are unrelated to the new observation and are not evaluated. Such algorithm makes the checking more efficient.

5.2 Real-Time Java Implementation

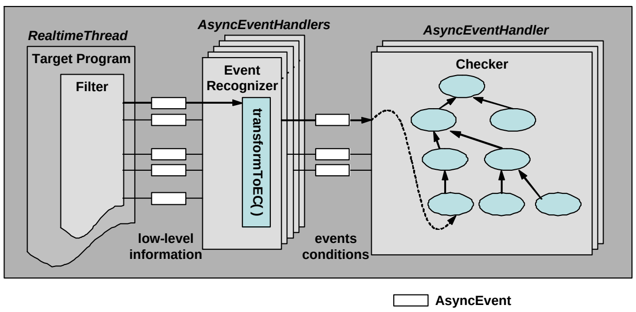

Figure 4: MaC Implementation in Real-Time Java

For the Real-Time Java implementation, the three components, 1) the target program with the filter, 2) the event recognizer, and 3) the checker, are in the same Java program. The target program with the filter is implemented using RealtimeThread while the event recognizer and the checker are implemented using EventSyncHandler. Figure 4 illustrates the overview of RT-MaC in Real-Time Java whereas Figure 5 provides the RT-MaC written in Real-Time Java sourcecode. The three components communicate via shared variables and asynchronous events. The Real-Time Java implementation works as follows. At the instrumentation point, the target program fires an asynchronous event *lowLevelAsyncEvent*, which is handled by an asynchronous event handler *eventRecognizer*. When *lowLevelInfo* is fired, *eventRecognizer* is invoked to read the low level information, transform them into events and conditions, and fire *abstractAsyncEvent*. When *abstractAsyncEvent* is fired, *checker* is invoked to evaluate properties. The evaluation by traversing upward on the property graph remains the same.

5.3 Time Driven Evaluation

When designing the implementation for quantitative properties, we would like to evaluate quantitative properties 1) correctly, 2) as soon as possible, and 3) with minimal overhead. With the three goals in mind, this section explores some implementation options for evaluating quantitative properties.

Most runtime verification systems for quantitative properties, including EAGLE, the works by Thati and Rosu [17] and by Barringer et al. [2], use event-driven evaluation. Event-driven evaluation evaluates properties in response to computation changes. While event-driven evaluation evaluates correctly and with minimal overhead, it might not be done as soon as possible as we shall see in the following example. Consider an autopilot and a tracking system where a property to be checked is "*if the plane crosses to an enemy territory, the tracking system must notify the autopilot within d ms*" written in RT-MEDL as *[enterEnemyT, notifyAutopilot)*$_{\{\leq d\}}$. Assume s_i is a state where

```
/** MaC **/
AsyncEvent lowLevelAsyncEvent = new AsyncEvent();
AsyncEvent abstractAsyncEvent = new AsyncEvent();
AsyncEventHandler eventRecognizer =
                new AsyncEventHandler() {
 public void handleAsyncEvent() {
  observe = readTarget();
  abstractObserve = transformToEC(observe);
  writeChecker(abstractObserve);
  abstractAsyncEvent.fire();
 }
}
AsyncEventHandler checker =
                new AsyncEventHandler() {
 public void handleAsyncEvent() {
  abstractObserve = readER();
  evaluate(abstractObserve);
 }
}
lowLevelAsyncEvent.addHandler(eventRecognizer);
abstractAsyncEvent.addHandler(checker);

/** a method added to a target program **/
public instrument(Info i) {
 writeTarget(i);
 lowLevelAsyncEvent.fire();
}
```

Figure 5: Real-Time Java RT-MaC

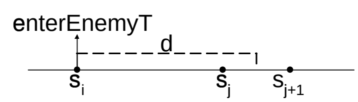

Figure 6: Event-Driven Evaluation

enterEnemyT occurs, and s_j is the last state before the time bound d is up. More formally, s_j and s_{j+1} are states where $i \leq j$ and $\tau(s_j) < \tau(s_i)+d$ and $\tau(s_{j+1}) \geq \tau(s_i)+d$. Fig. 6 illustrates the situation. If *notifyAutopilot* occurs at any state between s_i and s_j, then no violation occurs. Otherwise, at s_{j+1}, the value of *[enterEnemyT, notifyAutopilot)*$_{\{\leq d\}}$ must be changed to false and the violation must be reported so that the plane can be manually steered out of the enemy territory. The problem arises when s_{j+1} occurs much later than s_j. In such case, the violation would not be reported promptly and passengers on the plane could be in great danger. In the worst case, s_{j+1} could be the state where the plane is shot down!

To solve the problem, we propose time-driven evaluation. Time-driven evaluation evaluates properties in response to time changes in addition to events or computation changes. There are two options for time-driven evaluation.

1. **Heart Beat**: Evaluate quantitative properties at every single time step, i.e., every "heart beat". This time step could be large or small and can be defined by a user to suit their needs.

2. **Timeout**: Evaluate quantitative properties when a timeout signal is triggered indicating the time bound d

is up. The timeout signal would initiate the evaluation and update the values of properties as they change. One way to signal a timeout is to use a timer. We set a delay in which we want the timer to run. This delay is a time bound specified in the quantitative conditions. When the timer expires, it would send a signal to the verifier and initiate the evaluation.

For the rest of this section, we assume a quantitative property to be verified is $[e_1, e_2)_{\{\leq d\}}$. The property is considered violated if e_2 does not occur within d after e_1 occurs. The evaluation for $[e_1, e_2)_{\{<d\}}$ and $[e_1, e_2)_{\{=d\}}$ can be adapted straightforwardly.

The heart beat method could be implemented easily by setting a periodic timer according to the given user-defined time step. Whenever the timer expires, quantitative properties are evaluated. Although the implementation is clean and simple, evaluating the properties at every time step can be redundant and inefficient, especially for time steps that do not change any values of quantitative properties. This method would only work when choosing the time step properly, otherwise the evaluation will be done unnecessarily. If the steps are too small, this method can be evaluated too often and cause large overhead. If the steps are too big, the evaluation might not be done as soon as possible.

In the timeout method, a one-time timer with a deadline of d is set whenever e_1 occurs. If e_2 occurs before the timer expires, the property is satisfied. If the timer expires before e_2 occurs, the property is violated. It is efficient because this method evaluates only when the time change induces the change of quantitative property values. Therefore, this method always reaches the third implementation goal of evaluating with minimal overhead. It however can get complicated because of communication delays between the target program, the event recognizer, and the checker. Because of the communication delays, there might exist a data race between events occuring in the target program and the timer expiring interrupt. Therefore, the deadline cannot be set too early because e_2 occurring before d might be evaluated after the timer expires and cause the property evaluation to result in violation, which is incorrect and therefore not meeting our first implementation goal of evaluating correctly. If the deadline is too large, the violation is not reported soon enough, violating the second implementation goal of evaluating as soon as possible.

Section 5.3.1 investigates in details how communication delays can cause data race and in turn violate the first and second implementation goals which are 1) incorrectly evaluated or 2) not evaluated soon enough. Section 5.3.2 provides a simple trick to avoid such data race.

5.3.1 Data Race Problem

Assume e_1 occurs at t_1 and the filter starts communication of an event (e_1, t_1) to the checker. The checker then receives and processes the message (e_1, t_1^p) at its local time of $t_1 + t_{comm}$. After the checker processes e_1, it sets a timer with different deadlines depending on different cases. Let t_{comm} be a communication delay between the filter and the checker. We consider three cases and address different problems caused by the data race.

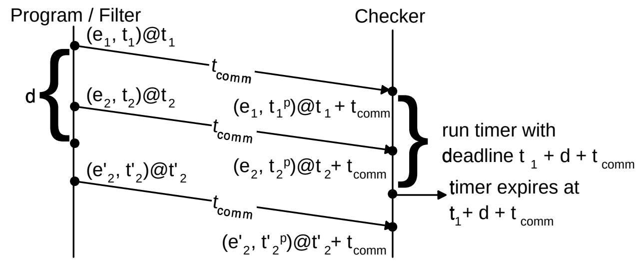

Figure 7: Case 1: Constant t_{comm}

5.3.1.1 Case 1: Constant t_{comm}.

In our first case, shown in Fig. 7, we assume that the communication delay t_{comm} is constant. When e_1 occurs at time t_1, the filter relays an event message (e_1, t_1) at time t_1. The checker would process the message (e_1, t_1) at its local time of $t_1 + t_{comm}$. The checker then starts the timer and sets the deadline to $t_1 + d + t_{comm}$.

Based on Assumption 6, Assumption 7, and that the event message sent at t_1^p by the filter is received and processed at t_1^c by the checker, since the communication delay t_{comm} is always constant, any event messages sent between t_1 and $t_1 + d$ must arrive and be processed between $t_1 + t_{comm}$ and $t_1 + dt_{comm}$. Therefore, they are processed before the timer expires. On the contrary, any event messages sent after $t_1 + d$ are processed after the timer has expired. Thus, when e_2 occurs at t_2 where $t_1 \leq t_2 \leq t_1 + d$, it is processed at t_2 where $t_1 + t_{comm} \leq t_2 + t_{comm} \leq t_1 + d + t_{comm}$, which is before the timer expires. Consequently, the checker interprets that the property is satisfied. On the other hand, when e_2' occurs and is sent at t_2' where $t_2' > t_1 + d$, the checker receives and processes the message at $t_2' + t_{comm}$ where $t_2' > t_1 + d + t_{comm}$, which is after the timer expiring event has been processed. The checker therefore interprets that the property is not satisfied, which means it also correctly checks the program. It therefore correctly checks the program satisfying the first implementation goals. Moreover, the checker would notify the user of this quantitative violation at time $t_1 + d + t_{comm}$ and therefore, the violations are reported with only t_{comm} delay and satisfying the second implementation goal of evaluation as soon as possible. Hence, there is no data race in this case.

5.3.1.2 Case 2: Bounded t_{comm}.

In this case shown in Figure 8, we assume t_{comm} is bounded by b. Therefore, when the checker processes e_1, we can set the deadline to $t_1 + d + b$. The reason is shown as follows.

Assume e_1 occurs at t_1. Let c_1 be transmission and processing time for e_1 and $0 < c_1 \leq b$. Hence, e_1 is processed by the checker at time $t_1 + c_1$. Let c_2 be defined similarly for some event occurring at $t_1 + d$. Such event should arrive at the checker by $t_1 + d + c_2$. Thus, the time between processing e_1 and processing any event occurring at $t_1 + d$ is $(t_1 + d + c_2) - (t_1 + c_1) = d + (c_2 - c_1)$. Since $0 < c_1 \leq b$ and $0 < c_2 \leq b$, the maximum value of $(c_2 - c_1)$ is $(b - 1)$

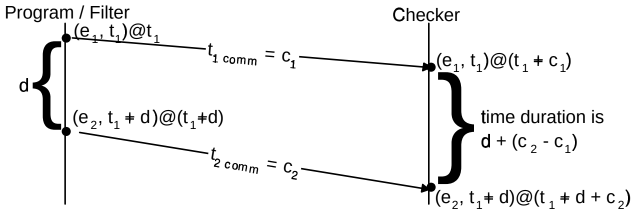

Figure 8: Case 2: Bounded t_{comm}

when $c_2 = b$, $c_1 = 1$. Hence, the maximum duration for any events occurring before $t_1 + d$ to process at the checker is $d+b$. Since the checker processes e_1 at t_1 and the maximum duration is $d + b$, we can set a deadline to $t_1 + d + b$. Such deadline assures any events occurring before $t_1 + d$ will be processed before the timer expires.

With this bound, it is still possible that events processed before the timer expires actually occur after $t_1 + d$. The checker can compare the timestamp t_1 of e_1 to the timestamp t_2 of e_2 and check if $t_1 + d \geq t_2$. If so, the quantitative property is satisfied. Otherwise, the quantitative property is violated. However, the bound b assures that events occurring before the time $t_1 + d$ will be processed before the timer expires at t_1+d+b. Therefore, when the timer expires and the property is evaluated to violated, the checker can be sure that e_2 processed after the timer expires cannot occur before $t_1 + d$ and that the evaluation is correct.

5.3.1.3 Case 3: Nondeterministic t_{comm}.

The communication time t_{comm} can vary nondeterministically depending on workload and a number of events and conditions on the system machine. In this case, we assume that t_{comm} is nondeterministic. Based on this assumption, we cannot be certain that events processed before the timer expires actually occur before $t_1 + d$. Figure 9 (a) illustrates this situation. Assume we set the same deadline as Case 1, i.e., $t_1 + d + t_{1\ comm}$. Assume e'_2 occurs after $t_1 + d$. It is possible that $t_{1\ comm}$ might be so large that the timer is started late and $t'_{2\ comm}$ might be so small that the message (e'_2, t'_2) is processed before the timer expires. If this is the case, the checker can compare the timestamp t_1 of e_1 to the timestamp t'_2 of e'_2 and check if $t_1 + d \geq t'_2$. If so, the quantitative property is satisfied. Otherwise, the quantiative property is violated. By analogy, we cannot be certain that the events processed after the timer expires actually occur after $t_1 + d$. Figure 9 (b) illustrates this situation. It is possible that $t_{1\ comm}$ might be so small that the timer expires early and $t_{2\ comm}$ might be so large that the message (e_2, t_2) arrives after the timer expires. If this is the case, when the timer expires, the checker must have decided that the quantitative propery is violated. This, however, is not correct. In this case, no matter what the deadline the timer is set to, the checker can behave nondeterministically. Therefore, when the checker raises violations, we can never be certain whether or not the checker evaluates the quantitative property correctly.

While the first and the second case have no data race problem and provide all implementation goals of evaluating correctly, as soon as possible and with minimal overhead, they require strictly constant t_{comm} or bounded t_{comm} and may not work in practice. The last case when we relax the assumptions to fit what would occur in practice, the data race between events handled by the filter and timer expiring interrupt handled by the checker can occur and properties may not be evaluated correctly.

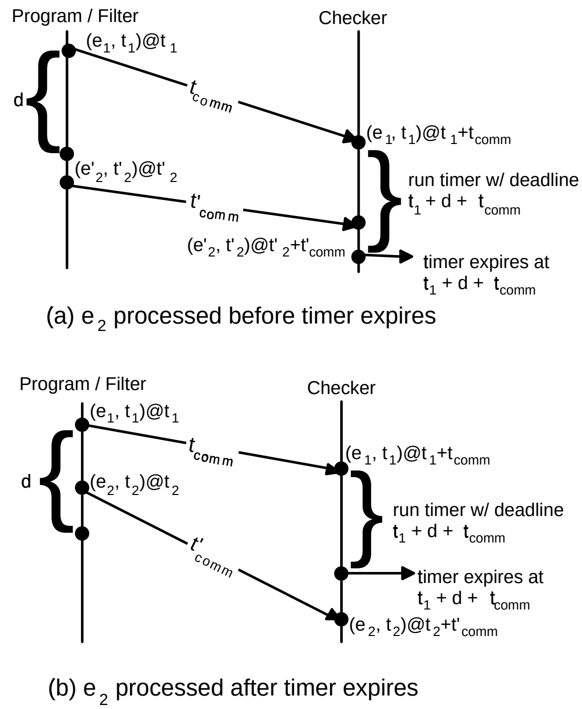

Figure 9: Case 3: Nondeterministic t_{comm}

5.3.2 Solution

This section we present a solution, shown in Figure 10 to the data race problem addressed in the last section when t_{comm} is nondeterministic. We create another an asynchronous event *TimerAsyncEvent*. After the checker processes and evaluates the event message (e_1, t_1), it sets a timer the deadline t_1+d. When the timer expires, it fires $TimerAsyncEvent$, which would be handled by the filter. When $TimerAsyncEvent$ is fired, the filter will create a timing expiring message with its current timestamp and notify the event recognizer, which in turn notifies the checker. Since both events and the timing expiring is handled by the filter, there is no data race between them.

It is possible that an event e'_2 occurring after t_1+d arrives at the checker before timer expiring message because of nondeterministic communication delay. In such cases, the checker can compare the timestamp t_1 of e_1 to the timestamp t'_2 of e'_2 and check if $t_1+d \geq t'_2$. If so, the quantitative property is satisfied. Otherwise, the quantitative property is violated. After the quantitative property is evaluated, either satisfied or violated, the checker would ignore the associated timer expiring message. On the other hand, this timer expiring message can be used to indicate that any other events arriving after the checker has processed the timer expiring message occur after $t_1 + d$. Therefore, if the timer expiring message is received but no e_2 occurs, the checker can correctly evaluate the property as violated.

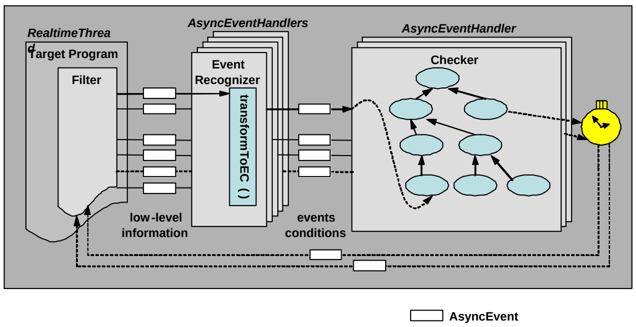

Figure 10: MaC in Real-Time Java with Timer

The analysis shows that when the filter runs the timer, the checker can evaluate quantitative properties correctly even if the communication is nondeterministic. Thus, if the communication delay is known to be bounded, the checker certainly can evaluate quantitative properties correctly. Since the solution accomplish all the implementation goals of evaluating correctly, as soon as possible, and with minimal overhead, it is the best approach for time-driven evaluation.

6. CONCLUSION AND FUTURE WORK

The contributions of the paper are threefold. First, we extend the MaC specification language to support quantitative properties for verifying timing correctness of a real-time system. The resulting language is similar to Metric Temporal Logic (MTL) [10]. Second, we address the problem and provide an efficient solution when evaluating in a time-driven approach. The time-driven evaluation enables the verifier to efficiently update values of time-bounded operators as their values change instead of waiting for event occurrences. Lastly, we present how we could implement such system in Real-Time Java. The future work includes examining overhead and interference incurred by the RT-MaC and experimenting the RT-MaC with real systems.

7. REFERENCES

[1] R. Alur and T. A. Henzinger. A Really Temporal Logic. *Journal of the ACM*, 41:181–2004, 1994.

[2] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proc. of 5th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 44–57, Vanice, Italy, 2004.

[3] D. Bartetzko, C. Fischer, M. Mller, and H. Wehrheim. Jass - Java with Assertions. In *Proc. of the 1st International Workshop on Run-time Verification*, 2001.

[4] D. Drusinsky. Monitoring Temporal Logic Specifications Combined with Time Series Constraints. *Journal of Universal Computer Science*, 9(11), Nov 2003.

[5] A. Gates, S. Roach, O. Mondragon, and N. Delgado. DynaMICs: Comprehensive Support for Run-Time Monitoring. In *Proc. of the 1st International Workshop on Run-time Verification*, 2001.

[6] K. Havelund and G. Rosu. Java PathExplorer – A Runtime Verification Tool. In *Proc. of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2001.

[7] J. Jayaputera, I. Poernomo, and H. Schmidt. Runtime Verification of Timing and Probabilistic Properties using WMI and .NET. In *Proc. of the 30th EUROMICRO Conference*, 2004.

[8] M. Kim. *Information Extraction for Run-time Formal Analysis*. PhD thesis, University of Pennsylvania, 2001.

[9] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a Run-time Assurance Approach for Java Programs. *Formal Methods in Systems Design*, 24(2):129–155, Mar 2004.

[10] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real Time Systems*, 2(4):255–299, 1990.

[11] K. J. Kristoffersen, C. Pedersen, and H. R. Anderson. Event-Based Runtime Checking of Timed LTL. Technical report, IT University of Copenhagen, Nov 2003.

[12] K. J. Kristoffersen, C. Pedersen, and H. R. Anderson. Runtime Verification of Timed LTL Using using Disjunctive Normalized Equation Systems. In *Proc. of the 2nd International Workshop on Run-time Verification*, 2003.

[13] C. G. Lee, A. K. Mok, and P. Konana. Monitoring of Timing Constraints with Confidence Threshold Requirements. In *IEEE Real-Time Systems Symposium*, pages 178–187, 2003.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[15] A. K. Mok, C. G. Lee, H. Woo, and P. Konana. Monitoring of Timing Constraints on Time Intervals. In *IEEE Real-Time Systems Symposium*, pages 191–200, 2002.

[16] A. K. Mok and G. Liu. Efficient Run-Time Monitoring of Timing Constraints. In *Proc. of the 3rd IEEE Real-Time Technology and Applications Symposium*, pages 252–262, June 1997.

[17] P. Thati and G. Rosu. Monitoring Algorithm for MTL Specification. In *Proc. of the 2nd International Workshop on Run-time Verification*, 2004.