August 2005

RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties

Usa Sammapun
University of Pennsylvania, usa@cis.upenn.edu

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

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

Copyright 2005 IEEE. Reprinted from *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005)*, pages 147-153.

RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties

Abstract

Correctness of a real-time system depends on its computation as well as its timeliness and its reliability. In recent years, researches have focused on verifying correctness of a real-time system during runtime by monitoring its execution and checking it against its formal specifications. Such verification method is called Runtime Verification. Most existing runtime verification tools verify computation correctness using qualitative property specifications but do not verify timeliness nor reliability correctness. In this paper, we investigate the verification on timeliness and reliability correctness by offering quantitative and probabilistic property specifications and implementing efficient verifiers.

RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties

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

Abstract

Correctness of a real-time system depends on its computation as well as its timeliness and its reliability. In recent years, researches have focused on verifying correctness of a real-time system during runtime by monitoring its execution and checking it against its formal specifications. Such verification method is called Runtime Verification. Most existing runtime verification tools verify computation correctness using qualitative property specifications but do not verify timeliness nor reliability correctness. In this paper, we investigate the verification on timeliness and reliability correctness by offering quantitative and probabilistic property specifications and implementing efficient verifiers.

1 Introduction

Correctness of a real-time system depends on its computation as well as its timeliness and its reliability. The need to consider all computation, timeliness, and reliability 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 [5, 6, 10], 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. It is also based on formal logics and provides formalism in which testing is lacking. Runtime verification can be used both online to ensure property correctness of a given application or during development to find bugs.

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 on timeliness nor reliability correctness. One of such tools is MaC (Monitoring and Checking) [9, 10]. MaC verifies computation correctness of a system by providing qualitative property specification based on Linear Temporal Logic (LTL) [15]. 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 and reliability correctness by providing quantitative and probabilistic property specifications and call the extension RT-MaC. The additional quantitative and probabilistic property specification is achieved by introducing time-bound temporal operators and probabilistic operators, respectively. The time-bound temporal operators can specify a time limit in which a property must hold, and therefore is more appropriate for specifying quantitative properties. The resulting language is similar to Continuous Stochastic Logic (CSL) [2].

There exist a few runtime verification works aiming at 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 and time changes as "event-driven" and "time-driven", respectively. In this paper, we propose a solution to correctly implement time-driven evaluation. For probabilistic properties, RT-MaC uses statistical analysis to provide confidence intervals in verifying probabilistic properties. Such confidence intervals are not included in most runtime verification systems.

The paper is organized as follows. Section 2 provides background on MaC. Section 3 describes the RT-MaC language, which provides additional operators for quantitative and probabilistic properties. Section 4 describes implementation and its overhead. Section 5 provides related work. Section 6 concludes the paper and presents future work.

Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)
1533-2306/05 $20.00 © 2005 **IEEE**

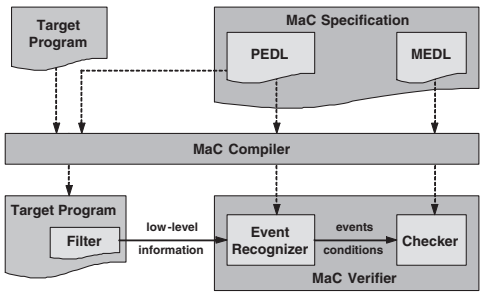

Figure 1. Overview of the MaC architecture

2 MaC Overview

2.1 MaC Architecture

MaC has been developed to ensure that a program runs correctly with respect to its formal specification. Fig. 1 shows the overall MaC architecture. It works as follows. A user specifies a specification of a target program in a formal MaC language. Given the target program and the specification, MaC inserts probes or a *filter* into the program to extract observations such as assignments to program variables and method calls and returns. The specification is compiled into a MaC verifier specialized for the 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. Primitive events are variable updates, method calls and returns. Primitive conditions are predicates over program variables. These events and conditions are then sent to a *runtime checker*, which determines whether or not the events and conditions satisfy the program's specification. If the checker detects any violation, it notifies the user.

2.2 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) [15], 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 any implementation. We do not discuss PEDL here. See [9] for details.

Events and Conditions. The underlying foundation for MEDL is the logic of events and conditions [9]. 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 conditions.

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

The boolean connectives used in events and conditions are defined in the usual way. $start(c)$ and $end(c)$ events occur when a 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 a 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 [15]. MEDL distinguishes special events and conditions denoting application-level requirements. Safety properties are conditions that must be *always* true whereas alarms are events that must *never* be raised.

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 [10]. In the next section, we describe the RT-MEDL extension of MEDL.

3 RT-MEDL Extension

3.1 Quantitative Specification

3.1.1 Syntax

RT-MEDL provides time-bound conditions $[E, E)_{\{\leq d\}}$, $[E, E)_{\{<d\}}$, and $[E, E)_{\{=d\}}$ for quantitative properties. $[e_1, e_2)_{\{\leq d\}}$ indicates an event e_2 must occur after an event e_1 within d time units. $[e_1, e_2)_{\{<d\}}$ and $[e_1, e_2)_{\{=d\}}$ have similar meanings. Here, d is a non-negative constant. With these quantitative operators, the language is similar to Metric Temporal Logic (MTL) [11]. MTL has a time-bound $until$ operator allowing 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 [15], 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 [11].

3.1.2 Semantics

The semantics of the quantitative conditions uses the model presented in Section 2. A model M models a condition c when the value of $\mathcal{D}^t_M(c)$ is $true$. $\mathcal{D}^t_M(c)$ represents a value of a condition c at time t. Formally, $M, t \models c$ iff $\mathcal{D}^t_M(c)$ = true. $\mathcal{D}^t_M(c)$ for the new conditions is presented below.

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

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

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

$[e_1, e_2)_{\{\leq d\}}$ is undefined when e_1 has never occurred. When e_1 occurs, it stays true unless e_2 does not occur within d. If e_1 occurs but e_2 does not occur within d time units, $[e_1, e_2)_{\{\leq d\}}$ becomes false. $[e_1, e_2)_{\{<d\}}$ and $[e_1, e_2)_{\{=d\}}$ are defined similarly. An example, *"real-time tasks must execute within time limits"*, can be expressed using a 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. The condition indicates *endT* must occur after *startT* within 80 time units.

3.2 Probabilistic Properties

Probabilistic correctness is another important aspect of reliable real-time systems. Verifying probabilistic properties in model checking can be done using numerical or statistical techniques [22]. The numerical technique needs a complete probabilistic model of a system to verify probabilistic properties. However, in runtime verification, we have no such information and thus cannot use such technique. Instead, we adopt the statistical technique which simulates, samples execution paths, and calculates probabilities from the paths. One difference between statistical probabilistic checking in model checking and in runtime verificaition is runtime verification uses execution paths from an actual target system and hence has no control over its samples whatsoever. The statistical probabilistic model checking, on the other hand, can control simulation to produce execution paths as needed.

Another difference is instead of collecting hundreds different execution paths as in statistical probabilistic model checking, we only have one execution path. In order to collect multiple samples from one execution path, it requires a target system with repeating or periodic behaviors such as soft real-time schedulers, network protocols or web servers. Each repeated behavior is used as a sample execution, called an experiment. Experiments, for example, are flipping a coin or executing a soft real-time task. The probability of experiments being successful are defined by a ratio of successful experiments and a total number of experiments. Successful experiments, for example, are a coin being head when flipped or a soft real-time task finishing within its deadline. Such successful experiments can be thought of as a property being satisfied. The ratio however cannot be used as is but needs statistical analysis to support its result.

Given a set of experiments and a probabilistic property, RT-MaC statistically determines whether a system satisfies the property by using statistical hypothesis testing. Statistical hypothesis testing provides a systematic procedure with an adequate level of confidence in determining a probability of successful experiments. Because RT-MaC observes experiments online, a particular statistical hypothesis testing called sequential hypothesis testing [20] is appropriate since it determines a decision in real-time as we observe data and provides answers quickly within a given error bound. The sequential hypothesis testing depends on an outcome of previous testing. After each experiment, the testing can say the probabilistic property is either satisfied, not satisfied, or needing more experiments. In case of early termination, the verifier would say satisfied or not satisfied, both with a quantitative measure of confidence in the answer.

3.2.1 Syntax

RT-MEDL adds one probabilistic event $e\ prob(\odot p, e_{exp})$ and one probabilistic condition $c\ prob(\odot p, c_{exp})$ where $\odot \in \{<, >, \leq, \geq, =, \neq\}$ and p is a probability. e_{exp} and c_{exp} are experiments for the probabilistic event and the probabilistic condition, respectively. e_{exp} and c_{exp} are necessary because we need to identify repeated behaviors and collect them as experiments from one arbitrary execution path. For example, a probabilistic alarm *"a soft real-time task must not miss a deadline of 80 time units with probability $\geq$ 0.2"* can be written as $end([startT, endT)\{\leq 80\})prob(\geq\ 0.2, startT)$. The formula states an alarm should be raised when a task misses its deadline with a probability ≥ 0.2 where an event $startT$ is an experiment. Another example is *"a car velocity must be less than 50mph with probability ≥ 0.9 in work zones, and otherwise must be less than 50mph with probability ≥ 0.8"* and can be written as a property $(v < 50)prob(\geq\ 0.9, work)\&\&(v < 50)prob(\geq\ 0.8, true)$. The condition $work$ is true only

when a car is in a work zone. The formula states when $work$ is true, $(v < 50)$ must hold true with probability ≥ 0.9, and at all times, $(v < 50)$ with probability ≥ 0.8. This syntax with time-bound conditions, a probabilistic event and condition is similar to Continuous Stochastic Logic (CSL) [2].

3.2.2 Semantics

We describe semantics for probabilistic properties by first introducing a model for experiments and successful experiments, and then, laying out how to use sequential hypothesis testing to statistically determine satisfiability.

Model. Let a probabilistic property we want to verify is $\phi\ prob\ (\odot p_0, X)$ where $\odot \in \{<, \leq, >, \geq, =, \neq\}$, ϕ is either an event or a condition, p_0 is a probability, and X is an experiment. When an event ϕ is satisfied, it means ϕ occurs. When a condition ϕ is satisfied, it means ϕ is true. Let n be a number of experiments performed during execution. Let $X = X_1 + X_2 + ... + X_n$ be a random variable representing a number of successful experiments. Let X_i be a random variable representing a result of the i^{th} experiment where $X_i = 1$ when the i^{th} experiment satisfies ϕ (successful experiment), and $X_i = 0$ otherwise. Thus, each X_i has a Bernoulli distribution with an unknown parameter $p \in [0, 1]$ where $Pr(X_i = 1) = p$, meaning the probability of the i^{th} experiment being successful is equal to p. X, therefore, has a Binomial distribution with parameters n and p. This value p is the probability of ϕ being satisfied and is what we want to test against p_0. Finally, let $\bar{p} = \frac{\Sigma X_i}{n}$ be an experimental probability obtained from the observed experiments.

Sequential Hypothesis Testing. To statistically check probabilistic properties, we use the experimental probability $\bar{p}$ to approximate the actual Binomial probability p based on a mathematically founded procedure of sequential hypothesis testing. The first step, done before running experiments, is to set up two hypotheses H_0 and H_1. H_0, called null hypothesis, is what we want to disprove, and H_1, called alternative hypothesis, is an alternative to H_0. The next step, done after m experiments, is to make one of the following three decisions: 1) accept H_0, 2) reject H_0, and 3) continue observing experiments. In our case, acceptance of H_0 means $M, t \not\models \phi\ prob\ (\odot p_0, X)$ and rejection of H_0 means $M, t \models \phi\ prob\ (\odot p_0, X)$. Thus, the number of experiments n is not determined in advance but depends on an outcome of the previous test analysis. This characteristic of sequential hypothesis testing ensure that we have a decision quickly.

Approximation of Binomial Probability. To accept or reject hypotheses, we calculate how far apart $\bar{p}$ is from p_0 and in what direction, whether greater or less than p_0. The *z-score* is a value that statisticians use for this purpose. The following equation is used to calculate the *z-score* for $\bar{p}$ [21].

$$z = \frac{\bar{p} - p_0}{\sqrt{\frac{\bar{p}(1-\bar{p})}{n}}}$$

In this case, $\bar{p}$ is greater than p_0 iff z is positive, and $\bar{p}$ is less than p_0 iff z is negative. When z is close to zero, there are two interpretations: 1) $\bar{p}$ is close to p_0, or 2) we do not have a sufficient number of experiments to decide, and we cannot differentiate between them. To differentiate between the two interpretations, we determine the minimum number of observations n_0 based on an error bound d. If the number of experiments is greater than n_0, then we can ensure that the value of z close to zero means that the value of $\bar{p}$ is close to p_0, not that we have an insufficient number of experiments. A confidence interval d indicates an indifferent zone. If $\bar{p}$ is between $p_0 - d\%$ and $p_0 + d\%$, it can be considered as either successful or unsuccessful because $\bar{p}$ is so close to p_0 and the result is considered indifferent. Let δ be a confidence level. If $Pr(|\bar{p} - p_0| = d) \geq \delta$, then n_0 is given as follows [4].

$$n_0 = \frac{z_\delta^2 p_0(1 - p_0)}{d^2}$$

Error Bound. The probabilistic checking is done within error bounds of α and β. α is the largest acceptable probability of incorrectly verifying a true property and β is the largest acceptable probability of incorrectly verifying a false property. Formally, $\alpha = Pr\{\text{reject } H_0 | H_0 \text{ is true}\}$ and $\beta = Pr\{\text{reject } H_1 | H_1 \text{ is true}\}$. The bounds α and β are used to calculate the *z-score* corresponding to α and β written as z_α and z_β, respectively. z_α and z_β provide thresholds of the *z-score* for $\bar{p}$ to make one of the three decisions in the sequential hypothesis testing.

Next, we present how to set up hypotheses and make a decision. Since inequality symbols are more or less symmetric, we consider $\phi\ prob\ (<\ p_0, X)$ and $\phi\ prob\ (=\ p_0, X)$.

1. $\boldsymbol{\phi\ prob\ (<\ p_0, X)}$ **:** We set up hypotheses as follows. $H_0 : p \geq p_0$ and $H_1 : p < p_0$. We accept H_0 iff $z \geq z_\beta$. We accept H_1 iff $z < z_\alpha$. We need more sample iff $z_\alpha \leq z < z_\beta$. If the program terminates before accepting any hypotheses, we return either accept or reject H_0 with a probability of error called *p-value* (a statistical term described in [7]). If after n_0 experiments have been performed and $z_\alpha < z < z_\beta$, then we are indifferent between accepting H_0 and H_1 because approximately $\bar{p} = p_0$. Saying the property is satisfied would produce an error less than the bound and therefore, we accept H_1.

2. $\boldsymbol{\phi\ prob\ (=\ p_0, X)}$ **:** We set up hypotheses as follows. $H_0 : p \neq p_0$ and $H_1 : p = p_0$. After we perform n_0 experiments, we make following decisions. Accept H_0 if $z < z_\alpha$ or $z > z_\beta$. Accept H_1 if $z_\alpha < z < z_\beta$. If we have

performed at least n_0, we can decide because the number of n_0 already guarantees for error bounds. If we have not performed at least n_0, we return the decision with *p-value* [7]. This rule in making a decision between acceptance and rejection, therefore, ensures that the decision is within error bounds.

4 Implementation and Overhead

4.1 Implementation

The checker for RT-MEDL is obtained by extending the existing checker for MEDL implemented in Java. We add time-driven evaluation for quantitative properties and a machinery to perform statistical analysis testing for probabilistic properties. The MEDL checker evaluates without storing a trace of observations. Its algorithm works as follows. A target system, an event recognizer, and a checker communicate via TCP/IP sockets[1]. The target system sends its observation with an absolute timestamp in milliseconds to the event recognizer, which then transforms the observation to an abstract form of events and conditions and forwards them to the checker. Upon receiving, the checker updates the values of conditions and event timestamps. If properties become false or alarms occur, the checker notifies violations to users. Then, the observation can be discarded and the algorithm proceeds to the next observation.

For RT-MaC, the checker provides a machinery to perform statistical analysis for probabilistic properties. It keeps track of the number of all experiments and successful experiments, calculates *z-scores* and error bounds, and evaluates probabilistic events and conditions accordingly.

For quantitative properties, in addition to evaluating upon receiving a new observation, the RT-MEDL checker also evaluates in response to a timeout from time-bound conditions. Such timeout is necessary when the next observation comes much later causing the time-bound conditions to be evaluated much later than when their values change. A simple timeout signal can initiate the evaluation and update the values of these conditions as soon as their values 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 delay expires, the timer would send a signal to the checker and initiate the evaluation. Setting a timer, however, is not as easy when the target system and the checker reside on different machines. The differences in their clock speeds and the message delay between them can complicate the process of setting a timer.

Quantitative conditions specify that two events must occur within time bounds with respect to the clock where the events occur. Thus, the timer must run at the same clock speed as the target system. Since the checker cannot provide such guarantee, we run the timer on the target system machine. We describe our solution using an example of evaluating a condition $[e_1, e_2)_{\{\leq d\}}$. After an event e_1 occurs with an absolute timestamp t, the checker sends a message telling the target system to set a timer with an absolute deadline of $t+d$. When the target system receives the message, it sets a timer with a deadline of $t + d$. When the deadline is met, it sends a message with its current timestamp to the checker. Upon receiving, the checker evaluates the property as follows. When an event e_2 occurs with a timestamp $\leq t+d$, the condition remains true. On the other hand, when an event e_2 occurs with a timestamp $> t + d$, the condition becomes false. When no e_2 occurs and the checker receives a timeout signal with a timestamp $> t+d$, the condition also becomes false.

One might question that the overhead of communicating via TCP/IP can delay the message and the target system might receive the deadline message that is less than its current time. However, this does not cause the checker to incorrectly evaluate quantitative properties. When the absolute deadline is less than the target system current time, the target system would send a message back to the checker right away with the timestamp of its current time. The delayed timeout signal only causes the checker to delay its evaluation but it does not alter any evaluation results.

4.2 Overhead

The overhead of our implementation depends on communication cost, and cost of running timers. For each experiment, we use three different machines. All of them have 1GB memory and are operated using SunOS 5.8.

Machine 1: Sun Enterprise 3000 with four 250Mhz UltraSPARC processor and 20% utilization.

Machine 2: Sun Blade 1000 with one 750Mhz UltraSPARC III processor and 10% utilization.

Machine 3: Sun Ultra Enterprise 4000 with eight 167Mhz UltraSPARC processors and 20 % utilization.

Experiment I: Communication Cost The only overhead that RT-MaC adds to the communication cost is the deadline messages exchanging between the checker and the filter. To get the pure communication cost, we set the time bound on a quantitative condition to 1, ensuring that once the filter receives the deadline message, it must send a reply back right away. At checker, we measure in milliseconds its round-trip traveling (rrt) time by taking the difference between the time right before the checker sends the message and the time when the checker receives the reply message. We run four experiments on different machine setups as shown in Fig. 1. We repeat the test 200 times and take the average for each experiment. As expected, when all are running on

[1] See[9] for the reasons why we choose to run the MaC system on a different machine.

Target System on Machine	Event Recognizer on Machine	Checker on Machine	Avg rrt Time
1	1	1	80.5 ms
1	2	2	213 ms
1	3	3	241 ms
1	2	3	244 ms
1	3	2	218 ms

Table 1. Average rrt times

Freq. of Extracting	RT-MaC	Run Timer	Avg Exec. Time	Overhead
50 ms	No	-	598 ms	-
50 ms	Yes	No	604 ms	1.003%
50 ms	Yes	Yes	607 ms	1.505%
100 ms	No	-	1095 ms	-
100 ms	Yes	No	1100 ms	0.457%
100 ms	Yes	Yes	1104 ms	0.822%

Table 2. Intrumentation cost

the same machine, its rrt time is the least. However, even when running on different machines, the rrt times are still very small, only about 0.2 second. Therefore, if no events occur but the value of a quantitative condition changes due to a timeout, the checker would evaluate the condition after only at most 0.2 second delay. It seems insignificant for a system where events do not occur frequently. For a system where events occur frequently, this delay would not matter because the condition would have been evaluated already in response to other event occurrences. For any messages from the checker with a deadline greater than the filter's current time, the delay would have been only half of the rrt time shown in the table. The reason is that the timer would be set with the difference between the message deadline and its current time, and thus, the overhead is only a one-way traveling time of sending an expiring message from the filter to the checker.

Experiment II: Instrumentation Overhead We evaluate the cost of extracting information from the target program by comparing the execution time of the program with and without RT-MaC. The experiment is measured in milliseconds where all components are on Machine 1. The reason is that we want the worst-case execution time when all the components are competing for CPU utilization. The test program is written such that it is extracted at different frequencies by letting program sleep for specified milliseconds. For tests with no timers, we set the time bound to 1 so that the target system would send the reply message back right away without setting a timer. For tests with timers, we set the time bound to 200 to ensure that the target system would run a timer. The overhead is the percentage that the instrumentation poses to the target program. The result in Fig. 2 shows that the instrumentation and the execution of timers pose very little execution time overhead to the target system.

5 Related Work

There are a few existing works for runtime verification of quantitative and probabilistic properties. Mok and Liu [17] present an efficient runtime monitoring mechanism to detect violations for timing constraints using graph theories. The successors of the paper [16, 14] present the monitoring of timing constraints on time intervals and timing constraints with confidence threshold requirements. Thati and Rosu [19] studied the monitoring algorithms for MTL [11]. Their algorithms evaluate by transforming MTL formulae into a "canonical form" after each occurrence of events. The transformed formulae specify what need to be held on the next state. The evaluation algorithm of EAGLE specification by Barringer et al. [3] is similar. Their specification language allows one to express temporal logic, extended regular expression, quantitative properties, and recursive language among others. The runtime verifications of quantitative properties by Thati and Rosu [19] and by Barringer et al. [3] implement only the event-driven evaluation, not the time-driven one. While the works by Mok and Liu [17] and Thati and Rosu [19] provide no support for probabilistic properties, the work by Barringer et al. [3] allows probabilistic properties but its evaluation is not based on statistical analysis.

Other runtime checking of quantitative properties are studied by Jayaputera et al. [8] and by Kristoffersen et al. [13]. Jayaputera et al. [8] provides quantitative and probabilistic property checking using Windows Management Instrumentation and .NET. The work by Kristoffersen et al. [13] is similar to Thati and Rosu [19]. To evaluate properties, they also transform timing properties based on Timed-LTL [1] into a canonical form called disjunctive normalized equation. Both of these works take a different approach. They evaluate properties every time step. In practice, predicates do not usually change every time step, and therefore, it is considered redundant and inefficient. Kristoffersen et al. [12]'s following work improves and allows both event-driven and time-driven evaluations. Their timers however are set and run by a verifier. Assuming that their implementation is done on a single machine, it is acceptable. However, if the program and the verifier reside on different machines, their timer might not be correct because clock speed on different machines can be different. The work by Kristoffersen et al. [13, 12] focuses only the quantitative properties but not probabilistic properties. The work by Jayaputera et al. [8] allows probabilistic properties but its evaluation is not based on statistical analysis.

There are works for probabilistic model checking using statistical analysis [23, 18] that are closely related to our probabilistic property checking based on statistics. The general idea is to use a simulator to generate different execution paths and analyze them using statistical hypothesis testing. Younes and Simmons [23] simulate execution paths and apply sequential hypothesis testing to analyze data. Their work is very similar to ours because they use sequential hypothesis testing. However, they can control the simulated execution paths while we observe online executions that cannot be controlled. In addition, because they can control data, they can ensure that their decision is within a given error bound. In our case, when a system somehow terminates before we have enough samples, we cannot assure the error bound but return the decision along with the probability of errors. Sen et al. [18] present a similar method except that they assume the generated sample paths cannot be controlled and their method is not sequential hypothesis testing. Because their number of observations is fixed and uncontrolled, they cannot assure an error bound and have to return their decision with the probability of errors. In our case, if the answer is undecided, we can still wait for new samples except in the case that the system terminates early, which we return answers with the probability of errors. The two works also assume prior knowledge of a system model while we assume no prior knowledge of such model.

6 Conclusion

The contributions of the paper are twofold. First, we extend MaC language to support quantitative and probabilistic properties. Second, we efficiently implement time-driven evaluation for quantitative properties. Probabilistic properties are evaluated using statistical analysis testing to provide confidence interval. The experiments show that RT-MaC poses minimal overheads to the system being verified. Future works include using first-order temporal logics to provide better expressiveness in addition to the propositional one used in this paper. We also would like to port our current implementation from Java to Real-Time Java for more accurate monitoring and checking real-time properties.

References

[1] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41:181–204, 1994.

[2] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *Proc. of the 10th International Conference on Concurrency Theory*, pages 146–161. Springer-Verlag, 1999.

[3] 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.

[4] W. G. Cochran. *Sampling Techniques*. Wiley, 1963.

[5] D. Drusinsky. Monitoring temporal logic specifications combined with time series constraints. *Journal of Universal Computer Science*, 9(11), November 2003.

[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] R. V. Hogg and A. T. Craig. *Introduction to Mathematical Statistics*. Macmillan, New York, NY, 1978.

[8] 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.

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

[10] 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, March 2004.

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

[12] K. J. Kristoffersen, C. Pedersen, and H. R. Anderson. Event-based runtime checking of Timed LTL. Technical report, IT University of Copenhagen, Nov 2003.

[13] 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.

[14] 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.

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

[16] A. K. Mok, C. G. Lee, H. Woo, and P. Konana. Monitoring of timing constraints on time intervals. In *IEEE Real-Time Systems Symposium*, 2002.

[17] 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*, 1997.

[18] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *Proc. of the 16th International Conference on Computer Aided Verification*, 2004.

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

[20] A. Wald. *Sequential Analysis*. Wiley, 1947.

[21] H. M. Walker and J. Lev. *Statistical Inference*. Holt, Rinehart and Winston, 1953.

[22] H. L. S. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking: An empirical study. In *Proc. of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2004.

[23] H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *Proc. of the 14th International Conference on Computer Aided Verification*, volume 2404 of *LNCS*. Springer, 2002.